Collagories of Coalgebras

CoalgebraCollagories-0.1

Wolfram Kahl

kahl@cas.mcmaster.ca

Department of Computing and Software, McMaster University
Hamilton, Ontario, Canada L8S 4K1

16 January 2017

Abstract

We present the Agda theories implementing a generic construction of categories, allegories, and collagories of coalgebras over arbitrary appropriate base categories, respectively allegories, collagories.

This research is supported by the National Science and Engineering Research Council (NSERC), Canada

Contents

Chapter 1

Introduction

In the context of the algebraic approach to graph transformation, graph structures have traditionally been presented as unary algebras Löwe (1990); Corradini et al. (1997). However, as such they are the intersection between algebras and coalgebras, and the current development is a first exploration of using more general coalgebras to model advanced graph features, including symbolic attribution and successor lists of term graphs.

We formalise our approach using the dependently-typed programming language (and proof checker) Agda2 (Norell, 2007; Danielsson et al., 2013), which allows us to integrate programs and their correctness proof in a single source language. We base our development on the the category formalisations of (Kahl, 2014), see also (Kahl, 2011, 2012).

1.1 Overview

The following chapters are in top-down sequence.

Each section of these chapters is the document view of a literate Agda module file, processed by `lhs2TeX -agda`. The modules making up the section in each chapter are in bottom-up sequence.

Chapter 2 contains the definition of coalgebras, and the constructions of categories, OCCs, allegories, collagories, up to division allegories and Kleene collagories of coalgebras, as well as theorems about tabulations, cotabulations and direct products in these collagories.

Chapter 3 contains definitions and properties of relators, since these are not yet present in RATH-Agda-2.2.

Chapter 4 contains additional basic material that will eventually be moved into a future version of the RATH-Agda library.

The Agda source code for this development is available on-line at the following URL:

`http://relmics.mcmaster.ca/RATH-Agda/`

1.2 CoalgebraCollagoriesAll

This module only serves as a single entry point pulling in all coalgebra collagories modules.

```
module CoalgebraCollagoriesAll where
import Categoric.Coalgebra.DivAllegory
import Categoric.Coalgebra.KleeneCollagory
import Categoric.Coalgebra.Cotabulation
import Categoric.Coalgebra.Tabulation
```

```
import Categoric.Coalgebra.DirectProduct
import Categoric.Relator.DirectSum
import Categoric.Relator.Type
import Categoric.Relator.Cotype
```

Chapter 2

Coalgebras — Definition, Allegories, Collagories

2.1 Categoric.Coalgebra.OCC

```
open import RATH.Level
open import RATH.Data.Product using (proj₁)
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
open import Categoric.Category
open import Categoric.OCC
open import Categoric.Relator.OCC
open import Categoric.Relator.OCC.Retract
open import Categoric.Functor
open import Categoric.Functor.Retract
open import Categoric.IdOp


module Categoric.Coalgebra.OCC
  {i j k₁ k₂ : Level} {Obj : Set i} {𝒞 : OCC j k₁ k₂ Obj}
  where
open OCC 𝒞


record Coalgebra (ℛ : Relator 𝒞 𝒞) : Set (i ⊍ j ⊍ k₂) where
  private module ℛ = Relator ℛ
  field Carrier : Obj
        op : MappingI Carrier (ℛ.obj Carrier)
  open MappingI op public renaming
    (mor to op₀
    ; prf        to op-isMappingI
    ; isMapping to op-isMapping
    ; univalentI  to op-univalentI
    ; univalent   to op-univalent
    ; totalI      to op-totalI
    ; total       to op-total
    )

module _ {ℛ : Relator 𝒞 𝒞} where
  private
    module ℛ = Relator ℛ
```

```
  record CARelMor (A B : Coalgebra 𝓡) : Set (j ⊍ k₂) where
    private
      open module A = Coalgebra A using () renaming (Carrier to A₀)
      open module B = Coalgebra B using () renaming (Carrier to B₀)
    field mor : Mor A₀ B₀
          commutes : mor ⨾ B.op₀ ⊑ A.op₀ ⨾ 𝓡.mor mor
    commutesL˘ : mor ˘ ⨾ A.op₀ ⊑ B.op₀ ⨾ 𝓡.mor mor ˘
    commutesL˘ = mappingHom-to-L A.op-isMapping B.op-isMapping commutes
    commutesL : A.op₀ ˘ ⨾ mor ⊑ 𝓡.mor mor ⨾ B.op₀ ˘
    commutesL = ˘⨾-˘ ⟨≈˘⊑⟩ ˘-monotone commutesL˘ ⟨⊑≈⟩ ⨾˘-˘
    commutesL-⊒ : IsSurjective mor → IsInjective (𝓡.mor mor)
                → 𝓡.mor mor ⨾ B.op₀ ˘ ⊑ A.op₀ ˘ ⨾ mor
    commutesL-⊒ mor-surj 𝓡mor-inj = ⊑-begin
        𝓡.mor mor ⨾ B.op₀ ˘
      ⊑⟨ ⨾-monotone₂ (˘-monotone (proj₁ mor-surj ⟨⊑≈⟩ ⨾-assoc)) ⟩
        𝓡.mor mor ⨾ (mor ˘ ⨾ mor ⨾ B.op₀) ˘
      ⊑⟨ ⨾-monotone₂ (˘-monotone (⨾-monotone₂ commutes) ⟨⊑≈⟩ ˘⨾⨾-˘) ⟩
        𝓡.mor mor ⨾ 𝓡.mor mor ˘ ⨾ A.op₀ ˘ ⨾ mor
      ⊑⟨ ⨾-assocL ⟨≈⊑⟩ proj₁ 𝓡mor-inj ⟩
        A.op₀ ˘ ⨾ mor
      □
    commutes-𝓡 : 𝓡.mor mor ⨾ 𝓡.mor B.op₀ ⊑ 𝓡.mor (A.op₀ ⨾ 𝓡.mor mor)
    commutes-𝓡 = ⊑-begin
        𝓡.mor mor ⨾ 𝓡.mor B.op₀
      ≈˘⟨ 𝓡.mor-⨾ ⟩
        𝓡.mor (mor ⨾ B.op₀)
      ⊑⟨ 𝓡.monotone commutes ⟩
        𝓡.mor (A.op₀ ⨾ 𝓡.mor mor)
      □


CA-Id : {A : Coalgebra 𝓡} → CARelMor A A
CA-Id {A} = record
  {mor = Id {A.Carrier}
  ;commutes = ⊑-begin
      Id ⨾ A.op₀
    ≈⟨ leftId ⟩
      A.op₀
    ≈˘⟨ rightId ⟩
      A.op₀ ⨾ Id
    ≈˘⟨ ⨾-cong₂ 𝓡.mor-Id ⟩
      A.op₀ ⨾ 𝓡.mor Id
    □
  }
  where
    module A = Coalgebra A


module CAMor-Comp {A B C : Coalgebra 𝓡}
                  (F : CARelMor A B) (G : CARelMor B C) where
  module A = Coalgebra A
  module B = Coalgebra B
  module C = Coalgebra C
  module F = CARelMor F
  module G = CARelMor G
  H = F.mor ⨾ G.mor
```

```
  comp : CARelMor A C
  comp = record
    {mor = F.mor ⨾ G.mor
    ;commutes = ⊑-begin
        (F.mor ⨾ G.mor) ⨾ C.op₀
      ⊑⟨ ⨾-assoc ⟨≈⊑⟩ ⨾-monotone₂ G.commutes ⟩
        F.mor ⨾ B.op₀ ⨾ ℛ.mor G.mor
      ⊑⟨ ⨾-monotone₁&₂₁ F.commutes ⟩
        A.op₀ ⨾ ℛ.mor F.mor ⨾ ℛ.mor G.mor
      ≈˘⟨ ⨾-cong₂ ℛ.mor-⨾ ⟩
        A.op₀ ⨾ ℛ.mor (F.mor ⨾ G.mor)
      □
    }
open CAMor-Comp public using (comp)

module CAMor-Conv {A B : Coalgebra ℛ} (F : CARelMor A B) where
  module A = Coalgebra A
  module B = Coalgebra B
  module F = CARelMor F
  conv : CARelMor B A
  conv = record
    {mor = F.mor ˘
    ;commutes = F.commutesL˘ ⟨⊑≈˘⟩ ⨾-cong₂ ℛ.mor-˘
    }
open CAMor-Conv public using (conv)
```

CA-OCC ℛ is the OCC of ℛ-coalgebras, and U-CA-OCC ℛ the associated forgetful relator into the underlying OCC 𝒞.

Once Agda implements such sharing, the following definition should have advantages over separate invocations of retract²OCC and retract²Relator.

```
module _ (ℛ : Relator 𝒞 𝒞) where
  open Coalgebra
  open CARelMor
  private
    module ℛcoalg = Retract²Relator 𝒞
      (λ {A} → CA-Id {ℛ} {A})
      conv
      comp
      Carrier
      mor
      ≈-refl
      ≈-refl
      ≈-refl
  CA-OCC : OCC (j ⊍ k₂) k₁ k₂ (Coalgebra ℛ)
  CA-OCC = ℛcoalg.𝒞₂
  U-CA-OCC : Relator CA-OCC 𝒞
  U-CA-OCC = ℛcoalg.retract²Relator
```

2.2 Categoric.Coalgebra.Allegory

```
open import RATH.Level
```

```
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
open import Categoric.Category
open import Categoric.Allegory
open import Categoric.Relator.OCC
open import Categoric.Relator.Allegory
open import Categoric.IdOp
open import Categoric.Coalgebra.OCC


module Categoric.Coalgebra.Allegory
  {i j k₁ k₂ : Level} {Obj : Set i} (𝒜 : Allegory j k₁ k₂ Obj)
  where
open Allegory 𝒜
open MeetPres 𝒜 𝒜


module _ {ℛ : Relator occ occ}
         (let private module ℛ = Relator ℛ)
         (ℛ-mor-⊓-⊒ : PreservesMeets-⊒ ℛ)
  where
  ℛ-mor-⊓ : PreservesMeets ℛ
  ℛ-mor-⊓ = PreservesMeets-from-⊒ ℛ ℛ-mor-⊓-⊒


module CAMor-Meet {A B : Coalgebra ℛ} (F G : CARelMor A B) where
  module A = Coalgebra A
  module B = Coalgebra B
  module F = CARelMor F
  module G = CARelMor G


  meetMor : CARelMor A B
  meetMor = record
    {mor = F.mor ⊓ G.mor
    ; commutes = ⊑-begin
        (F.mor ⊓ G.mor) ⨾ B.op₀
      ⊑⟨ ⨾-⊓-subdistribL ⟨⊑⊑⟩ ⊓-monotone F.commutes G.commutes ⟩
        A.op₀ ⨾ ℛ.mor F.mor ⊓ A.op₀ ⨾ ℛ.mor G.mor
      ≈˘⟨ ⨾-⊓-distribR A.op-univalent ⟩
        A.op₀ ⨾ (ℛ.mor F.mor ⊓ ℛ.mor G.mor)
      ⊑⟨ ⨾-monotone₂ ℛ-mor-⊓-⊒ ⟩
        A.op₀ ⨾ ℛ.mor (F.mor ⊓ G.mor)
      □
    }
open CAMor-Meet public using (meetMor)


CA-Allegory : (ℛ : Relator occ occ)
              (let module ℛ = Relator ℛ)
              (ℛ-mor-⊓-⊒ : {A B : Obj} {R S : Mor A B}
                         → ℛ.mor R ⊓ ℛ.mor S ⊑ ℛ.mor (R ⊓ S))
            → Allegory (j ⊍ k₂) k₁ k₂ (Coalgebra ℛ)
CA-Allegory ℛ ℛ-mor-⊓-⊒ = retract²Allegory 𝒜
  (λ {A} → CA-Id {ℛ = ℛ} {A})
  conv
  comp
  (meetMor ℛ-mor-⊓-⊒)
```

```
  Carrier
  mor
  ≈-refl
  ≈-refl
  ≈-refl
  ≈-refl
  where open Coalgebra; open CARelMor

module _ where
  private module ℛ = Relator (Identity occ)
  Identity-mor-⊓-⊒ : PreservesMeets-⊒ (Identity occ)
  Identity-mor-⊓-⊒ {A} {B} {R} {S} = ⊑-refl

module _ (A : Obj) where
  private module ℛ = Relator (Const occ A)
  Const-mor-⊓-⊒ : PreservesMeets-⊒ (Const occ A)
  Const-mor-⊓-⊒ {A} {B} {R} {S} = ⊑-reflexive ⊓-idempotent

open import Categoric.TopMor
module _ (hasTopMors : HasTopMors orderedSemigroupoid) where
  open HasTopMors hasTopMors
  open import Categoric.Allegory.TopMor 𝒜 hasTopMors
  open import Categoric.Coalgebra.TopMor occ hasTopMors
  module _ where
    private module ℛ = Relator (Identity occ)
    Identity-creates-⊤ : Creates-⊤ (Identity occ)
    Identity-creates-⊤ A B = ⊑-begin
          ⊤ ⨾ B.op₀
        ⊑⟨ ⊑-⊤ ⟨⊑≈˘⟩ leftId ⟩
          Id ⨾ ⊤
        ≈⟨ ⨾-cong₁ (total-dom A.op-total) ⟨≈˘≈⟩ dom⨾⊤ ⟩
          A.op₀ ⨾ ⊤
        □
      where
        module A = Coalgebra A
        module B = Coalgebra B
```

2.3 Categoric.Coalgebra.Collagory

```
open import RATH.Level
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
open import Categoric.Category
open import Categoric.Allegory
open import Categoric.Collagory
open import Categoric.Relator.OCC
open import Categoric.Relator.Allegory
open import Categoric.Relator.JoinOp
open import Categoric.IdOp
open import Categoric.Coalgebra.OCC

module Categoric.Coalgebra.Collagory
  {i j k₁ k₂ : Level} {Obj : Set i} (𝒞 : Collagory j k₁ k₂ Obj)
```

```
  where
open Collagory 𝒞
open MeetPres allegory allegory
open JoinPres occ occ joinOp joinOp
open import Categoric.Coalgebra.Allegory allegory

module _ {ℛ : Relator occ occ}
         (let private module ℛ = Relator ℛ)
  where

  module CAMor-Join {A B : Coalgebra ℛ} (F G : CARelMor A B) where
    module A = Coalgebra A
    module B = Coalgebra B
    module F = CARelMor F
    module G = CARelMor G

    joinMor : CARelMor A B
    joinMor = record
      {mor = F.mor ⊔ G.mor
      ;commutes = ⊑-begin
          (F.mor ⊔ G.mor) ⨾ B.op₀
        ⊑⟨ ⨾-⊔-distribL ⟨≈⊑⟩ ⊔-monotone F.commutes G.commutes ⟩
          A.op₀ ⨾ ℛ.mor F.mor ⊔ A.op₀ ⨾ ℛ.mor G.mor
        ≈˘⟨ ⨾-⊔-distribR ⟩
          A.op₀ ⨾ (ℛ.mor F.mor ⊔ ℛ.mor G.mor)
        ⊑⟨ ⨾-monotone₂ (preservesJoins-⊒ ℛ) ⟩
          A.op₀ ⨾ ℛ.mor (F.mor ⊔ G.mor)
        □
      }
  open CAMor-Join public using (joinMor)

CA-Collagory : (ℛ : Relator occ occ)
               (let module ℛ = Relator ℛ)
               (ℛ-mor-⊓-⊒ : PreservesMeets-⊒ ℛ)
  →            Collagory (j ⊍ k₂) k₁ k₂ (Coalgebra ℛ)
CA-Collagory ℛ ℛ-mor-⊓-⊒ = retract²Collagory 𝒞
  (λ {A} → CA-Id {ℛ = ℛ} {A})
  conv
  comp
  (meetMor ℛ-mor-⊓-⊒)
  joinMor
  Carrier
  mor
  ≈-refl
  ≈-refl
  ≈-refl
  ≈-refl
  ≈-refl
  where open Coalgebra; open CARelMor

module _ where
  private module ℛ = Relator (Identity occ)
  Identity-mor-⊔-⊑ : PreservesJoins-⊑ (Identity occ)
  Identity-mor-⊔-⊑ {A} {B} {R} {S} = ⊑-refl
```

```
module _ (A : Obj) where
  private module ℛ = Relator (Const occ A)
  Const-mor-⊔-⊑ : PreservesJoins-⊑ (Const occ A)
  Const-mor-⊔-⊑ {A} {B} {R} {S} = ⊑-reflexive′ ⊔-idempotent
```

2.4 Categoric.Coalgebra.DistrAllegory

```
open import RATH.Level
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
open import Categoric.Category
open import Categoric.Allegory
open import Categoric.DistrAllegory
open import Categoric.Relator.OCC
open import Categoric.Relator.Allegory
open import Categoric.Relator.JoinOp
open import Categoric.IdOp
open import Categoric.Coalgebra.OCC


module Categoric.Coalgebra.DistrAllegory
  {i j k₁ k₂ : Level} {Obj : Set i} (𝒜 : DistrAllegory j k₁ k₂ Obj)
  where
open DistrAllegory 𝒜
open MeetPres allegory allegory
open JoinPres occ occ joinOp joinOp
open import Categoric.Coalgebra.Allegory allegory using (meetMor)
open import Categoric.Coalgebra.Collagory collagory using (joinMor)


module _ {ℛ : Relator occ occ}
         (let private module ℛ = Relator ℛ)
  where

  module CAMor-Bot {A B : Coalgebra ℛ} where
    module A = Coalgebra A
    module B = Coalgebra B

    CAbot : CARelMor A B
    CAbot = record
      {mor = ⊥
      ;commutes = ⊑-begin
            ⊥ ⨾ B.op₀
          ≈⟨ leftZero ⟩
            ⊥
          ⊑⟨ ⊥-⊑ ⟩
            A.op₀ ⨾ ℛ.mor ⊥
          □
      }
  open CAMor-Bot public using (CAbot)

CA-DistrAllegory : (ℛ : Relator occ occ)
    (let module ℛ = Relator ℛ)
```

```
    (ℛ-mor-⊓-⊒ : PreservesMeets-⊒ ℛ)
  → DistrAllegory (j ⊍ k₂) k₁ k₂ (Coalgebra ℛ)
CA-DistrAllegory ℛ ℛ-mor-⊓-⊒ = retract²DistrAllegory 𝒜
  (λ {A} → CA-Id {ℛ = ℛ} {A})
  conv
  comp
  CAbot
  (meetMor ℛ-mor-⊓-⊒)
  joinMor
  Carrier
  mor
  ≈-refl
  ≈-refl
  ≈-refl
  ≈-refl
  ≈-refl
  ≈-refl
  where open Coalgebra; open CARelMor
```

2.5 Categoric.Coalgebra.DivAllegory

```
open import RATH.Level
open import RATH.Data.Product using (proj₁; proj₂)
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
open import Categoric.Category
open import Categoric.Allegory
open import Categoric.DivAllegory
open import Categoric.Relator.OCC
open import Categoric.Relator.Allegory
open import Categoric.Relator.JoinOp
open import Categoric.Relator.Residuals
open import Categoric.IdOp
open import Categoric.Coalgebra.OCC

module Categoric.Coalgebra.DivAllegory
  {i j k₁ k₂ : Level} {Obj : Set i} (𝒜 : DivAllegory j k₁ k₂ Obj)
  where
open DivAllegory 𝒜
open MeetPres allegory allegory
open JoinPres occ occ joinOp joinOp
open RResPres occ occ rightResOp rightResOp
open import Categoric.Coalgebra.Allegory allegory using (meetMor)
open import Categoric.Coalgebra.Collagory collagory using (joinMor)
open import Categoric.Coalgebra.DistrAllegory distrAllegory using (CAbot)

module _ {ℛ : Relator occ occ}
         (let private module ℛ = Relator ℛ)
         (ℛ-preserves-\-⊒ : PreservesRRes-⊒ ℛ)
  where

  module CAMor-RRes {A B C : Coalgebra ℛ}
                    (Q : CARelMor A B) (S : CARelMor A C) where
```

```
    module A  =  Coalgebra A
    module B  =  Coalgebra B
    module C  =  Coalgebra C
    module Q  =  CARelMor Q
    module S  =  CARelMor S

    CArres : IsSurjective Q.mor → IsInjective (ℛ.mor Q.mor)
           → CARelMor B C
    CArres Qmor-surj ℛQmor-inj  =  record
      {mor  =  Q.mor \ S.mor
      ;commutes  =  ⊑-begin
           (Q.mor \ S.mor) ⨾ C.op₀
        ⊑⟨ \-outer-⨾ ⟩
           Q.mor \ (S.mor ⨾ C.op₀)
        ⊑⟨ \-monotone S.commutes ⟩
           Q.mor \ (A.op₀ ⨾ ℛ.mor S.mor)
        ≈˘⟨ \-flip-M A.op-isMapping ⟩
           (A.op₀ ˘ ⨾ Q.mor) \ ℛ.mor S.mor
        ⊑⟨ \-antitone (Q.commutesL-⊒ Qmor-surj ℛQmor-inj) ⟩
           (ℛ.mor Q.mor ⨾ B.op₀ ˘) \ ℛ.mor S.mor
        ≈˘⟨ \-inner-⨾ B.op-isMapping ⟩
           B.op₀ ⨾ (ℛ.mor Q.mor \ ℛ.mor S.mor)
        ⊑⟨ ⨾-monotone₂ ℛ-preserves-\-⊒ ⟩
           B.op₀ ⨾ ℛ.mor (Q.mor \ S.mor)
        □
      }
  open CAMor-RRes public using (CArres)


module _ where
  private
    ℛ  =  Identity occ
    module ℛ  =  Relator ℛ
  Identity-preservesRRes-⊒ : {A B C : Obj} {Q : Mor A B} {S : Mor A C}
                           → ℛ.mor Q \ ℛ.mor S ⊑ ℛ.mor (Q \ S)
  Identity-preservesRRes-⊒ {A} {B} {C} {Q} {S}  =  ⊑-refl


module _ (X : Obj) where
  private
    ℛ  =  Const occ X
    module ℛ  =  Relator ℛ
  Const-preservesRRes-⊒ : {A B C : Obj} {Q : Mor A B} {S : Mor A C}
                        → ℛ.mor Q \ ℛ.mor S ⊑ ℛ.mor (Q \ S)
  Const-preservesRRes-⊒ {A} {B} {C} {Q} {S}  =  ⊑-reflexive Id-\
```

2.6 Categoric.Coalgebra.KleeneCollagory

```
open import RATH.Level
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
open import Categoric.Category
open import Categoric.Allegory
open import Categoric.Collagory
```

```
open import Categoric.KleeneCollagory
open import Categoric.Relator.OCC
open import Categoric.Relator.Allegory
open import Categoric.Relator.JoinOp
open import Categoric.IdOp
open import Categoric.Coalgebra.OCC


module Categoric.Coalgebra.KleeneCollagory
  {i j k₁ k₂ : Level} {Obj : Set i} (𝒞 : KleeneCollagory j k₁ k₂ Obj)
  where
open KleeneCollagory 𝒞
open MeetPres allegory allegory
open JoinPres occ occ joinOp joinOp
open import Categoric.Coalgebra.Allegory allegory
open import Categoric.Coalgebra.Collagory collagory


module _ {ℛ : Relator occ occ}
         (let private module ℛ = Relator ℛ)
  where
  preservesStar-⊒ : {A : Obj} {R : Mor A A} → ℛ.mor R * ⊑ ℛ.mor (R *)
  preservesStar-⊒ {A} {R} = ⊑-begin
      ℛ.mor R *
    ≈⟨ leftId ⟨≈˘≈˘⟩ ⨾-cong₁ ℛ.mor-Id ⟩
      ℛ.mor Id ⨾ ℛ.mor R *
    ⊑⟨ ⨾-monotone₁ (ℛ.monotone *-isReflexive) ⟩
      ℛ.mor (R *) ⨾ ℛ.mor R *
    ⊑⟨ *-rightInd (⊑-begin
         ℛ.mor (R *) ⨾ ℛ.mor R
      ≈˘⟨ ℛ.mor-⨾ ⟩
         ℛ.mor (R * ⨾ R)
      ⊑⟨ ℛ.monotone *-stepR ⟩
         ℛ.mor (R *)
      □) ⟩
      ℛ.mor (R *)
    □


module CAMor-Star {A : Coalgebra ℛ} (F : CARelMor A A) where
  module A = Coalgebra A
  module F = CARelMor F


  starMor : CARelMor A A
  starMor = record
    {mor = F.mor *
    ;commutes = ⊑-begin
        F.mor * ⨾ A.op₀
      ⊑⟨ *-⨾-commute-⊑ F.commutes ⟩
        A.op₀ ⨾ ℛ.mor F.mor *
      ⊑⟨ ⨾-monotone₂ preservesStar-⊒ ⟩
        A.op₀ ⨾ ℛ.mor (F.mor *)
      □
    }
open CAMor-Star public using (starMor)
```

```
CA-KleeneCollagory : (ℛ : Relator occ occ)
    (let module ℛ = Relator ℛ)
    (ℛ-mor-⊓-⊒ : PreservesMeets-⊒ ℛ)
  → KleeneCollagory (j ⊍ k₂) k₁ k₂ (Coalgebra ℛ)
CA-KleeneCollagory ℛ ℛ-mor-⊓-⊒ = retract²KleeneCollagory 𝒞
  (λ {A} → CA-Id {ℛ = ℛ} {A})
  conv
  comp
  (meetMor ℛ-mor-⊓-⊒)
  joinMor
  starMor
  Carrier
  mor
  ≈-refl
  ≈-refl
  ≈-refl
  ≈-refl
  ≈-refl
  ≈-refl
  where open Coalgebra; open CARelMor
```

2.7 Categoric.Coalgebra.Cotabulation

```
open import RATH.Level
open import RATH.Data.Product using (_,_)
open import Categoric.Collagory using (Collagory; module Collagory)
open import Categoric.Cotabulation using
  (Cotabulation; module Cotabulation; module DefaultCofork
  ; HasCotabulations; module HasCotabulations)
open import Categoric.Relator.OCC
open import Categoric.Relator.Allegory using (module MeetPres)
open import Categoric.Coalgebra.OCC

module Categoric.Coalgebra.Cotabulation
  {i j k₁ k₂ : Level} {Obj : Set i}
  (𝒞 : Collagory j k₁ k₂ Obj)
  (let open Collagory 𝒞)
  (hasCotabulations : HasCotabulations uslcc)
  where
open MeetPres allegory allegory using (PreservesMeets-⊒)
open import Categoric.Coalgebra.Collagory 𝒞
open HasCotabulations hasCotabulations using (cotabulate)

module _ {ℛ : Relator occ occ}
         (ℛ-mor-⊓-⊒ : PreservesMeets-⊒ ℛ)
  where
  private
    module ℛ = Relator ℛ
    CA-𝒞 = CA-Collagory ℛ ℛ-mor-⊓-⊒
    module CA-𝒞 = Collagory CA-𝒞

  module CAMor-Cotab {A B : Coalgebra ℛ} (R : CARelMor A B)
                     (cotab : Cotabulation uslcc (CARelMor.mor R)) where
```

```
private
  module A = Coalgebra A
  module B = Coalgebra B
  module R = CARelMor R
  open Cotabulation cotab
    renaming (obj to D₀; left to ι; right to κ)
  module ι = MappingI leftM
  module κ = MappingI rightM
  D-op : Mor D₀ (𝓡.obj D₀)
  D-op = cofork (A.op₀ ⨾ 𝓡.mor ι) (B.op₀ ⨾ 𝓡.mor κ)
  subfactorLeft′ : R.mor ⨾ B.op₀ ⨾ 𝓡.mor κ ⊑ A.op₀ ⨾ 𝓡.mor ι
  subfactorLeft′ = ⊑-begin
        R.mor ⨾ B.op₀ ⨾ 𝓡.mor κ
     ⊑⟨ ⨾-monotone₁&₂₁ R.commutes ⟩
        A.op₀ ⨾ 𝓡.mor R.mor ⨾ 𝓡.mor κ
     ⊑⟨ ⨾-monotone₂ (𝓡.mor-⨾ ⟨≈˘⊑⟩ 𝓡.monotone subfactorLeft) ⟩
        A.op₀ ⨾ 𝓡.mor ι
     □
  subfactorRight′ : R.mor ˘ ⨾ A.op₀ ⨾ 𝓡.mor ι ⊑ B.op₀ ⨾ 𝓡.mor κ
  subfactorRight′ = ⊑-begin
        R.mor ˘ ⨾ A.op₀ ⨾ 𝓡.mor ι
     ⊑⟨ ⨾-monotone₁&₂₁ R.commutesL˘ ⟩
        B.op₀ ⨾ 𝓡.mor R.mor ˘ ⨾ 𝓡.mor ι
     ⊑⟨ ⨾-monotone₂ (⨾-cong₁ 𝓡.mor-˘ ⟨≈˘≈˘⟩ 𝓡.mor-⨾
                         ⟨≈⊑⟩ 𝓡.monotone subfactorRight) ⟩
        B.op₀ ⨾ 𝓡.mor κ
     □
  D-op-unival : IsUnivalentI D-op
  D-op-unival = cofork-univalentI subfactorLeft′ subfactorRight′
    (⨾-IsUnivalentI A.op-univalentI (𝓡.mor-IsUnivalentI ι.univalentI))
    (⨾-IsUnivalentI B.op-univalentI (𝓡.mor-IsUnivalentI κ.univalentI))
  D-op-total : IsTotalI D-op
  D-op-total = cofork-totalI subfactorLeft′ subfactorRight′
    (⨾-IsTotalI A.op-totalI (𝓡.mor-IsTotalI ι.totalI))
    (⨾-IsTotalI B.op-totalI (𝓡.mor-IsTotalI κ.totalI))
D : Coalgebra 𝓡
D = record
  {Carrier = D₀
  ; op = record {mor = D-op; prf = D-op-unival , D-op-total}
  }
module D = Coalgebra D


iota : CARelMor A D
iota = record
  {mor = ι
  ; commutes = ⊑-begin
       ι ⨾ D.op₀
    ≈⟨ left⨾cofork subfactorLeft′ subfactorRight′ ⟩
       A.op₀ ⨾ 𝓡.mor ι
    □
  }


kappa : CARelMor B D
kappa = record
```

```
      {mor = κ
      ; commutes = ⊑-begin
          κ ⨾ D.op₀
        ≈⟨ right⨾cofork subfactorLeft′ subfactorRight′ ⟩
          B.op₀ ⨾ ℛ.mor κ
        □
      }

    cotabulation : Cotabulation CA-𝒞.uslcc R
    cotabulation = record
      {obj = D
      ; left = iota
      ; right = kappa
      ; isCotabulation = record
        {commutes  = commutes
        ; jointId     = jointId
        ; leftKernel   = leftKernel
        ; rightKernel = rightKernel
          -- ; DefaultCofork CA-𝒞.uslcc iota kappa – not yet possibly in Agda-2.4.2.5
        ; cofork = Cofork.cofork
        ; cofork-def = λ {D′} {R′} {S′} → Cofork.cofork-def {D′} {R′} {S′}
        }
      }
      where module Cofork = DefaultCofork CA-𝒞.uslcc iota kappa

  open CAMor-Cotab

  CA-hasCotabulations : HasCotabulations CA-𝒞.uslcc
  CA-hasCotabulations = record
    {cotabulate = cotab
    }
    where
      cotab : {A B : Coalgebra ℛ} (R : CARelMor A B) → Cotabulation CA-𝒞.uslcc R
      cotab R = cotabulation R (cotabulate (CARelMor.mor R))
```

2.8 Categoric.Coalgebra.Tabulation

```
open import RATH.Level
open import RATH.Data.Product using (_,_; proj₁)
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
open import Categoric.Category
open import Categoric.Allegory
open import Categoric.Allegory.Tabulation
open import Categoric.Relator.OCC
open import Categoric.Relator.Allegory
open import Categoric.Relator.JoinOp
open import Categoric.IdOp
open import Categoric.Coalgebra.OCC
```

```
module Categoric.Coalgebra.Tabulation
  {i j k₁ k₂ : Level} {Obj : Set i}
  (𝒜 : Allegory j k₁ k₂ Obj)
```

```
  (hasTabulations : HasTabulations 𝒜)
  where
open Allegory 𝒜
open MeetPres 𝒜 𝒜
open import Categoric.Coalgebra.Allegory 𝒜
open HasTabulations hasTabulations using (tabulate)

module _ {ℛ : Relator occ occ}
         (let private module ℛ = Relator ℛ)
         (ℛ-mor-⊓-⊒ : PreservesMeets-⊒ ℛ)
  where
  private
    CA-𝒜 = CA-Allegory ℛ ℛ-mor-⊓-⊒

  module CAMor-Tab {A B : Coalgebra ℛ} (R : CARelMor A B) (tab : Tabulation 𝒜 (CARelMor.mor R)) where
    private
      module A = Coalgebra A
      module B = Coalgebra B
      module R = CARelMor R
      open Tabulation tab
        renaming (obj to P₀)
      P-op : Mor P₀ (ℛ.obj P₀)
      P-op = π ⨾ A.op₀ ⨾ ℛ.mor π ˘ ⊓ ρ ⨾ B.op₀ ⨾ ℛ.mor ρ ˘
      P-op-unival : IsUnivalentI P-op
      P-op-unival = ⊑-begin
          (π ⨾ A.op₀ ⨾ ℛ.mor π ˘ ⊓ ρ ⨾ B.op₀ ⨾ ℛ.mor ρ ˘) ˘ ⨾ (π ⨾ A.op₀ ⨾ ℛ.mor π ˘ ⊓ ρ ⨾ B.op₀ ⨾ ℛ.mor ρ ˘)
        ≈⟨ ⨾-cong₁ (˘-⊓-distrib ⟨≈≈⟩ ⊓-cong (⨾⨾˘-˘ ⟨≈≈⟩ ⨾-assocL) (⨾⨾˘-˘ ⟨≈≈⟩ ⨾-assocL) ⟨≈≈˘⟩ fork-def) ⟩
          (fork (ℛ.mor π ⨾ A.op₀ ˘) (ℛ.mor ρ ⨾ B.op₀ ˘)) ⨾ (π ⨾ A.op₀ ⨾ ℛ.mor π ˘ ⊓ ρ ⨾ B.op₀ ⨾ ℛ.mor ρ ˘)
        ⊑⟨ ⨾-⊓-subdistribR ⟨⊑⊑⟩ ⊓-monotone (⨾-monotone₁&₂₁ fork⨾π-⊑) (⨾-monotone₁&₂₁ fork⨾ρ-⊑) ⟩
          ℛ.mor π ⨾ A.op₀ ˘ ⨾ A.op₀ ⨾ ℛ.mor π ˘ ⊓ ℛ.mor ρ ⨾ B.op₀ ˘ ⨾ B.op₀ ⨾ ℛ.mor ρ ˘
        ⊑⟨ ⊓-monotone (⨾-monotone₂ (⨾-assocL ⟨≈⊑⟩ proj₁ A.op-univalent)) (⨾-monotone₂ (⨾-assocL ⟨≈⊑⟩ proj₁ B.op-univalent)) ⟩
          ℛ.mor π ⨾ ℛ.mor π ˘ ⊓ ℛ.mor ρ ⨾ ℛ.mor ρ ˘
        ≈⟨ ⊓-cong (⨾-cong₂ ℛ.mor-˘ ⟨≈˘≈˘⟩ ℛ.mor-⨾) (⨾-cong₂ ℛ.mor-˘ ⟨≈˘≈˘⟩ ℛ.mor-⨾) ⟩
          ℛ.mor (π ⨾ π ˘) ⊓ ℛ.mor (ρ ⨾ ρ ˘)
        ⊑⟨ ℛ-mor-⊓-⊒ ⟨⊑≈⟩ ℛ.mor-cong extensionality ⟨⊑≈⟩ ℛ.mor-Id ⟩
          Id
        □
      P-op-⨾-ℛπ : P-op ⨾ ℛ.mor π ≈ π ⨾ A.op₀ ⊓ (ρ ⨾ B.op₀) ⨾ ℛ.mor R.mor ˘
      P-op-⨾-ℛπ = ≈-begin
          P-op ⨾ ℛ.mor π
        ≈⟨ ⨾-cong₁ (⊓-cong ⨾-assocL ⨾-assocL) ⟩
          ((π ⨾ A.op₀) ⨾ ℛ.mor π ˘ ⊓ (ρ ⨾ B.op₀) ⨾ ℛ.mor ρ ˘) ⨾ ℛ.mor π
        ≈⟨ modal₂′unival (ℛ.mor-IsUnivalent π.univalent) ⟨≈˘≈⟩ ⊓-cong₂ ⨾-assoc ⟩
          π ⨾ A.op₀ ⊓ (ρ ⨾ B.op₀) ⨾ ℛ.mor ρ ˘ ⨾ ℛ.mor π
        ≈⟨ ⊓-cong₂ (⨾-cong₂ (⨾-cong₁ ℛ.mor-˘ ⟨≈˘≈˘⟩ ℛ.mor-⨾)) ⟩
          π ⨾ A.op₀ ⊓ (ρ ⨾ B.op₀) ⨾ ℛ.mor (ρ ˘ ⨾ π)
        ≈⟨ ⊓-cong₂ (⨾-cong₂ (ℛ.mor-cong ρ˘⨾π ⟨≈≈⟩ ℛ.mor-˘)) ⟩
          π ⨾ A.op₀ ⊓ (ρ ⨾ B.op₀) ⨾ ℛ.mor R.mor ˘
        □
      P-op-⨾-ℛρ : P-op ⨾ ℛ.mor ρ ≈ (π ⨾ A.op₀) ⨾ ℛ.mor R.mor ⊓ ρ ⨾ B.op₀
      P-op-⨾-ℛρ = ≈-begin
          P-op ⨾ ℛ.mor ρ
        ≈⟨ ⨾-cong₁ (⊓-cong ⨾-assocL ⨾-assocL) ⟩
          ((π ⨾ A.op₀) ⨾ ℛ.mor π ˘ ⊓ (ρ ⨾ B.op₀) ⨾ ℛ.mor ρ ˘) ⨾ ℛ.mor ρ
        ≈⟨ modal₂unival (ℛ.mor-IsUnivalent ρ.univalent) ⟨≈˘≈⟩ ⊓-cong₁ ⨾-assoc ⟩
          (π ⨾ A.op₀) ⨾ ℛ.mor π ˘ ⨾ ℛ.mor ρ ⊓ ρ ⨾ B.op₀
```

```
      ≈⟨ ⊓-cong_1 (⨾-cong_2 (⨾-cong_1 𝓡.mor-˘ ⟨≈˘≈˘⟩ 𝓡.mor-⨾)) ⟩
        (π ⨾ A.op_0) ⨾ 𝓡.mor (π ˘ ⨾ ρ) ⊓ ρ ⨾ B.op_0
      ≈⟨ ⊓-cong_1 (⨾-cong_2 (𝓡.mor-cong π˘⨾ρ)) ⟩
        (π ⨾ A.op_0) ⨾ 𝓡.mor R.mor ⊓ ρ ⨾ B.op_0
      □
    lemma_1 : Id ⊑ ((ρ ⨾ B.op_0) ⨾ 𝓡.mor R.mor ˘ ⨾ A.op_0 ˘) ⨾ π ˘
    lemma_1 = swap-⨾-⊑-total π.total (leftId ⟨≈⊑⟩ swap-⨾-⊑-total A.op-total (⊑-begin
        π ⨾ A.op_0
      ⊑⟨ ⨾-monotone_1 (proj_1 ρ.total ⟨⊑≈⟩ ⨾-assoc) ⟩
        (ρ ⨾ ρ ˘ ⨾ π) ⨾ A.op_0
      ≈⟨ ⨾-cong_12 ρ˘⨾π ⟩
        (ρ ⨾ R.mor ˘) ⨾ A.op_0
      ⊑⟨ ⨾-monotone_12&_2 R.commutesL˘ ⟩
        (ρ ⨾ B.op_0) ⨾ 𝓡.mor R.mor ˘
      □)⟨⊑≈⟩ ⨾-assoc)
    lemma_2 : Id ⊑ ((π ⨾ A.op_0) ⨾ 𝓡.mor R.mor ⨾ B.op_0 ˘) ⨾ ρ ˘
    lemma_2 = Id˘ ⟨≈˘⊑⟩ ˘-monotone lemma_1 ⟨⊑≈⟩
           (⨾˘-˘ ⟨≈≈⟩ ⨾-cong_2 (⨾⨾˘-˘ ⟨≈≈⟩ ⨾-cong_2 (⨾-cong ˘˘ ⨾-˘))
                ⟨≈≈⟩ (⨾-assocL ⟨≈≈⟩ (⨾-cong_2 ⨾-assocL ⟨≈≈⟩ ⨾-assocL)))
    P-op-total : IsTotalI P-op
    P-op-total = ⊑-begin
        Id
      ⊑⟨ ⊓-universal (⊓-universal ⊑-refl lemma_1 ⟨⊑⊑⟩ modal_2˘′′)
                     (⊓-universal lemma_2 ⊑-refl ⟨⊑⊑⟩ modal_2˘) ⟩
        (Id ⨾ π ⊓ (ρ ⨾ B.op_0) ⨾ 𝓡.mor R.mor ˘ ⨾ A.op_0 ˘) ⨾ π ˘ ⊓
        ((π ⨾ A.op_0) ⨾ 𝓡.mor R.mor ⨾ B.op_0 ˘ ⊓ Id ⨾ ρ) ⨾ ρ ˘
      ⊑⟨ ⊓-monotone (⨾-monotone_1 (⊓-cong leftId ⨾-assocL ⟨≈⊑⟩ modal_2˘′′) ⟨⊑≈⟩ ⨾-assoc)
                     (⨾-monotone_1 (⊓-cong ⨾-assocL leftId ⟨≈⊑⟩ modal_2˘) ⟨⊑≈⟩ ⨾-assoc) ⟩
        (π ⨾ A.op_0 ⊓ (ρ ⨾ B.op_0) ⨾ 𝓡.mor R.mor ˘) ⨾ A.op_0 ˘ ⨾ π ˘ ⊓
        ((π ⨾ A.op_0) ⨾ 𝓡.mor R.mor ⊓ ρ ⨾ B.op_0) ⨾ B.op_0 ˘ ⨾ ρ ˘
      ≈˘⟨ ⊓-cong (⨾-assocL ⟨≈≈⟩ ⨾-cong_1 P-op-⨾-𝓡π) (⨾-assocL ⟨≈≈⟩ ⨾-cong_1 P-op-⨾-𝓡ρ) ⟩
        P-op ⨾ 𝓡.mor π ⨾ A.op_0 ˘ ⨾ π ˘  ⊓
        P-op ⨾ 𝓡.mor ρ ⨾ B.op_0 ˘ ⨾ ρ ˘
      ≈˘⟨ ⨾-⊓-distribR (IsUnivalent-from-I P-op-unival) ⟩
        P-op ⨾ (𝓡.mor π ⨾ A.op_0 ˘ ⨾ π ˘ ⊓ 𝓡.mor ρ ⨾ B.op_0 ˘ ⨾ ρ ˘)
      ≈˘⟨ ⨾-cong_2 (˘-⊓-distrib ⟨≈≈⟩ ⊓-cong ⨾⨾˘-˘ ⨾⨾˘-˘) ⟩
        P-op ⨾ (π ⨾ A.op_0 ⨾ 𝓡.mor π ˘ ⊓ ρ ⨾ B.op_0 ⨾ 𝓡.mor ρ ˘) ˘
      □
  P : Coalgebra 𝓡
  P = record
    {Carrier = P_0
    ;op = record {mor = P-op; prf = P-op-unival , P-op-total}
    }
  module P = Coalgebra P


  pi : CARelMor P A
  pi = record
    {mor = π
    ;commutes = ⊑-begin
        π ⨾ A.op_0
      ≈˘⟨ ⊑-to-⊓_1 (⊑-begin
          π ⨾ A.op_0
        ⊑⟨ proj_1 ρ.total ⟨⊑≈⟩ ⨾-_22assoc_121 ⟩
          ρ ⨾ (ρ ˘ ⨾ π) ⨾ A.op_0
        ≈⟨ ⨾-cong_21 ρ˘⨾π ⟩
          ρ ⨾ R.mor ˘ ⨾ A.op_0
```

```
          ⊑⟨ ⨾-monotone2 R.commutesL˘ ⟨⊑≈⟩ ⨾-assocL ⟩
            (ρ ⨾ B.op0) ⨾ ℛ.mor R.mor ˘
          □) ⟩
          π ⨾ A.op0 ⊓ (ρ ⨾ B.op0) ⨾ ℛ.mor R.mor ˘
        ≈˘⟨ P-op-⨾-ℛπ ⟩
          P.op0 ⨾ ℛ.mor π
        □
      }

  rho : CARelMor P B
  rho = record
    {mor = ρ
    ;commutes = ⊑-begin
          ρ ⨾ B.op0
        ≈˘⟨ ⊑-to-⊓2 (⊑-begin
            ρ ⨾ B.op0
          ⊑⟨ proj1 π.total ⟨⊑≈⟩ ⨾-22assoc121 ⟩
            π ⨾ (π ˘ ⨾ ρ) ⨾ B.op0
          ≈⟨ ⨾-cong21 π˘⨾ρ ⟩
            π ⨾ R.mor ⨾ B.op0
          ⊑⟨ ⨾-monotone2 R.commutes ⟨⊑≈⟩ ⨾-assocL ⟩
            (π ⨾ A.op0) ⨾ ℛ.mor R.mor
          □) ⟩
          (π ⨾ A.op0) ⨾ ℛ.mor R.mor ⊓ ρ ⨾ B.op0
        ≈˘⟨ P-op-⨾-ℛρ ⟩
          P.op0 ⨾ ℛ.mor ρ
        □
      }

  tabulation : Tabulation CA-𝒜 R
  tabulation = record
    {obj = P
    ;π = pi
    ;ρ = rho
    ;isTabulation = record
      {π˘⨾ρ = π˘⨾ρ
      ;extensionality = extensionality
      ;π˘⨾π = π˘⨾π
      ;ρ˘⨾ρ = ρ˘⨾ρ
        -- ;DefaultFork CA-𝒜 pi rho – not yet possibly in Agda-2.4.2.5
      ;fork = D.fork
      ;fork-def = λ {D′} {R′} {S′} → D.fork-def {D′} {R′} {S′}
      }
    }
    where module D = DefaultFork CA-𝒜 pi rho

open CAMor-Tab
CA-hasTabulations : HasTabulations CA-𝒜
CA-hasTabulations = record
  {tabulate = tab
    -- ;Default-par CA-𝒜 tab – not yet possibly in Agda-2.4.2.5
  ;par = D.par
  ;par-def = λ {A1} {A2} {B1} {B2} {Q1} {Q2} {R} {S}
            → D.par-def {A1} {A2} {B1} {B2} {Q1} {Q2} {R} {S}
  }
```

```
    where
      tab : {A B : Coalgebra ℛ} (R : CARelMor A B) → Tabulation CA-𝒜 R
      tab R = tabulation R (tabulate (CARelMor.mor R))
      module D = Default-par CA-𝒜 tab
```

2.9 Categoric.Coalgebra.TopMor

```
open import RATH.Level
open import RATH.Data.Product using (proj₁)
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
open import Categoric.Category
open import Categoric.OCC
open import Categoric.TopMor
open import Categoric.Relator.OCC
open import Categoric.Functor
open import Categoric.Functor.Retract
open import Categoric.Coalgebra.OCC


module Categoric.Coalgebra.TopMor
  {i j k₁ k₂ : Level} {Obj : Set i} (𝒞 : OCC j k₁ k₂ Obj)
  (let open OCC 𝒞)
  (hasTopMors : HasTopMors orderedSemigroupoid)
  where
open HasTopMors hasTopMors


module _ (ℛ : Relator 𝒞 𝒞) where
  private
    module ℛ = Relator ℛ
  Creates-⊤ : Set (i ⊍ j ⊍ k₂)
  Creates-⊤ = (A B : Coalgebra ℛ) → ⊤ ⨾ op₀ B ⊑ op₀ A ⨾ ℛ.mor ⊤
    where open Coalgebra


module _ {ℛ : Relator 𝒞 𝒞}
         (ℛ-creates-⊤ : Creates-⊤ ℛ)
  where
  private
    module ℛ = Relator ℛ
    module 𝒪 = OCC (CA-OCC ℛ)
  open TopMor 𝒪.orderedSemigroupoid


module CAMor-Top {A B : Coalgebra ℛ} where
  module A = Coalgebra A
  module B = Coalgebra B


  top : CARelMor A B
  top = record
    {mor = ⊤
    ;commutes = ⊑-begin
        ⊤ ⨾ B.op₀
      ⊑⟨ ℛ-creates-⊤ A B ⟩
```

```
          A.op₀ ⨾ 𝓡.mor ⊤
        □
      }
   CA-topMor : TopMor {A} {B}
   CA-topMor = record {mor = top; proof = ⊑-⊤}
open CAMor-Top public using (top; CA-topMor)

CA-HasTopMors : HasTopMors 𝓞.orderedSemigroupoid
CA-HasTopMors = record {topMor = CA-topMor}
   where
      open Coalgebra
      open CARelMor
```

2.10 Categoric.Coalgebra.DirectProduct

```
open import RATH.Level
open import RATH.Data.Product using (_,_; proj₁)
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
open import Categoric.Category
open import Categoric.Allegory
open import Categoric.Allegory.DirectProduct
open import Categoric.TopMor
open import Categoric.Relator.OCC
open import Categoric.Relator.Allegory
open import Categoric.Relator.JoinOp
open import Categoric.IdOp
open import Categoric.Coalgebra.OCC
open import Categoric.Relator.DirectProduct using (module DirProd)
```

```
module Categoric.Coalgebra.DirectProduct
   {i j k₁ k₂ : Level} {Obj : Set i}
   (𝓐 : Allegory j k₁ k₂ Obj)
   (let open Allegory 𝓐)
   {hasTopMors : HasTopMors orderedSemigroupoid}
   (dirProd : HasDirectProducts 𝓐 hasTopMors)
   where
open MeetPres 𝓐 𝓐
open import Categoric.Coalgebra.Allegory 𝓐
open HasTopMors hasTopMors
open import Categoric.Allegory.TopMor 𝓐 hasTopMors
open import Categoric.Coalgebra.TopMor occ hasTopMors
open HasDirectProducts dirProd
open DirProd 𝓐 dirProd
```

```
module _ {𝓡 : Relator occ occ}
         (let private module 𝓡 = Relator 𝓡)
         (𝓡-mor-⊓-⊒ : PreservesMeets-⊒ 𝓡)
         (𝓡-creates-⊤ : Creates-⊤ 𝓡)
   where
   private
      CA-𝓐 = CA-Allegory 𝓡 𝓡-mor-⊓-⊒
      CA-hasTop = CA-HasTopMors 𝓡-creates-⊤
```

```
module CAMor-DirProd (A B : Coalgebra ℛ) where
  private
    module A = Coalgebra A
    module B = Coalgebra B
    P₀ : Obj
    P₀ = A.Carrier ⊠ B.Carrier
    P-op : Mor P₀ (ℛ.obj P₀)
    P-op = π ⨾ A.op₀ ⨾ ℛ.mor π ˘ ⊓ ρ ⨾ B.op₀ ⨾ ℛ.mor ρ ˘
    P-op-unival : IsUnivalentI P-op
    P-op-unival = ⊑-begin
        (π ⨾ A.op₀ ⨾ ℛ.mor π ˘ ⊓ ρ ⨾ B.op₀ ⨾ ℛ.mor ρ ˘) ˘ ⨾ (π ⨾ A.op₀ ⨾ ℛ.mor π ˘ ⊓ ρ ⨾ B.op₀ ⨾ ℛ.mor ρ ˘)
      ≈⟨ ⨾-cong₁ (˘-⊓-distrib ⟨≈≈⟩ ⊓-cong (⨾⨾˘-˘ ⟨≈≈⟩ ⨾-assocL) (⨾⨾˘-˘ ⟨≈≈⟩ ⨾-assocL) ⟨≈≈˘⟩ ▽-def) ⟩
        ((ℛ.mor π ⨾ A.op₀ ˘) ▽ (ℛ.mor ρ ⨾ B.op₀ ˘)) ⨾ (π ⨾ A.op₀ ⨾ ℛ.mor π ˘ ⊓ ρ ⨾ B.op₀ ⨾ ℛ.mor ρ ˘)
      ⊑⟨ ⨾-⊓-subdistribR ⟨⊑⊑⟩ ⊓-monotone (⨾-monotone₁&₂₁ ▽⨾π-⊑) (⨾-monotone₁&₂₁ ▽⨾ρ-⊑) ⟩
        ℛ.mor π ⨾ A.op₀ ˘ ⨾ A.op₀ ⨾ ℛ.mor π ˘ ⊓ ℛ.mor ρ ⨾ B.op₀ ˘ ⨾ B.op₀ ⨾ ℛ.mor ρ ˘
      ⊑⟨ ⊓-monotone (⨾-monotone₂ (⨾-assocL ⟨≈⊑⟩ proj₁ A.op-univalent)) (⨾-monotone₂ (⨾-assocL ⟨≈⊑⟩ proj₁ B.op-univalent)) ⟩
        ℛ.mor π ⨾ ℛ.mor π ˘ ⊓ ℛ.mor ρ ⨾ ℛ.mor ρ ˘
      ≈⟨ ⊓-cong (⨾-cong₂ ℛ.mor-˘ ⟨≈˘≈˘⟩ ℛ.mor-⨾) (⨾-cong₂ ℛ.mor-˘ ⟨≈˘≈˘⟩ ℛ.mor-⨾) ⟩
        ℛ.mor (π ⨾ π ˘) ⊓ ℛ.mor (ρ ⨾ ρ ˘)
      ⊑⟨ ℛ-mor-⊓-⊒ ⟨⊑≈⟩ ℛ.mor-cong ⊠-extensionality ⟨⊑≈⟩ ℛ.mor-Id ⟩
        Id
      □
    P-op-⨾-ℛπ : P-op ⨾ ℛ.mor π ≈ π ⨾ A.op₀ ⊓ (ρ ⨾ B.op₀) ⨾ ℛ.mor ⊤
    P-op-⨾-ℛπ = ≈-begin
        P-op ⨾ ℛ.mor π
      ≈⟨ ⨾-cong₁ (⊓-cong ⨾-assocL ⨾-assocL) ⟩
        ((π ⨾ A.op₀) ⨾ ℛ.mor π ˘ ⊓ (ρ ⨾ B.op₀) ⨾ ℛ.mor ρ ˘) ⨾ ℛ.mor π
      ≈⟨ modal₂′unival (ℛ.mor-IsUnivalent π.univalent) ⟨≈˘≈⟩ ⊓-cong₂ ⨾-assoc ⟩
        π ⨾ A.op₀ ⊓ (ρ ⨾ B.op₀) ⨾ ℛ.mor ρ ˘ ⨾ ℛ.mor π
      ≈⟨ ⊓-cong₂ (⨾-cong₂ (⨾-cong₁ ℛ.mor-˘ ⟨≈˘≈˘⟩ ℛ.mor-⨾)) ⟩
        π ⨾ A.op₀ ⊓ (ρ ⨾ B.op₀) ⨾ ℛ.mor (ρ ˘ ⨾ π)
      ≈⟨ ⊓-cong₂ (⨾-cong₂ (ℛ.mor-cong ρ˘⨾π)) ⟩
        π ⨾ A.op₀ ⊓ (ρ ⨾ B.op₀) ⨾ ℛ.mor ⊤
        -- ≈⟨ {!⊓-cong₂ (total⨾⊤ (⨾-IsTotal ρ.total B.op-total)) ⟨≈≈⟩ ⊓-⊤!} ⟩
          -- π ⨾ A.op₀
      □
    P-op-⨾-ℛρ : P-op ⨾ ℛ.mor ρ ≈ (π ⨾ A.op₀) ⨾ ℛ.mor ⊤ ⊓ ρ ⨾ B.op₀
    P-op-⨾-ℛρ = ≈-begin
        P-op ⨾ ℛ.mor ρ
      ≈⟨ ⨾-cong₁ (⊓-cong ⨾-assocL ⨾-assocL) ⟩
        ((π ⨾ A.op₀) ⨾ ℛ.mor π ˘ ⊓ (ρ ⨾ B.op₀) ⨾ ℛ.mor ρ ˘) ⨾ ℛ.mor ρ
      ≈⟨ modal₂unival (ℛ.mor-IsUnivalent ρ.univalent) ⟨≈˘≈⟩ ⊓-cong₁ ⨾-assoc ⟩
        (π ⨾ A.op₀) ⨾ ℛ.mor π ˘ ⨾ ℛ.mor ρ ⊓ ρ ⨾ B.op₀
      ≈⟨ ⊓-cong₁ (⨾-cong₂ (⨾-cong₁ ℛ.mor-˘ ⟨≈˘≈˘⟩ ℛ.mor-⨾)) ⟩
        (π ⨾ A.op₀) ⨾ ℛ.mor (π ˘ ⨾ ρ) ⊓ ρ ⨾ B.op₀
      ≈⟨ ⊓-cong₁ (⨾-cong₂ (ℛ.mor-cong π˘⨾ρ)) ⟩
        (π ⨾ A.op₀) ⨾ ℛ.mor ⊤ ⊓ ρ ⨾ B.op₀
      □
    lemma₁ : Id ⊑ ((ρ ⨾ B.op₀) ⨾ ℛ.mor ⊤ ⨾ A.op₀ ˘) ⨾ π ˘
    lemma₁ = swap-⨾-⊑-total π.total (leftId ⟨≈⊑⟩ swap-⨾-⊑-total A.op-total (⊑-begin
        π ⨾ A.op₀
      ⊑⟨ ⨾-monotone₁ (proj₁ ρ.total ⟨⊑≈⟩ ⨾-assoc) ⟩
        (ρ ⨾ ρ ˘ ⨾ π) ⨾ A.op₀
      ≈⟨ ⨾-cong₁₂ ρ˘⨾π ⟩
        (ρ ⨾   ⊤) ⨾ A.op₀
      ⊑⟨ ⨾-monotone₁₂&₂ (ℛ-creates-⊤ B A) ⟩
```

```
        (ρ ⨾ B.op0) ⨾ 𝓡.mor ⊤
      □)⟨⊑≈⟩ ⨾-assoc)
    lemma2 : Id ⊑ ((π ⨾ A.op0) ⨾ 𝓡.mor ⊤ ⨾ B.op0 ˘) ⨾ ρ ˘
    lemma2 = Id˘ ⟨≈˘⊑⟩ ˘-monotone lemma1 ⟨⊑≈⟩
      (⨾˘-˘           ⟨≈≈⟩ ⨾-cong2 (⨾⨾˘-˘ ⟨≈≈⟩ ⨾-cong2 (⨾-cong (𝓡.mor-˘ ⟨≈˘≈⟩ 𝓡.mor-cong ⊤˘) ⨾-˘))
                      ⟨≈≈⟩ (⨾-assocL ⟨≈≈⟩ (⨾-cong2 ⨾-assocL ⟨≈≈⟩ ⨾-assocL)))
    P-op-total : IsTotalI P-op
    P-op-total = ⊑-begin
        Id
      ⊑⟨ ⊓-universal (⊓-universal ⊑-refl lemma1 ⟨⊑⊑⟩ modal2˘′′)
                      (⊓-universal lemma2 ⊑-refl ⟨⊑⊑⟩ modal2˘) ⟩
        (Id ⨾ π ⊓ (ρ ⨾ B.op0) ⨾ 𝓡.mor ⊤ ⨾ A.op0 ˘) ⨾ π ˘ ⊓
        ((π ⨾ A.op0) ⨾ 𝓡.mor ⊤ ⨾ B.op0 ˘ ⊓ Id ⨾ ρ) ⨾ ρ ˘
      ⊑⟨ ⊓-monotone (⨾-monotone1 (⊓-cong leftId ⨾-assocL ⟨≈⊑⟩ modal2˘′′) ⟨⊑≈⟩ ⨾-assoc)
                      (⨾-monotone1 (⊓-cong ⨾-assocL leftId ⟨≈⊑⟩ modal2˘) ⟨⊑≈⟩ ⨾-assoc) ⟩
        (π ⨾ A.op0 ⊓ (ρ ⨾ B.op0) ⨾ 𝓡.mor ⊤) ⨾ A.op0 ˘ ⨾ π ˘ ⊓
        ((π ⨾ A.op0) ⨾ 𝓡.mor ⊤ ⊓ ρ ⨾ B.op0) ⨾ B.op0 ˘ ⨾ ρ ˘
      ≈˘⟨ ⊓-cong (⨾-assocL ⟨≈≈⟩ ⨾-cong1 P-op-⨾-𝓡π) (⨾-assocL ⟨≈≈⟩ ⨾-cong1 P-op-⨾-𝓡ρ) ⟩
        P-op ⨾ 𝓡.mor π ⨾ A.op0 ˘ ⨾ π ˘ ⊓
        P-op ⨾ 𝓡.mor ρ ⨾ B.op0 ˘ ⨾ ρ ˘
      ≈˘⟨ ⨾-⊓-distribR (IsUnivalent-from-I P-op-unival) ⟩
        P-op ⨾ (𝓡.mor π ⨾ A.op0 ˘ ⨾ π ˘ ⊓ 𝓡.mor ρ ⨾ B.op0 ˘ ⨾ ρ ˘)
      ≈˘⟨ ⨾-cong2 (˘-⊓-distrib ⟨≈≈⟩ ⊓-cong ⨾⨾˘-˘ ⨾⨾˘-˘) ⟩
        P-op ⨾ (π ⨾ A.op0 ⨾ 𝓡.mor π ˘ ⊓ ρ ⨾ B.op0 ⨾ 𝓡.mor ρ ˘) ˘
      □
  P : Coalgebra 𝓡
  P = record
    {Carrier = P0
    ;op = record {mor = P-op; prf = P-op-unival , P-op-total}
    }
  module P = Coalgebra P


  pi : CARelMor P A
  pi = record
    {mor = π
    ;commutes = ⊑-begin
        π ⨾ A.op0
      ⊑⟨ ⊓-idempotent ⟨≈˘⊑⟩ ⊓-monotone2 (⨾-monotone1 (⊑-⊤ ⟨⊑≈˘⟩ total⨾⊤ ρ.total) ⟨⊑≈⟩ ⨾-assoc) ⟩
        π ⨾ A.op0 ⊓ ρ ⨾ ⊤ ⨾ A.op0
      ⊑⟨ ⊓-monotone2 (⨾-monotone2 (𝓡-creates-⊤ B A) ⟨⊑≈⟩ ⨾-assocL) ⟩
        π ⨾ A.op0 ⊓ (ρ ⨾ B.op0) ⨾ 𝓡.mor ⊤
      ≈˘⟨ P-op-⨾-𝓡π ⟩
        P.op0 ⨾ 𝓡.mor π
      □
    }


  rho : CARelMor P B
  rho = record
    {mor = ρ
    ;commutes = ⊑-begin
        ρ ⨾ B.op0
      ⊑⟨ ⊓-idempotent ⟨≈˘⊑⟩ ⊓-monotone1 (⨾-monotone1 (⊑-⊤ ⟨⊑≈˘⟩ total⨾⊤ π.total) ⟨⊑≈⟩ ⨾-assoc) ⟩
        π ⨾ ⊤ ⨾ B.op0 ⊓ ρ ⨾ B.op0
      ⊑⟨ ⊓-monotone1 (⨾-monotone2 (𝓡-creates-⊤ A B) ⟨⊑≈⟩ ⨾-assocL) ⟩
        (π ⨾ A.op0) ⨾ 𝓡.mor ⊤ ⊓ ρ ⨾ B.op0
```

```
      ≈˘⟨ P-op-⨾-ℛρ ⟩
        P.op₀ ⨾ ℛ.mor ρ
      □
    }

  isDirProd : IsDirectProduct CA-𝒜 CA-hasTop A B P
  isDirProd = record
    {π = pi
    ; ρ = rho
    ; tabulates-⊤ = record
      {π˘⨾ρ = π˘⨾ρ
      ; extensionality = ⊠-extensionality
      ; π˘⨾π = π˘⨾π
      ; ρ˘⨾ρ = ρ˘⨾ρ
        -- ; DefaultFork CA-𝒜 CA-hasTop pi rho – [ WK: Not yet possible in Agda-2.4.2.5 ]
      ; fork = Fork.fork
      ; fork-def = λ {D′} {R′} {S′} → Fork.fork-def {D′} {R′} {S′}
      }
    }
    where module Fork = DefaultFork CA-𝒜 CA-hasTop pi rho

open CAMor-DirProd

CA-hasDirectProducts : HasDirectProducts CA-𝒜 CA-hasTop
CA-hasDirectProducts = record
  { _⊠_ = P; isDirectProduct = isDirProd
    -- ; Default-⊗ CA-𝒜 CA-hasTop P isDirProd – [ WK: Not yet possible in Agda-2.4.2.5 ]
  ; _⊗_ = Par._⊗_
  ; ⊗-def = λ {A₁} {A₂} {B₁} {B₂} {R} {S} → Par.⊗-def {A₁} {A₂} {B₁} {B₂} {R} {S}
  }
  where module Par = Default-⊗ CA-𝒜 CA-hasTop P isDirProd
```

Chapter 3

Relators

3.1 Categoric.Relator.OCC

```
module Categoric.Relator.OCC where
open import RATH.Level
open import Categoric.Category
open import Categoric.OCC
```

In comparison with lax natural transformations, a different option for integrating functors with the ordering is to demand that the functor respects the ordering; such a monotone functor is frequently called a *relator*, essentially going back to Kawahara (1973):

```
record Relator {i₁ j₁ k₁₁ k₂₁ : Level} {Obj₁ : Set i₁} (Src : OCC j₁ k₁₁ k₂₁ Obj₁)
               {i₂ j₂ k₁₂ k₂₂ : Level} {Obj₂ : Set i₂} (Trg : OCC j₂ k₁₂ k₂₂ Obj₂)
               : Set (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂ ⊍ k₁₁ ⊍ k₁₂ ⊍ k₂₁ ⊍ k₂₂) where
  private
    module Src = OCC Src
    module Trg = OCC Trg
  open Category₁ Src.category
  open Category₂ Trg.category
  open OCC Src using () renaming (_⊑_ to _⊑₁_ ; _˘ to _˘₁)
  open OCC Trg using () renaming (_⊑_ to _⊑₂_ ; _˘ to _˘₂)
  field
    obj : Obj₁ → Obj₂
    mor : {A B : Obj₁} → Mor₁ A B → Mor₂ (obj A) (obj B)
    monotone : {A B : Obj₁} → {F G : Mor₁ A B} → F ⊑₁ G → mor F ⊑₂ mor G
    mor-⨾ : {A B C : Obj₁} → {F : Mor₁ A B} → {G : Mor₁ B C}
          → mor (F ⨾₁ G) ≈₂ mor F ⨾₂ mor G
    mor-Id : {A : Obj₁} → mor (Id₁ {A}) ≈₂ Id₂ {obj A}
    mor-˘ : {A B : Obj₁} {F : Mor₁ A B} → mor (F ˘₁) ≈₂ (mor F) ˘₂

  mor-cong : {A B : Obj₁} → {F G : Mor₁ A B} → F ≈₁ G → mor F ≈₂ mor G
  mor-cong F≈G = Trg.⊑-antisym (monotone (Src.⊑-reflexive F≈G))
                               (monotone (Src.⊑-reflexive′ F≈G))

  mor-IsUnivalentI : {A B : Obj₁} {F : Mor₁ A B} → Src.IsUnivalentI F → Trg.IsUnivalentI (mor F)
  mor-IsUnivalentI {A} {B} {F} F-unival = Trg.⊑-begin
      mor F ˘₂ ⨾₂ mor F
    Trg.≈⟨ Trg.⨾-cong₁ mor-˘ ⟨≈₂˘≈˘⟩ mor-⨾ ⟩
      mor (F ˘₁ ⨾₁ F)
    Trg.⊑⟨ monotone F-unival ⟩
```

```
      mor Id₁
    Trg.≈⟨ mor-Id ⟩
      Id₂
    Trg.□
  mor-IsUnivalent : {A B : Obj₁} {F : Mor₁ A B} → Src.IsUnivalent F → Trg.IsUnivalent (mor F)
  mor-IsUnivalent F-unival = Trg.IsUnivalent-from-I (mor-IsUnivalentI (Src.IsUnivalent-to-I F-unival))
  mor-IsTotalI : {A B : Obj₁} {F : Mor₁ A B} → Src.IsTotalI F → Trg.IsTotalI (mor F)
  mor-IsTotalI {A} {B} {F} F-total = Trg.⊑-begin
      Id₂
    Trg.≈˘⟨ mor-Id ⟩
      mor Id₁
    Trg.⊑⟨ monotone F-total ⟩
      mor (F ⨾₁ F ˘₁)
    Trg.≈⟨ mor-⨾ ⟨≈₂≈⟩ Trg.⨾-cong₂ mor-˘ ⟩
      mor F ⨾₂ mor F ˘₂
    Trg.□
  mor-IsTotal : {A B : Obj₁} {F : Mor₁ A B} → Src.IsTotal F → Trg.IsTotal (mor F)
  mor-IsTotal F-total = Trg.IsTotal-from-I (mor-IsTotalI (Src.IsTotal-to-I F-total))
```

Relators are closed under composition:

```
_⨾⨾_ : {i₁ j₁ k₁₁ k₂₁ : Level} {Obj₁ : Set i₁} {C₁ : OCC j₁ k₁₁ k₂₁ Obj₁}
       {i₂ j₂ k₁₂ k₂₂ : Level} {Obj₂ : Set i₂} {C₂ : OCC j₂ k₁₂ k₂₂ Obj₂}
       {i₃ j₃ k₁₃ k₂₃ : Level} {Obj₃ : Set i₃} {C₃ : OCC j₃ k₁₃ k₂₃ Obj₃}
       (F : Relator C₁ C₂) → (G : Relator C₂ C₃) → Relator C₁ C₃
_⨾⨾_ {C₃ = C₃} F G = record
  {obj = λ x → G.obj (F.obj x)
  ; mor = λ x → G.mor (F.mor x)
  ; monotone = λ x → G.monotone (F.monotone x)
  ; mor-⨾ = G.mor-cong F.mor-⨾ ⟨≈₃≈⟩ G.mor-⨾
  ; mor-Id = G.mor-cong F.mor-Id ⟨≈₃≈⟩ G.mor-Id
  ; mor-˘ = G.mor-cong F.mor-˘ ⟨≈₃≈⟩ G.mor-˘
  }
  where
    module F = Relator F
    module G = Relator G
    module C₃ = OCC C₃
    open Category₃ C₃.category
```

Each OCC has an identity relator:

```
Identity : {i j k₁ k₂ : Level} {Obj : Set i} (C : OCC j k₁ k₂ Obj) → Relator C C
Identity C = record
  {obj = λ x → x
  ; mor = λ x → x
  ; monotone = λ x → x
  ; mor-⨾ = OCC.≈-refl C
  ; mor-Id = OCC.≈-refl C
  ; mor-˘ = OCC.≈-refl C
  }
```

Constant relators:

```
Const : {i j k₁ k₂ : Level} {Obj : Set i} (C : OCC j k₁ k₂ Obj) → Obj → Relator C C
Const C A = record
  {obj = λ _ → A
```

```
  ; mor  =  λ _ → Id
  ; monotone  =  λ _ → ⊑-refl
  ; mor-⨾  =  ≈-sym leftId
  ; mor-Id  =  ≈-refl
  ; mor-˘  =  ≈-sym Id˘
  }
  where open OCC C
```

```
record NatSim {i₁ j₁ k₁₁ k₂₁ : Level} {Obj₁ : Set i₁} {Src : OCC j₁ k₁₁ k₂₁ Obj₁}
              {i₂ j₂ k₁₂ k₂₂ : Level} {Obj₂ : Set i₂} {Trg : OCC j₂ k₁₂ k₂₂ Obj₂}
              (F G : Relator Src Trg)
              : Set (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂ ⊍ k₁₂ ⊍ k₂₂) where
  private
    module Src  =  OCC Src
    module Trg  =  OCC Trg
    module F  =  Relator F
    module G  =  Relator G
  open Category₁ Src.category
  open Category₂ Trg.category
  field
    indmor : {A : Obj₁} → Mor₂ (F.obj A) (G.obj A)
    indmor-isMappingI : {A : Obj₁} → Trg.IsMappingI (indmor {A})
    naturality  :  {A B : Obj₁} {f : Mor₁ A B}
                → F.mor f ⨾₂ indmor {B} ≈₂ indmor {A} ⨾₂ G.mor f
  indmor-isMapping : {A : Obj₁} → Trg.IsMapping (indmor {A})
  indmor-isMapping  =  Trg.IsMapping-from-I indmor-isMappingI
```

3.2 Categoric.Relator.OCC.Retract

```
open import RATH.Level
open import Categoric.OCC
open import Categoric.Relator.OCC
```

```
module Categoric.Relator.OCC.Retract {i j k₁ k₂ : Level} {Obj : Set i} (𝒞 : OCC j k₁ k₂ Obj) where
open OCC 𝒞
```

```
retractRelator :  {i₂ : Level} {Obj₂ : Set i₂} (F : Obj₂ → Obj)
               → Relator (retractOCC F 𝒞) 𝒞
retractRelator F  =  record
  {obj         =  F
  ; mor        =  λ f → f
  ; monotone   =  λ f⊑g → f⊑g
  ; mor-⨾      =  ≈-refl
  ; mor-Id     =  ≈-refl
  ; mor-˘      =  ≈-refl
  }
```

```
module Retract²Relator
  {i₂ j₂   : Level} {Obj₂ : Set i₂} {Mor₂ : Obj₂ → Obj₂ → Set j₂}
```

```
  (Id₂ : {A : Obj₂} → Mor₂ A A)
  (_˘₂ : {A B : Obj₂} → Mor₂ A B → Mor₂ B A)
  (_⨾₂_ : {A B C : Obj₂} → Mor₂ A B → Mor₂ B C → Mor₂ A C)
  (FO : Obj₂ → Obj)
  (FM : {A B : Obj₂} → Mor₂ A B → Mor (FO A) (FO B))
  (FM-Id₂ : {A : Obj₂} → FM Id₂ ≈ Id {FO A})
  (FM-˘₂ : {A B : Obj₂} {f : Mor₂ A B} → FM (f ˘₂) ≈ FM f ˘)
  (FM-⨾₂ : {A B C : Obj₂} {f : Mor₂ A B} {g : Mor₂ B C}
    → FM (f ⨾₂ g) ≈ FM f ⨾ FM g)
  where
  𝒞₂ = retract²OCC 𝒞 Id₂ _˘₂ _⨾₂_ FO FM FM-Id₂ FM-˘₂ FM-⨾₂
  private
    module 𝒞₂ = OCC 𝒞₂
  retract²Relator : Relator 𝒞₂ 𝒞
  retract²Relator = record
    {obj = FO
    ; mor = FM
    ; monotone = λ f⊑g → f⊑g
    ; mor-⨾      = FM-⨾₂
    ; mor-Id     = FM-Id₂
    ; mor-˘      = FM-˘₂
    }
open Retract²Relator public hiding (𝒞₂)
```

3.3 Categoric.Relator.Allegory

```
module Categoric.Relator.Allegory where
open import RATH.Level
open import Categoric.Category
open import Categoric.OrderedSemigroupoid using (module LocOrd)
open import Categoric.Allegory
open import Categoric.Functor hiding (Identity; _⨾⨾_)
open import Categoric.Relator.OCC
open import Function using (_⟨_⟩_)
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
```

```
module _ {i₁ j₁ k₁₁ k₂₁ : Level} {Obj₁ : Set i₁} (Src : Allegory j₁ k₁₁ k₂₁ Obj₁)
         {i₂ j₂ k₁₂ k₂₂ : Level} {Obj₂ : Set i₂} (Trg : Allegory j₂ k₁₂ k₂₂ Obj₂) where
  private
    module Src = Allegory Src
    module Trg = Allegory Trg
  open Category₁ Src.category
  open Category₂ Trg.category
  open Allegory Src using () renaming (_⊑_ to _⊑₁_; _⊓_ to _⊓₁_; _˘ to _˘₁)
  open Allegory Trg using () renaming (_⊑_ to _⊑₂_; _⊓_ to _⊓₂_; _˘ to _˘₂)
```

```
  module MeetPres (ℛ : Relator Src.occ Trg.occ) where
    private module ℛ = Relator ℛ
    preservesMeets-⊑ : {A B : Obj₁} {R S : Mor₁ A B} → ℛ.mor (R ⊓₁ S) ⊑₂ ℛ.mor R ⊓₂ ℛ.mor S
    preservesMeets-⊑ = Trg.⊓-universal (ℛ.monotone Src.⊓-lower₁) (ℛ.monotone Src.⊓-lower₂)
```

```
  PreservesMeets-⊒ : Set (i₁ ⊍ j₁ ⊍ k₂₂)
  PreservesMeets-⊒ = {A B : Obj₁} {R S : Mor₁ A B} → ℛ.mor R ⊓₂ ℛ.mor S ⊑₂ ℛ.mor (R ⊓₁ S)

  PreservesMeets : Set (i₁ ⊍ j₁ ⊍ k₁₂)
  PreservesMeets = {A B : Obj₁} {R S : Mor₁ A B} → ℛ.mor (R ⊓₁ S) ≈₂ ℛ.mor R ⊓₂ ℛ.mor S

  module _ (preservesMeets-⊒ : PreservesMeets-⊒) where
    PreservesMeets-from-⊒ : PreservesMeets
    PreservesMeets-from-⊒ = Trg.⊑-antisym preservesMeets-⊑ preservesMeets-⊒

    preservesDom : {A B : Obj₁} {R : Mor₁ A B} → ℛ.mor (Src.dom R) ≈₂ Trg.dom (ℛ.mor R)
    preservesDom {A} {B} {R} = ≈₂-begin
        ℛ.mor (Src.dom R)
      ≈₂⟨⟩
        ℛ.mor (Id₁ ⊓₁ R ⨾₁ R ˘₁)
      ≈₂⟨ PreservesMeets-from-⊒ ⟨≈₂≈⟩ Trg.⊓-cong ℛ.mor-Id (ℛ.mor-⨾ ⟨≈₂≈⟩ Trg.⨾-cong₂ ℛ.mor-˘) ⟩
        Id₂ ⊓₂ ℛ.mor R ⨾₂ ℛ.mor R ˘₂
      ≈₂⟨⟩
        Trg.dom (ℛ.mor R)
      □₂
```

3.4 Categoric.Relator.DirectProduct

```
module Categoric.Relator.DirectProduct where

open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)

open import Categoric.OCC
open import Categoric.Relator.OCC
open import Categoric.Product.OCC
open import Categoric.Allegory
open import Categoric.Allegory.DirectProduct
open import Categoric.TopMor


module DirProd
    {i j k₁ k₂ : Level} {Obj : Set i}
    (𝒜 : Allegory j k₁ k₂ Obj)
    (let open Allegory 𝒜)
    {hasTopMors : HasTopMors orderedSemigroupoid}
    (dirProd : HasDirectProducts 𝒜 hasTopMors)
  where
  open HasTopMors hasTopMors
  open HasDirectProducts dirProd


DirProd : Relator (ProductOCC occ occ) occ
DirProd = record
  {obj = λ {(A , B) → A ⊠ B}
  ; mor = λ {(R , S) → R ⊗ S}
  ; monotone = λ {(F₁⊑F₂ , G₁⊑G₂) → ⊗-monotone F₁⊑F₂ G₁⊑G₂}
  ; mor-⨾      = ⨾-⊗-⨾
  ; mor-Id     = Id-⊗-Id
  ; mor-˘      = ≈-sym ⊗-˘
  }


module _
  {i₁ j₁ k₁₁ k₂₁ : Level} {Obj₁ : Set i₁} {𝒞₁ : OCC j₁ k₁₁ k₂₁ Obj₁}
```

```
  {i₂ j₂ k₁₂ k₂₂ : Level} {Obj₂ : Set i₂} {𝒞₂ : OCC j₂ k₁₂ k₂₂ Obj₂}
    where
  private
    module 𝒞₁ = OCC 𝒞₁
    module 𝒞₂ = OCC 𝒞₂
  _⊠⊠_ : Relator 𝒞₁ occ → Relator 𝒞₂ occ → Relator (ProductOCC 𝒞₁ 𝒞₂) occ
  ℱ ⊠⊠ 𝒢 = ProductRelator ℱ 𝒢 ⨾ DirProd

module _ where
  private module ℛ = Relator DirProd
  DirProd-mor-⊓-⊒ : {A₁ A₂ B₁ B₂ : Obj} {R₁ S₁ : Mor A₁ B₁} {R₂ S₂ : Mor A₂ B₂}
                  → ℛ.mor (R₁ , R₂) ⊓ ℛ.mor (S₁ , S₂) ⊑ ℛ.mor (R₁ ⊓ S₁ , R₂ ⊓ S₂)
  DirProd-mor-⊓-⊒ {A₁} {A₂} {B₁} {B₂} {R₁} {R₂} {S₁} {S₂} = ⊑-reflexive ⊗-⊓-⊗

open DirProd public
```

3.5 Categoric.Relator.JoinOp

```
module Categoric.Relator.JoinOp where
open import RATH.Level
open import Categoric.Category
open import Categoric.USLSemigroupoid
open import Categoric.OCC
open import Categoric.Relator.OCC

module _ {i₁ j₁ k₁₁ k₂₁ : Level} {Obj₁ : Set i₁} (Src : OCC j₁ k₁₁ k₂₁ Obj₁)
         {i₂ j₂ k₁₂ k₂₂ : Level} {Obj₂ : Set i₂} (Trg : OCC j₂ k₁₂ k₂₂ Obj₂) where
  private
    module Src = OCC Src
    module Trg = OCC Trg
  open Category₁ Src.category
  open Category₂ Trg.category
  open OCC Src using () renaming (_⊑_ to _⊑₁_ ; _˘ to _˘₁)
  open OCC Trg using () renaming (_⊑_ to _⊑₂_ ; _˘ to _˘₂)

  module JoinPres (joinOp₁ : JoinOp Src.orderedSemigroupoid)
                  (joinOp₂ : JoinOp Trg.orderedSemigroupoid)
                  (ℛ : Relator Src Trg) where
    private
      module joinOp₁ = JoinOp joinOp₁
      module joinOp₂ = JoinOp joinOp₂
      module ℛ = Relator ℛ
    open joinOp₁ using () renaming (_⊔_ to _⊔₁_)
    open joinOp₂ using () renaming (_⊔_ to _⊔₂_)
    PreservesJoins-⊑ : Set (i₁ ⊍ j₁ ⊍ k₂₂)
    PreservesJoins-⊑ = {A B : Obj₁} {R S : Mor₁ A B} → ℛ.mor (R ⊔₁ S) ⊑₂ ℛ.mor R ⊔₂ ℛ.mor S
    preservesJoins-⊒ : {A B : Obj₁} {R S : Mor₁ A B} → ℛ.mor R ⊔₂ ℛ.mor S ⊑₂ ℛ.mor (R ⊔₁ S)
    preservesJoins-⊒ = joinOp₂.⊔-universal (ℛ.monotone joinOp₁.⊔-upper₁) (ℛ.monotone joinOp₁.⊔-upper₂)
    PreservesJoins : Set (i₁ ⊍ j₁ ⊍ k₁₂)
    PreservesJoins = {A B : Obj₁} {R S : Mor₁ A B} → ℛ.mor (R ⊔₁ S) ≈₂ ℛ.mor R ⊔₂ ℛ.mor S
    PreservesJoins-from-⊑ : PreservesJoins-⊑ → PreservesJoins
    PreservesJoins-from-⊑ preservesJoins-⊑ = Trg.⊑-antisym preservesJoins-⊑ preservesJoins-⊒
```

3.6 Categoric.Relator.Collagory

```
module Categoric.Relator.Collagory where
open import RATH.Level
open import Categoric.Collagory
import Categoric.Relator.JoinOp
```

Here we only re-export Categoric.Relator.JoinOp.JoinPres with more convenient parameterisation.

```
module JoinPres {i₁ j₁ k₁₁ k₂₁ : Level} {Obj₁ : Set i₁} (Src : Collagory j₁ k₁₁ k₂₁ Obj₁)
                {i₂ j₂ k₁₂ k₂₂ : Level} {Obj₂ : Set i₂} (Trg : Collagory j₂ k₁₂ k₂₂ Obj₂) where
  private
    module Src = Collagory Src
    module Trg = Collagory Trg
  open Categoric.Relator.JoinOp.JoinPres Src.occ Trg.occ Src.joinOp Trg.joinOp public
```

3.7 Categoric.Relator.Residuals

```
module Categoric.Relator.Residuals where
open import RATH.Level
open import Categoric.Category
open import Categoric.OrderedSemigroupoid
open import Categoric.OrderedSemigroupoid.Residuals
open import Categoric.OCC
open import Categoric.Relator.OCC
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
```

```
module _ {i₁ j₁ k₁₁ k₂₁ : Level} {Obj₁ : Set i₁} (Src : OCC j₁ k₁₁ k₂₁ Obj₁)
         {i₂ j₂ k₁₂ k₂₂ : Level} {Obj₂ : Set i₂} (Trg : OCC j₂ k₁₂ k₂₂ Obj₂) where
  private
    module Src = OCC Src
    module Trg = OCC Trg
  open Category₁ Src.category
  open Category₂ Trg.category
  open OCC Src using () renaming (_⊑_ to _⊑₁_; _˘ to _˘₁)
  open OCC Trg using () renaming (_⊑_ to _⊑₂_; _˘ to _˘₂)

  module RResPres (rightResOp₁ : RightResOp Src.orderedSemigroupoid)
                  (rightResOp₂ : RightResOp Trg.orderedSemigroupoid)
                  (𝓡 : Relator Src Trg) where
    private
      module rightResOp₁ = RightResOp rightResOp₁
      module rightResOp₂ = RightResOp rightResOp₂
      module 𝓡 = Relator 𝓡
    open rightResOp₁ using () renaming (_\_ to _\₁_)
    open rightResOp₂ using () renaming (_\_ to _\₂_)
    PreservesRRes-⊒ : Set (i₁ ⊍ j₁ ⊍ k₂₂)
    PreservesRRes-⊒ = {A B C : Obj₁} {Q : Mor₁ A B} {S : Mor₁ A C}
      → 𝓡.mor Q \₂ 𝓡.mor S ⊑₂ 𝓡.mor (Q \₁ S)
```

```
preservesRRes-⊑  : {A B C : Obj₁} {Q : Mor₁ A B} {S : Mor₁ A C}
                 → ℛ.mor (Q \₁ S) ⊑₂ ℛ.mor Q \₂ ℛ.mor S
preservesRRes-⊑ {A} {B} {C} {Q} {S} = rightResOp₂.\-universal (Trg.⊑-begin
    ℛ.mor Q ⨾₂ ℛ.mor (Q \₁ S)
  Trg.≈˘⟨ ℛ.mor-⨾ ⟩
    ℛ.mor (Q ⨾₁ (Q \₁ S))
  Trg.⊑⟨ ℛ.monotone rightResOp₁.\-cancel-outer ⟩
    ℛ.mor S
  Trg.□)
PreservesRRes : Set (i₁ ⊍ j₁ ⊍ k₁₂)
PreservesRRes   = {A B C : Obj₁} {Q : Mor₁ A B} {S : Mor₁ A C}
  → ℛ.mor (Q \₁ S) ≈₂ ℛ.mor Q \₂ ℛ.mor S
PreservesRRes-from-⊒ : PreservesRRes-⊒ → PreservesRRes
PreservesRRes-from-⊒ preservesRRes-⊒ = Trg.⊑-antisym preservesRRes-⊑ preservesRRes-⊒
```

3.8 Categoric.Relator.DirectSum

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import Categoric.OCC
open import Categoric.Relator.OCC
open import Categoric.Product.OCC
open import Categoric.Allegory
open import Categoric.DistrAllegory
open import Categoric.DirectSum
open import Categoric.TopMor


module Categoric.Relator.DirectSum
  {i j k₁ k₂ : Level} {Obj : Set i}
  (𝒜 : DistrAllegory j k₁ k₂ Obj)
  (let open DistrAllegory 𝒜)
  (dirSum : HasDirectSum-L uslcc botMor)
  where
open HasDirectSum uslcc zeroMor dirSum


DirSum : Relator (ProductOCC occ occ) occ
DirSum = record
  {obj = λ {(A , B) → A ⊞ B}
  ; mor = λ {(R , S) → R ⊕ S}
  ; monotone = λ {(F₁⊑F₂ , G₁⊑G₂) → ⊕-monotone F₁⊑F₂ G₁⊑G₂}
  ; mor-⨾ = ⨾-⊕-⨾
  ; mor-Id = Id-⊕-Id
  ; mor-˘ = ≈-sym ⊕-˘
  }


module _
  {i₁ j₁ k₁₁ k₂₁ : Level} {Obj₁ : Set i₁} {𝒞₁ : OCC j₁ k₁₁ k₂₁ Obj₁}
  {i₂ j₂ k₁₂ k₂₂ : Level} {Obj₂ : Set i₂} {𝒞₂ : OCC j₂ k₁₂ k₂₂ Obj₂}
    where
  private
    module 𝒞₁ = OCC 𝒞₁
```

```
    module 𝒞₂ = OCC 𝒞₂
  _⊞⊞_ : Relator 𝒞₁ occ → Relator 𝒞₂ occ → Relator (ProductOCC 𝒞₁ 𝒞₂) occ
  ℱ ⊞⊞ 𝒢 = ProductRelator ℱ 𝒢 ⨾⨾ DirSum


⊔-⊥ : {A B : Obj} {R : Mor A B} → R ⊔ ⊥ ≈ R
⊔-⊥ {A} {B} {R} = ⊑-antisym (⊔-universal ⊑-refl ⊥-⊑) ⊔-upper₁


ι˘⨾-⊓-κ˘⨾ : {A₁ A₂ B : Obj} {R₁ : Mor A₁ B} {R₂ : Mor A₂ B}
          → ι ˘ ⨾ R₁ ⊓ κ ˘ ⨾ R₂ ≈ ⊥
ι˘⨾-⊓-κ˘⨾ {A₁} {A₂} {B} {R₁} {R₂} = ⊑⊥-≈ (⊑-begin
    ι ˘ ⨾ R₁ ⊓ κ ˘ ⨾ R₂
  ⊑⟨ modal₁ ⟨⊑≈⟩ ⨾-cong₂ (⊓-cong₂ (⨾-cong₁ ˘˘)) ⟩
    ι ˘ ⨾ (R₁ ⊓ ι ⨾ κ ˘ ⨾ R₂)
  ⊑⟨ ⨾-monotone₂ (⊓-monotone₂ (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ commutes ⟨≈⊑⟩ leftZero⊑)) ⟩
    ι ˘ ⨾ (R₁ ⊓ ⊥)
  ⊑⟨ ⨾-monotone₂ ⊓-lower₂ ⟨⊑⊑⟩ rightZero⊑ ⟩
    ⊥
  □)


⊕-⊓-ι˘⨾ : {A₁ A₂ B : Obj} {R₁ S₁ : Mor A₁ B} {R₂ : Mor A₂ B}
  → (R₁ ⊕ R₂) ⊓ ι ˘ ⨾ S₁ ≈ ι ˘ ⨾ (R₁ ⊓ S₁)
⊕-⊓-ι˘⨾ {A₁} {A₂} {B} {R₁} {S₁} {R₂} = ≈-begin
    (R₁ ⊕ R₂) ⊓ ι ˘ ⨾ S₁
  ≈⟨ ⊓-cong₁ ⊕-def ⟩
    (ι ˘ ⨾ R₁ ⊔ κ ˘ ⨾ R₂) ⊓ ι ˘ ⨾ S₁
  ≈⟨ ⊓-⊔-distribL ⟩
    (ι ˘ ⨾ R₁ ⊓ ι ˘ ⨾ S₁) ⊔ (κ ˘ ⨾ R₂ ⊓ ι ˘ ⨾ S₁)
  ≈˘⟨ ⊔-cong (˘⨾-⊓-distribR leftInj) ⊓-commutative ⟩
    ι ˘ ⨾ (R₁ ⊓ S₁) ⊔ (ι ˘ ⨾ S₁ ⊓ κ ˘ ⨾ R₂)
  ≈⟨ ⊔-cong₂ ι˘⨾-⊓-κ˘⨾ ⟩
    ι ˘ ⨾ (R₁ ⊓ S₁) ⊔ ⊥
  ≈⟨ ⊔-⊥ ⟩
    ι ˘ ⨾ (R₁ ⊓ S₁)
  □


⊕-⊓-κ˘⨾ : {A₁ A₂ B : Obj} {R₁ : Mor A₁ B} {R₂ S₂ : Mor A₂ B}
  → (R₁ ⊕ R₂) ⊓ κ ˘ ⨾ S₂ ≈ κ ˘ ⨾ (R₂ ⊓ S₂)
⊕-⊓-κ˘⨾ {A₁} {A₂} {B} {R₁} {R₂} {S₂} = ≈-begin
    (R₁ ⊕ R₂) ⊓ κ ˘ ⨾ S₂
  ≈⟨ ⊓-cong₁ ⊕-def ⟩
    (ι ˘ ⨾ R₁ ⊔ κ ˘ ⨾ R₂) ⊓ κ ˘ ⨾ S₂
  ≈⟨ ⊓-⊔-distribL ⟩
    (ι ˘ ⨾ R₁ ⊓ κ ˘ ⨾ S₂) ⊔ (κ ˘ ⨾ R₂ ⊓ κ ˘ ⨾ S₂)
  ≈⟨ ⊔-cong ι˘⨾-⊓-κ˘⨾ (≈-sym (˘⨾-⊓-distribR rightInj)) ⟩
    ⊥ ⊔ κ ˘ ⨾ (R₂ ⊓ S₂)
  ≈⟨ ⊔-commutative ⟨≈≈⟩ ⊔-⊥ ⟩
    κ ˘ ⨾ (R₂ ⊓ S₂)
  □


⊕-⊓-⊕ : {A₁ A₂ B : Obj} {R₁ S₁ : Mor A₁ B} {R₂ S₂ : Mor A₂ B}
       → (R₁ ⊕ R₂) ⊓ (S₁ ⊕ S₂) ≈ (R₁ ⊓ S₁) ⊕ (R₂ ⊓ S₂)
⊕-⊓-⊕ {A₁} {A₂} {B} {R₁} {S₁} {R₂} {S₂} = ≈-begin
    (R₁ ⊕ R₂) ⊓ (S₁ ⊕ S₂)
```

```
  ≈⟨ ⊓-cong₂ ⊕-def ⟩
    (R₁ ⊕ R₂) ⊓ (ι ˘ ⨾ S₁ ⊔ κ ˘ ⨾ S₂)
  ≈⟨ ⊓-⊔-distribR ⟩
    ((R₁ ⊕ R₂) ⊓ ι ˘ ⨾ S₁) ⊔ ((R₁ ⊕ R₂) ⊓ κ ˘ ⨾ S₂)
  ≈⟨ ⊔-cong ⊕-⊓-ι˘⨾ ⊕-⊓-κ˘⨾ ⟩
    ι ˘ ⨾ (R₁ ⊓ S₁) ⊔ κ ˘ ⨾ (R₂ ⊓ S₂)
  ≈˘⟨ ⊕-def ⟩
    (R₁ ⊓ S₁) ⊕ (R₂ ⊓ S₂)
  □
```

```
⊕-⊓-⊕ : {A₁ A₂ B₁ B₂ : Obj} {R₁ S₁ : Mor A₁ B₁} {R₂ S₂ : Mor A₂ B₂}
       → (R₁ ⊕ R₂) ⊓ (S₁ ⊕ S₂) ≈ (R₁ ⊓ S₁) ⊕ (R₂ ⊓ S₂)
⊕-⊓-⊕ {A₁} {A₂} {B₁} {B₂} {R₁} {S₁} {R₂} {S₂} = ≈-begin
    (R₁ ⊕ R₂) ⊓ (S₁ ⊕ S₂)
  ≈⟨⟩
    ((R₁ ⨾ ι) ⊕ (R₂ ⨾ κ)) ⊓ ((S₁ ⨾ ι) ⊕ (S₂ ⨾ κ))
  ≈⟨ ⊕-⊓-⊕ ⟩
    (R₁ ⨾ ι ⊓ S₁ ⨾ ι) ⊕ (R₂ ⨾ κ ⊓ S₂ ⨾ κ)
  ≈˘⟨ ⊕-cong (⨾-⊓-distribL leftInj) (⨾-⊓-distribL rightInj) ⟩
    ((R₁ ⊓ S₁) ⨾ ι) ⊕ ((R₂ ⊓ S₂) ⨾ κ)
  ≈⟨⟩
    (R₁ ⊓ S₁) ⊕ (R₂ ⊓ S₂)
  □
```

3.9 Categoric.Relator.Type

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import Categoric.OCC
open import Categoric.Allegory
open import Categoric.Relator.OCC
open import Categoric.Relator.Allegory
open import Categoric.Product.OCC
open import Categoric.Product.Allegory
```

```
module Categoric.Relator.Type
  {i j k₁ k₂ : Level} {Obj : Set i}
  (𝒜 : Allegory j k₁ k₂ Obj)
  (let open Allegory 𝒜)
  where
```

```
record InitialAlgebra (ℛ : Relator occ occ) : Set (i ⊍ j ⊍ k₁) where
  private module ℛ = Relator ℛ
  field
    T : Obj
    α : Mor (ℛ.obj T) T
    ⦇_⦈ : {A : Obj} → Mor (ℛ.obj A) A → Mor T A
    cata-universal : {A : Obj} {f : Mor (ℛ.obj A) A} {h : Mor T A}
                   → h ≈ ⦇ f ⦈ → α ⨾ h ≈ ℛ.mor h ⨾ f
    cata-unique    : {A : Obj} {f : Mor (ℛ.obj A) A} {h : Mor T A}
                   → α ⨾ h ≈ ℛ.mor h ⨾ f → h ≈ ⦇ f ⦈
```

```
cata    : {A : Obj} {f : Mor (𝓡.obj A) A} → α ⨾ ⦇ f ⦈ ≈ 𝓡.mor ⦇ f ⦈ ⨾ f
cata  =  cata-universal ≈-refl
⦇||⦈-cong : {A : Obj} {f₁ f₂ : Mor (𝓡.obj A) A} → f₁ ≈ f₂ → ⦇ f₁ ⦈ ≈ ⦇ f₂ ⦈
⦇||⦈-cong {A} {f₁} {f₂} f₁≈f₂  =  cata-unique (≈-begin
    α ⨾ ⦇ f₁ ⦈
  ≈⟨ cata ⟩
    𝓡.mor ⦇ f₁ ⦈ ⨾ f₁
  ≈⟨ ⨾-cong₂ f₁≈f₂ ⟩
    𝓡.mor ⦇ f₁ ⦈ ⨾ f₂
  □)
reflection : ⦇ α ⦈ ≈ Id
reflection  =  ≈-sym (cata-unique (≈-begin
    α ⨾ Id
  ≈⟨ rightId ⟩
    α
  ≈⟨ leftId ⟨≈˘≈˘⟩ ⨾-cong₁ 𝓡.mor-Id ⟩
    𝓡.mor Id ⨾ α
  □))
fusion     : {A B : Obj} {f : Mor (𝓡.obj A) A} {g : Mor (𝓡.obj B) B} {h : Mor A B}
           → f ⨾ h ≈ 𝓡.mor h ⨾ g → ⦇ f ⦈ ⨾ h ≈ ⦇ g ⦈
fusion {A} {B} {f} {g} {h} f⨾h≈𝓡h⨾g  =  cata-unique (≈-begin
    α ⨾ ⦇ f ⦈ ⨾ h
  ≈⟨ ⨾-cong₁&₂₁ cata ⟩
    𝓡.mor ⦇ f ⦈ ⨾ f ⨾ h
  ≈⟨ ⨾-cong₂ f⨾h≈𝓡h⨾g ⟩
    𝓡.mor ⦇ f ⦈ ⨾ 𝓡.mor h ⨾ g
  ≈⟨ ⨾-assocL ⟨≈≈˘⟩ ⨾-cong₁ 𝓡.mor-⨾ ⟩
    𝓡.mor (⦇ f ⦈ ⨾ h) ⨾ g
  □)
α-isIso : IsIso α        -- "Lambek's lemma" according to Bird and de Moor (1997, p. 49)
α-isIso  =  record
  {_⁻¹  =  ⦇ 𝓡.mor α ⦈
  ; rightInverse  =  ≈-begin
      α ⨾ ⦇ 𝓡.mor α ⦈
    ≈⟨ cata ⟩
      𝓡.mor ⦇ 𝓡.mor α ⦈ ⨾ 𝓡.mor α
    ≈˘⟨ 𝓡.mor-⨾ ⟩
      𝓡.mor (⦇ 𝓡.mor α ⦈ ⨾ α)
    ≈⟨ 𝓡.mor-cong leftInv ⟨≈≈⟩ 𝓡.mor-Id ⟩
      Id
    □
  ; leftInverse  =  leftInv
  }
  where
    leftInv : ⦇ 𝓡.mor α ⦈ ⨾ α ≈ Id
    leftInv  =  ≈-begin
        ⦇ 𝓡.mor α ⦈ ⨾ α
      ≈⟨ fusion {f  =  𝓡.mor α} {g  =  α} {h  =  α} ≈-refl ⟩
        ⦇ α ⦈
      ≈⟨ reflection ⟩
        Id
      □
α-iso : Iso (𝓡.obj T) T
α-iso  =  record {isoMor  =  α; isIso  =  α-isIso}
α˘≈α⁻¹  =  isoMor-˘ α-iso
α-mappingI  =  Iso→MappingI α-iso
```

```
  module α where
    private module α-isIso = IsIso α-isIso
    open α-isIso public
    open MappingI α-mappingI public renaming (prf to isMappingI)
    rightInverse′ : α ⨾ α ˘ ≈ Id
    rightInverse′ = ⨾-cong₂ α˘≈α⁻¹ ⟨≈≈⟩ α-isIso.rightInverse
    injective : IsInjective α
    injective = IsInjective-from-I (⊑-reflexive rightInverse′)
  ⦇|⨾|⦈ :   {A B : Obj} {f : Mor (ℛ.obj A) B} {g : Mor B A}
    → ⦇| f ⨾ g |⦈ ≈ ⦇| ℛ.mor g ⨾ f |⦈ ⨾ g
  ⦇|⨾|⦈ {A} {B} {f} {g} = ≈-sym (cata-unique (≈-begin
      α ⨾ ⦇| ℛ.mor g ⨾ f |⦈ ⨾ g
    ≈⟨  ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ cata ⟩
      (ℛ.mor ⦇| ℛ.mor g ⨾ f |⦈ ⨾ (ℛ.mor g ⨾ f)) ⨾ g
    ≈˘⟨ ⨾-cong₁ ℛ.mor-⨾ ⟨≈≈⟩ ⨾-₂₂assoc₁₂₁ ⟨≈≈⟩ ⨾-assocL ⟩
      ℛ.mor (⦇| ℛ.mor g ⨾ f |⦈ ⨾ g) ⨾ f ⨾ g
    □))


record Type (ℛ : Relator (ProductOCC occ occ) occ)
            (ℛ-preserves-⊓ : MeetPres.PreservesMeets (ProductAllegory 𝒜 𝒜) 𝒜 ℛ) : Set (i ⊍ j ⊍ k₁) where
  field
    initialAlgebra : (A : Obj) → InitialAlgebra (ℛ at₁ A)
  private
    module ℛ = Relator ℛ
    module ℛ₁ {A : Obj} = Relator (ℛ at₁ A)
    module ℛ₂ {B : Obj} = Relator (ℛ at₂ B)
  module initialAlgebra (A : Obj) = InitialAlgebra (initialAlgebra A)
  open initialAlgebra public using (T)
  module initialAlgebra′ {A : Obj} = InitialAlgebra (initialAlgebra A)
  open initialAlgebra′ public hiding (T)
  T-mor : {A B : Obj} → Mor A B → Mor (T A) (T B)
  T-mor f = ⦇| ℛ.mor (f , Id) ⨾ α |⦈
  functor-fusion : {A B C : Obj} {g : Mor A B} {h : Mor (ℛ.obj (B , C)) C}
    → T-mor g ⨾ ⦇| h |⦈ ≈ ⦇| ℛ.mor (g , Id) ⨾ h |⦈
  functor-fusion {A} {B} {C} {g} {h} = ≈-begin
      T-mor g ⨾ ⦇| h |⦈
    ≈⟨⟩
      ⦇| ℛ.mor (g , Id) ⨾ α |⦈ ⨾ ⦇| h |⦈
    ≈⟨ fusion (≈-begin
          (ℛ.mor (g , Id) ⨾ α) ⨾ ⦇| h |⦈
        ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ cata ⟩
          ℛ.mor (g , Id) ⨾ ℛ.mor (Id , ⦇| h |⦈) ⨾ h
        ≈⟨ ⨾-cong₁&₂₁ (birelator-Id-commute ℛ) ⟩
          ℛ.mor (Id , ⦇| h |⦈) ⨾ ℛ.mor (g , Id) ⨾ h
        □) ⟩
      ⦇| ℛ.mor (g , Id) ⨾ h |⦈
    □
  T-mor-cong : {A B : Obj} {f₁ f₂ : Mor A B} → f₁ ≈ f₂ → T-mor f₁ ≈ T-mor f₂
  T-mor-cong {A} {B} {f₁} {f₂} f₁≈f₂ = ≈-begin
        T-mor f₁
      ≈⟨⟩
        ⦇| ℛ.mor (f₁ , Id) ⨾ α |⦈
      ≈⟨ ⦇||⦈-cong (⨾-cong₁ (ℛ₂.mor-cong f₁≈f₂)) ⟩
        ⦇| ℛ.mor (f₂ , Id) ⨾ α |⦈
      ≈⟨⟩
        T-mor f₂
```

```
      ∎
  α-⨾-T-mor : {A B : Obj} {f : Mor A B} → α ⨾ T-mor f ≈ 𝓡.mor (f , T-mor f) ⨾ α
  α-⨾-T-mor {A} {B} {f} = ≈-begin
      α ⨾ T-mor f
    ≈⟨ cata ⟩
      𝓡₁.mor (T-mor f) ⨾ 𝓡.mor (f , Id) ⨾ α
    ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (𝓡.mor-⨾ ⟨≈˘≈⟩ 𝓡.mor-cong (leftId , rightId)) ⟩
      𝓡.mor (f , T-mor f) ⨾ α
    ∎
  T-mor-⊓ : {A B : Obj} {f₁ f₂ : Mor A B} → T-mor (f₁ ⊓ f₂) ≈ T-mor f₁ ⊓ T-mor f₂
  T-mor-⊓ {A} {B} {f₁} {f₂} = ≈-sym (cata-unique (≈-begin
      α ⨾ (T-mor f₁ ⊓ T-mor f₂)
    ≈⟨ ⨾-⊓-distribR α.univalent ⟩
      α ⨾ T-mor f₁ ⊓ α ⨾ T-mor f₂
    ≈⟨ ⊓-cong α-⨾-T-mor α-⨾-T-mor ⟩
      𝓡.mor (f₁ , T-mor f₁) ⨾ α ⊓ 𝓡.mor (f₂ , T-mor f₂) ⨾ α
    ≈˘⟨ ⨾-⊓-distribL α.injective ⟩
      (𝓡.mor (f₁ , T-mor f₁) ⊓ 𝓡.mor (f₂ , T-mor f₂)) ⨾ α
    ≈˘⟨ ⨾-cong₁ 𝓡-preserves-⊓ ⟩
      𝓡.mor (f₁ ⊓ f₂ , T-mor f₁ ⊓ T-mor f₂) ⨾ α
    ≈˘⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (𝓡.mor-⨾ ⟨≈˘≈⟩ 𝓡.mor-cong (leftId , rightId)) ⟩
      𝓡₁.mor (T-mor f₁ ⊓ T-mor f₂) ⨾ 𝓡.mor (f₁ ⊓ f₂ , Id) ⨾ α
    ∎))
  T-mor-monotone : {A B : Obj} {f₁ f₂ : Mor A B} → f₁ ⊑ f₂ → T-mor f₁ ⊑ T-mor f₂
  T-mor-monotone {A} {B} {f₁} {f₂} f₁⊑f₂ = ⊑-from-⊓₁ (≈-begin
      T-mor f₁ ⊓ T-mor f₂
    ≈˘⟨ T-mor-⊓ ⟩
      T-mor (f₁ ⊓ f₂)
    ≈⟨ T-mor-cong (⊑-to-⊓₁ f₁⊑f₂) ⟩
      T-mor f₁
    ∎)
  relator : Relator occ occ
  relator = record
    {obj = T
    ; mor = T-mor
    ; monotone = T-mor-monotone
    ; mor-⨾ = λ {A} {B} {C} {f} {g} → ≈-begin
        T-mor (f ⨾ g)
      ≈⟨⟩
        ⦇ 𝓡.mor (f ⨾ g , Id) ⨾ α ⦈
      ≈⟨ ⦇⦈-cong (⨾-cong₁ 𝓡₂.mor-⨾ ⟨≈≈⟩ ⨾-assoc) ⟩
        ⦇ 𝓡.mor (f , Id) ⨾ 𝓡.mor (g , Id) ⨾ α ⦈
      ≈˘⟨ functor-fusion ⟩
        T-mor f ⨾ ⦇ 𝓡.mor (g , Id) ⨾ α ⦈
      ≈⟨⟩
        T-mor f ⨾ T-mor g
      ∎
    ; mor-Id = ≈-begin
        ⦇ 𝓡.mor (Id , Id) ⨾ α ⦈
      ≈⟨ ⦇⦈-cong (⨾-cong₁ 𝓡.mor-Id ⟨≈≈⟩ leftId) ⟩
        ⦇ α ⦈
      ≈⟨ reflection ⟩
        Id
      ∎
    ; mor-˘ = λ {A} {B} {f} → ≈-sym (cata-unique (≈-begin
          α ⨾ T-mor f ˘
```

```
        ≈⟨ ⨾-cong₂ (˘-cong (≈-begin
             T-mor f
           ≈⟨ (leftId ⟨≈˘≈˘⟩ ⨾-cong₁ α.leftInverse) ⟨≈≈⟩ ⨾-assoc ⟩
             α._⁻¹ ⨾ α ⨾ T-mor f
           ≈⟨ ⨾-cong (≈-sym (isoMor-˘ α-iso)) α-⨾-T-mor ⟩
             α ˘ ⨾ ℛ.mor (f , T-mor f) ⨾ α
           □)) ⟩
           α ⨾ (α ˘ ⨾ ℛ.mor (f , T-mor f) ⨾ α) ˘
        ≈⟨ ⨾-cong₂ (˘⨾⨾-˘ ⟨≈≈⟩ ⨾-cong₁ (isoMor-˘ α-iso)) ⟩
           α ⨾ α._⁻¹ ⨾ ℛ.mor (f , T-mor f) ˘ ⨾ α
        ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong α.rightInverse (⨾-cong₁ (≈-sym ℛ.mor-˘)) ⟨≈≈⟩ leftId ⟩
          ℛ.mor (f ˘ , T-mor f ˘) ⨾ α
        ≈⟨ ⨾-cong₁ (ℛ.mor-cong (leftId , rightId) ⟨≈˘≈⟩ ℛ.mor-⨾) ⟨≈≈⟩ ⨾-assoc ⟩
          ℛ₁.mor (T-mor f ˘) ⨾ ℛ.mor (f ˘ , Id) ⨾ α
        □))
    }
```

3.10 Categoric.Relator.Cotype

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import Categoric.OCC
open import Categoric.Allegory
open import Categoric.Relator.OCC
open import Categoric.Relator.Allegory
open import Categoric.Product.OCC
open import Categoric.Product.Allegory
```

```
module Categoric.Relator.Cotype
  {i j k₁ k₂ : Level} {Obj : Set i}
  (𝒜 : Allegory j k₁ k₂ Obj)
  (let open Allegory 𝒜)
  where
```

```
record FinalCoalgebra (ℛ : Relator occ occ) : Set (i ⊍ j ⊍ k₁) where
  private module ℛ = Relator ℛ
  field
    T : Obj
    α : Mor T (ℛ.obj T)
    ⦇_⦈ : {A : Obj} → Mor A (ℛ.obj A) → Mor A T
    ana-universal : {A : Obj} {f : Mor A (ℛ.obj A)} {h : Mor A T}
                  → h ≈ ⦇ f ⦈ → h ⨾ α ≈ f ⨾ ℛ.mor h
    ana-unique    : {A : Obj} {f : Mor A (ℛ.obj A)} {h : Mor A T}
                  → h ⨾ α ≈ f ⨾ ℛ.mor h → h ≈ ⦇ f ⦈
  ana  : {A : Obj} {f : Mor A (ℛ.obj A)} → ⦇ f ⦈ ⨾ α ≈ f ⨾ ℛ.mor ⦇ f ⦈
  ana  =  ana-universal ≈-refl
  ⦇⦈-cong : {A : Obj} {f₁ f₂ : Mor A (ℛ.obj A)} → f₁ ≈ f₂ → ⦇ f₁ ⦈ ≈ ⦇ f₂ ⦈
  ⦇⦈-cong {A} {f₁} {f₂} f₁≈f₂  =  ana-unique (≈-begin
      ⦇ f₁ ⦈ ⨾ α
    ≈⟨ ana ⟩
      f₁ ⨾ ℛ.mor ⦇ f₁ ⦈
```

```
    ≈⟨ ⨾-cong₁ f₁≈f₂ ⟩
      f₂ ⨾ 𝓡.mor ⦇ f₁ ⦈
    □)
  reflection : ⦇ α ⦈ ≈ Id
  reflection = ≈-sym (ana-unique (≈-begin
      Id ⨾ α
    ≈⟨ leftId ⟩
      α
    ≈⟨ rightId ⟨≈˘≈˘⟩ ⨾-cong₂ 𝓡.mor-Id ⟩
      α ⨾ 𝓡.mor Id
    □))
  fusion    : {A B : Obj} {f : Mor A (𝓡.obj A)} {g : Mor B (𝓡.obj B)} {h : Mor B A}
            → h ⨾ f ≈ g ⨾ 𝓡.mor h → h ⨾ ⦇ f ⦈ ≈ ⦇ g ⦈
  fusion {A} {B} {f} {g} {h} h⨾f≈g⨾𝓡h = ana-unique (≈-begin
      (h ⨾ ⦇ f ⦈) ⨾ α
    ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ana ⟩
      h ⨾ f ⨾ 𝓡.mor ⦇ f ⦈
    ≈⟨ ⨾-cong₁&₂₁ h⨾f≈g⨾𝓡h ⟩
      g ⨾ 𝓡.mor h ⨾ 𝓡.mor ⦇ f ⦈
    ≈˘⟨ ⨾-cong₂ 𝓡.mor-⨾ ⟩
      g ⨾ 𝓡.mor (h ⨾ ⦇ f ⦈)
    □)
  α-isIso : IsIso α
  α-isIso = record
    { _⁻¹ = ⦇ 𝓡.mor α ⦈
    ; rightInverse = rightInv
    ; leftInverse = ≈-begin
        ⦇ 𝓡.mor α ⦈ ⨾ α
      ≈⟨ ana ⟩
        𝓡.mor α ⨾ 𝓡.mor ⦇ 𝓡.mor α ⦈
      ≈˘⟨ 𝓡.mor-⨾ ⟩
        𝓡.mor (α ⨾ ⦇ 𝓡.mor α ⦈)
      ≈⟨ 𝓡.mor-cong rightInv ⟨≈≈⟩ 𝓡.mor-Id ⟩
        Id
      □
    }
    where
      rightInv : α ⨾ ⦇ 𝓡.mor α ⦈ ≈ Id
      rightInv = ≈-begin
          α ⨾ ⦇ 𝓡.mor α ⦈
        ≈⟨ fusion {f = 𝓡.mor α} {g = α} {h = α} ≈-refl ⟩
          ⦇ α ⦈
        ≈⟨ reflection ⟩
          Id
        □
  α-iso : Iso T (𝓡.obj T)
  α-iso = record {isoMor = α; isIso = α-isIso}
  α˘≈α⁻¹ = isoMor-˘ α-iso
  α-mappingI = Iso→MappingI α-iso
  module α where
    private module α-isIso = IsIso α-isIso
    open α-isIso public
    open MappingI α-mappingI public renaming (prf to isMappingI)
    rightInverse′ : α ⨾ α ˘ ≈ Id
    rightInverse′ = ⨾-cong₂ α˘≈α⁻¹ ⟨≈≈⟩ α-isIso.rightInverse
    injective : IsInjective α
```

```
    injective = IsInjective-from-I (⊑-reflexive rightInverse′)
  (|⨾|) :     {A B : Obj} {f : Mor A B} {g : Mor B (𝓡.obj A)}
    → (| f ⨾ g |) ≈ f ⨾ (| g ⨾ 𝓡.mor f |)
  (|⨾|) {A} {B} {f} {g} = ≈-sym (ana-unique (≈-begin
      (f ⨾ (| g ⨾ 𝓡.mor f |)) ⨾ α
    ≈⟨   ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ana ⟩
      f ⨾ (g ⨾ 𝓡.mor f) ⨾ 𝓡.mor (| g ⨾ 𝓡.mor f |)
    ≈⟨   ⨾-₁₂₁assoc₂₂ ⟨≈≈˘⟩ ⨾-cong₂ 𝓡.mor-⨾ ⟩
      (f ⨾ g) ⨾ 𝓡.mor (f ⨾ (| g ⨾ 𝓡.mor f |))
    □))


record Cotype (𝓡 : Relator (ProductOCC occ occ) occ)
              (𝓡-preserves-⊓ : MeetPres.PreservesMeets (ProductAllegory 𝒜 𝒜) 𝒜 𝓡) : Set (i ⊍ j ⊍ k₁) where
  field
    finalCoalgebra : (A : Obj) → FinalCoalgebra (𝓡 at₁ A)
  private
    module 𝓡 = Relator 𝓡
    module 𝓡₁ {A : Obj} = Relator (𝓡 at₁ A)
    module 𝓡₂ {B : Obj} = Relator (𝓡 at₂ B)
  module finalCoalgebra (A : Obj) = FinalCoalgebra (finalCoalgebra A)
  open finalCoalgebra public using (T)
  module finalCoalgebra′ {A : Obj} = FinalCoalgebra (finalCoalgebra A)
  open finalCoalgebra′ public hiding (T)
  T-mor : {A B : Obj} → Mor A B → Mor (T A) (T B)
  T-mor f = (| α ⨾ 𝓡.mor (f , Id) |)
  functor-fusion : {A B C : Obj} {g : Mor A B} {h : Mor C (𝓡.obj (A , C))}
    → (| h |) ⨾ T-mor g ≈ (| h ⨾ 𝓡.mor (g , Id) |)
  functor-fusion {A} {B} {C} {g} {h} = ≈-begin
      (| h |) ⨾ T-mor g
    ≈⟨⟩
      (| h |) ⨾ (| α ⨾ 𝓡.mor (g , Id) |)
    ≈⟨ fusion (≈-begin
          (| h |) ⨾ α ⨾ 𝓡.mor (g , Id)
        ≈⟨ ⨾-cong₁&₂₁ ana ⟩
          h ⨾ 𝓡.mor (Id , (| h |)) ⨾ 𝓡.mor (g , Id)
        ≈⟨ ⨾-cong₂ (birelator-Id-commute 𝓡) ⟨≈˘≈⟩ ⨾-assocL ⟩
          (h ⨾ 𝓡.mor (g , Id)) ⨾ 𝓡.mor (Id , (| h |))
        □) ⟩
      (| h ⨾ 𝓡.mor (g , Id) |)
    □
  T-mor-cong : {A B : Obj} {f₁ f₂ : Mor A B} → f₁ ≈ f₂ → T-mor f₁ ≈ T-mor f₂
  T-mor-cong {A} {B} {f₁} {f₂} f₁≈f₂ = ≈-begin
        T-mor f₁
      ≈⟨⟩
        (| α ⨾ 𝓡.mor (f₁ , Id) |)
      ≈⟨ (|||)-cong (⨾-cong₂ (𝓡₂.mor-cong f₁≈f₂)) ⟩
        (| α ⨾ 𝓡.mor (f₂ , Id) |)
      ≈⟨⟩
        T-mor f₂
      □
  T-mor-⨾-α : {A B : Obj} {f : Mor A B} → T-mor f ⨾ α ≈ α ⨾ 𝓡.mor (f , T-mor f)
  T-mor-⨾-α {A} {B} {f} = ≈-begin
      T-mor f ⨾ α
    ≈⟨ ana ⟨≈≈⟩ ⨾-assoc ⟩
      α ⨾ 𝓡.mor (f , Id) ⨾ 𝓡₁.mor (T-mor f)
    ≈⟨ ⨾-cong₂ (𝓡.mor-⨾ ⟨≈˘≈⟩ 𝓡.mor-cong (rightId , leftId)) ⟩
```

```
      α ⨾ ℛ.mor (f , T-mor f)
    □
  T-mor-⊓ : {A B : Obj} {f₁ f₂ : Mor A B} → T-mor (f₁ ⊓ f₂) ≈ T-mor f₁ ⊓ T-mor f₂
  T-mor-⊓ {A} {B} {f₁} {f₂} = ≈-sym (ana-unique (≈-begin
      (T-mor f₁ ⊓ T-mor f₂) ⨾ α
    ≈⟨ ⨾-⊓-distribL α.injective ⟩
      T-mor f₁ ⨾ α ⊓ T-mor f₂ ⨾ α
    ≈⟨ ⊓-cong T-mor-⨾-α T-mor-⨾-α ⟩
      α ⨾ ℛ.mor (f₁ , T-mor f₁) ⊓ α ⨾ ℛ.mor (f₂ , T-mor f₂)
    ≈˘⟨ ⨾-⊓-distribR α.univalent ⟩
      α ⨾ (ℛ.mor (f₁ , T-mor f₁) ⊓ ℛ.mor (f₂ , T-mor f₂))
    ≈˘⟨ ⨾-cong₂ ℛ-preserves-⊓ ⟩
      α ⨾ ℛ.mor (f₁ ⊓ f₂ , T-mor f₁ ⊓ T-mor f₂)
    ≈˘⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ (ℛ.mor-⨾ ⟨≈˘≈⟩ ℛ.mor-cong (rightId , leftId)) ⟩
      (α ⨾ ℛ.mor (f₁ ⊓ f₂ , Id)) ⨾ ℛ₁.mor (T-mor f₁ ⊓ T-mor f₂)
    □))
  T-mor-monotone : {A B : Obj} {f₁ f₂ : Mor A B} → f₁ ⊑ f₂ → T-mor f₁ ⊑ T-mor f₂
  T-mor-monotone {A} {B} {f₁} {f₂} f₁⊑f₂ = ⊑-from-⊓₁ (≈-begin
      T-mor f₁ ⊓ T-mor f₂
    ≈˘⟨ T-mor-⊓ ⟩
      T-mor (f₁ ⊓ f₂)
    ≈⟨ T-mor-cong (⊑-to-⊓₁ f₁⊑f₂) ⟩
      T-mor f₁
    □)
  relator : Relator occ occ
  relator = record
    {obj = T
    ; mor = T-mor
    ; monotone = T-mor-monotone
    ; mor-⨾ = λ {A} {B} {C} {f} {g} → ≈-begin
        T-mor (f ⨾ g)
      ≈⟨⟩
        ⦇ α ⨾ ℛ.mor (f ⨾ g , Id) ⦈
      ≈⟨ ⦇⦈-cong (⨾-cong₂ ℛ₂.mor-⨾ ⟨≈≈⟩ ⨾-assocL) ⟩
        ⦇ (α ⨾ ℛ.mor (f , Id)) ⨾ ℛ.mor (g , Id) ⦈
      ≈˘⟨ functor-fusion ⟩
        ⦇ α ⨾ ℛ.mor (f , Id) ⦈ ⨾ T-mor g
      ≈⟨⟩
        T-mor f ⨾ T-mor g
      □
    ; mor-Id = ≈-begin
        ⦇ α ⨾ ℛ.mor (Id , Id) ⦈
      ≈⟨ ⦇⦈-cong (⨾-cong₂ ℛ.mor-Id ⟨≈≈⟩ rightId) ⟩
        ⦇ α ⦈
      ≈⟨ reflection ⟩
        Id
      □
    ; mor-˘ = λ {A} {B} {f} → ≈-sym (ana-unique (≈-begin
          T-mor f ˘ ⨾ α
        ≈⟨ ⨾-cong₁ (˘-cong (≈-begin
            T-mor f
          ≈⟨ (rightId ⟨≈˘≈˘⟩ ⨾-cong₂ α.rightInverse) ⟨≈≈⟩ ⨾-assocL ⟩
            (T-mor f ⨾ α) ⨾ α._⁻¹
          ≈⟨ ⨾-cong T-mor-⨾-α (≈-sym (isoMor-˘ α-iso)) ⟨≈≈⟩ ⨾-assoc ⟩
            α ⨾ ℛ.mor (f , T-mor f) ⨾ α ˘
          □)) ⟩
```

```
        (α ⨾ 𝓡.mor (f , T-mor f) ⨾ α ˘) ˘ ⨾ α
      ≈⟨ ⨾-cong₁ (⨾⨾˘-˘ ⟨≈≈⟩ ⨾-cong₂₂ (isoMor-˘ α-iso)) ⟩
        (α ⨾ 𝓡.mor (f , T-mor f) ˘ ⨾ α._⁻¹) ⨾ α
      ≈⟨ ⨾-assoc₃₊₁ ⟨≈≈⟩ ⨾-assocL ⟨≈≈⟩ ⨾-cong (⨾-cong₂ (≈-sym 𝓡.mor-˘)) α.leftInverse ⟨≈≈⟩ rightId ⟩
        α ⨾ 𝓡.mor (f ˘ , T-mor f ˘)
      ≈⟨ ⨾-cong₂ (𝓡.mor-cong (rightId , leftId) ⟨≈˘≈⟩ 𝓡.mor-⨾) ⟨≈≈⟩ ⨾-assocL ⟩
        (α ⨾ 𝓡.mor (f ˘ , Id)) ⨾ 𝓡₁.mor (T-mor f ˘)
      □))
  }
```

Chapter 4

Utilities

4.1 Categoric.TopMor

```
module Categoric.TopMor where
open import RATH.Level
open import Categoric.OrderedSemigroupoid
```

```
module TopMor {i j k₁ k₂ : Level} {Obj : Set i}
              (OSG : OrderedSemigroupoid j k₁ k₂ Obj) where
  open OrderedSemigroupoid OSG
  record TopMor {A B : Obj} : Set (j ⊍ k₂) where
    field
      mor : Mor A B
      proof : isTop mor
  topMor-≈ : {A B : Obj} {t₁ t₂ : Mor A B} → isTop t₁ → isTop t₂ → t₁ ≈ t₂
  topMor-≈ {_} {_} {t₁} {t₂} t₁-isTop t₂-isTop = ⊑-antisym t₂-isTop t₁-isTop
```

```
record HasTopMors {i j k₁ k₂ : Level} {Obj : Set i}
                  (OSG : OrderedSemigroupoid j k₁ k₂ Obj)
                  : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  open OrderedSemigroupoid OSG
  open TopMor              OSG
  field
    topMor : {A B : Obj} → TopMor {A} {B}
  ⊤ : {A B : Obj} → Mor A B
  ⊤ {A} {B} = TopMor.mor (topMor {A} {B})
  is-⊤ : {A B : Obj} → isTop (⊤ {A} {B})
  is-⊤ {A} {B} = TopMor.proof (topMor {A} {B})
  ⊑-⊤ : {A B : Obj} {R : Mor A B} → R ⊑ ⊤
  ⊑-⊤ {A} {B} {R} = is-⊤ {A} {B} {R}
  topMor-≈-⊤ : {A B : Obj} {t : Mor A B} → isTop t → t ≈ ⊤
  topMor-≈-⊤ t-isTop = topMor-≈ t-isTop is-⊤
  ⊤⊑-≈ : {A B : Obj} {R : Mor A B} → ⊤ ⊑ R → R ≈ ⊤
  ⊤⊑-≈ {A} {B} {R} ⊤⊑R = ⊑-antisym ⊑-⊤ ⊤⊑R
  ≈⊤-⊒ : {A B : Obj} {t R : Mor A B} → t ≈ ⊤ → R ⊑ t
  ≈⊤-⊒ t≈⊤ = ⊑-⊤ ⟨⊑≈˘⟩ t≈⊤
```

```
retractHasTopMors : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                  → (F : Obj₂ → Obj₁)
                  → {base : OrderedSemigroupoid j k₁ k₂ Obj₁}
                  → HasTopMors base → HasTopMors (retractOrderedSemigroupoid F base)
retractHasTopMors F hasTop = let open HasTopMors hasTop in
  record {topMor = λ {A} {B} → record {mor = ⊤; proof = ⊑-⊤}}
```

4.2 Categoric.Allegory.TopMor

```
open import RATH.Level
open import Categoric.Allegory
open import Categoric.TopMor
```

```
module Categoric.Allegory.TopMor
  {i j k₁ k₂ : Level} {Obj : Set i}
  (𝒜 : Allegory j k₁ k₂ Obj)
  (let open Allegory 𝒜)
  (hasTopMors : HasTopMors orderedSemigroupoid)
  where
open HasTopMors hasTopMors
```

```
⊤˘ : {A B : Obj} → ⊤ {A} {B} ˘ ≈ ⊤
⊤˘ = ⊤⊑-≈ (⊑-˘-swap ⊑-⊤)
```

```
⊓-⊤ : {A B : Obj} {R : Mor A B} → R ⊓ ⊤ ≈ R
⊓-⊤ = ⊑-to-⊓₁ ⊑-⊤
⊤-⊓ : {A B : Obj} {R : Mor A B} → ⊤ ⊓ R ≈ R
⊤-⊓ = ⊑-to-⊓₂ ⊑-⊤
```

```
dom⨾⊤ : {A B C : Obj} {R : Mor A B} → dom R ⨾ ⊤ {A} {C} ≈ R ⨾ ⊤ {B} {C}
dom⨾⊤ {A} {B} {C} {R} = ≈-begin
    dom R ⨾ ⊤
  ≈⟨⟩
    (Id ⊓ R ⨾ R ˘) ⨾ ⊤
  ≈⟨ ⊑-antisym (⊑-begin
      (Id ⊓ R ⨾ R ˘) ⨾ ⊤
    ⊑⟨ ⨾-monotone₁ ⊓-lower₂ ⟨⊑≈⟩ ⨾-assoc ⟩
      R ⨾ R ˘ ⨾ ⊤
    ⊑⟨ ⨾-monotone₂ ⊑-⊤ ⟩
      R ⨾ ⊤
    □) (⊑-begin
      R ⨾ ⊤
    ≈⟨ ⨾-cong₁ dom-D1 ⟨≈˘≈⟩ ⨾-assoc ⟩
      dom R ⨾ R ⨾ ⊤
    ⊑⟨ ⨾-monotone₂ ⊑-⊤ ⟩
      dom R ⨾ ⊤
      □) ⟩
      R ⨾ ⊤
  □
```

```
total⨾⊤ : {A B C : Obj} {R : Mor A B} → IsTotal R → R ⨾ ⊤ {B} {C} ≈ ⊤ {A} {C}
total⨾⊤ {A} {B} {C} {R} R-total = ≈-begin
    R ⨾ ⊤
  ≈˘⟨ dom⨾⊤ ⟩
    dom R ⨾ ⊤
  ≈⟨ ⨾-cong₁ (total-dom R-total) ⟨≈≈⟩ leftId ⟩
    ⊤
  □

dom⨾-≈-⊓-⨾⊤ : {A B C : Obj} {R : Mor A B} {S : Mor A C} → dom S ⨾ R ≈ R ⊓ S ⨾ ⊤
dom⨾-≈-⊓-⨾⊤ {A} {B} {C} {R} {S} = ≈-begin
  dom S ⨾ R
  ≈⟨ ⊑-antisym (⊑-begin
      dom S ⨾ R
    ≈⟨⟩
      (Id ⊓ S ⨾ S ˘) ⨾ R
    ⊑⟨ ⨾-⊓-subdistribL ⟨⊑≈⟩ ⊓-cong leftId ⨾-assoc ⟩
      R ⊓ S ⨾ S ˘ ⨾ R
    ⊑⟨ ⊓-monotone₂ (⨾-monotone₂ ⊑-⊤) ⟩
      R ⊓ S ⨾ ⊤
    □) (⊑-begin
      R ⊓ S ⨾ ⊤
    ≈˘⟨ ⊓-cong₂ dom⨾⊤ ⟩
      R ⊓ dom S ⨾ ⊤
    ⊑⟨ modal₁″ ⟩
      dom S ⨾ (dom S ˘ ⨾ R ⊓ ⊤)
    ⊑⟨ ⨾-monotone₂ ⊓-lower₁ ⟨⊑≈⟩ ⨾-assocL ⟩
      (dom S ⨾ dom S ˘) ⨾ R
    ≈⟨ ⨾-cong₁ (⨾-cong₂ dom-˘ ⟨≈≈⟩ dom-⨾-idempotent) ⟩
      dom S ⨾ R
    □) ⟩
    R ⊓ S ⨾ ⊤
  □
```

4.3 Categoric.Allegory.Tabulation

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import Categoric.Allegory

module Categoric.Allegory.Tabulation
  {i j k₁ k₂ : Level} {Obj : Set i}
  (𝒜 : Allegory j k₁ k₂ Obj)
  where
open Allegory 𝒜

module DefaultFork {A B P : Obj} (π : Mor P A) (ρ : Mor P B) where
  fork : {Z : Obj} → Mor Z A → Mor Z B → Mor Z P
  fork R S = R ⨾ π ˘ ⊓ S ⨾ ρ ˘
  fork-def : {Z : Obj} {R : Mor Z A} {S : Mor Z B} → fork R S ≈ R ⨾ π ˘ ⊓ S ⨾ ρ ˘
  fork-def = ≈-refl
```

```
record IsTabulation {A B : Obj} (Q : Mor A B) {P : Obj} (π : Mor P A) (ρ : Mor P B) : Set (i ⊍ j ⊍ k₁) where
  field
    π˘⨾ρ : π ˘ ⨾ ρ ≈ Q
    extensionality : π ⨾ π ˘ ⊓ ρ ⨾ ρ ˘ ≈ Id
    π˘⨾π : π ˘ ⨾ π ≈ dom Q
    ρ˘⨾ρ : ρ ˘ ⨾ ρ ≈ ran Q
    fork : {Z : Obj} → Mor Z A → Mor Z B → Mor Z P
    fork-def : {Z : Obj} {R : Mor Z A} {S : Mor Z B} → fork R S ≈ R ⨾ π ˘ ⊓ S ⨾ ρ ˘
  ρ˘⨾π : ρ ˘ ⨾ π ≈ Q ˘
  ρ˘⨾π = ≈-begin
      ρ ˘ ⨾ π
    ≈˘⟨ ˘⨾-˘ ⟩
      (π ˘ ⨾ ρ) ˘
    ≈⟨ ˘-cong π˘⨾ρ ⟩
      Q ˘
    □
  module π = MappingI (record
    {mor = π
    ; prf = (π˘⨾π ⟨≈⊑⟩ ⊓-lower₁) , (extensionality ⟨≈˘⊑⟩ ⊓-lower₁)
    })
  module ρ = MappingI (record
    {mor = ρ
    ; prf = (ρ˘⨾ρ ⟨≈≈⟩ ran-def ⟨≈⊑⟩ ⊓-lower₁) , (extensionality ⟨≈˘⊑⟩ ⊓-lower₂)
    })
  fork-cong : {Z : Obj} {R₁ R₂ : Mor Z A} {S₁ S₂ : Mor Z B} → R₁ ≈ R₂ → S₁ ≈ S₂ → fork R₁ S₁ ≈ fork R₂ S₂
  fork-cong R₁≈R₂ S₁≈S₂ = fork-def ⟨≈≈⟩ ⊓-cong (⨾-cong₁ R₁≈R₂) (⨾-cong₁ S₁≈S₂) ⟨≈≈˘⟩ fork-def
  fork-cong₁ : {Z : Obj} {R₁ R₂ : Mor Z A} {S : Mor Z B} → R₁ ≈ R₂ → fork R₁ S ≈ fork R₂ S
  fork-cong₁ R₁≈R₂ = fork-cong R₁≈R₂ ≈-refl
  fork-cong₂ : {Z : Obj} {R : Mor Z A} {S₁ S₂ : Mor Z B} → S₁ ≈ S₂ → fork R S₁ ≈ fork R S₂
  fork-cong₂ = fork-cong ≈-refl
  fork-monotone : {Z : Obj} {R₁ R₂ : Mor Z A} {S₁ S₂ : Mor Z B} → R₁ ⊑ R₂ → S₁ ⊑ S₂ → fork R₁ S₁ ⊑ fork R₂ S₂
  fork-monotone R₁⊑R₂ S₁⊑S₂ = fork-def ⟨≈⊑⟩ ⊓-monotone (⨾-monotone₁ R₁⊑R₂) (⨾-monotone₁ S₁⊑S₂) ⟨⊑≈˘⟩ fork-def
  fork-monotone₁ : {Z : Obj} {R₁ R₂ : Mor Z A} {S : Mor Z B} → R₁ ⊑ R₂ → fork R₁ S ⊑ fork R₂ S
  fork-monotone₁ R₁⊑R₂ = fork-monotone R₁⊑R₂ ⊑-refl
  fork-monotone₂ : {Z : Obj} {R : Mor Z A} {S₁ S₂ : Mor Z B} → S₁ ⊑ S₂ → fork R S₁ ⊑ fork R S₂
  fork-monotone₂ = fork-monotone ⊑-refl
    -- Initial part of (Bird and de Moor, 1997, (5.6))
  fork⨾π : {Z : Obj} {R : Mor Z A} {S : Mor Z B} → fork R S ⨾ π ≈ R ⊓ S ⨾ Q ˘
  fork⨾π {Z} {R} {S} = ≈-begin
      (fork R S) ⨾ π
    ≈⟨ ⨾-cong₁ fork-def ⟩
      (R ⨾ π ˘ ⊓ S ⨾ ρ ˘) ⨾ π
    ≈⟨ modal₂′unival π.univalent ⟨≈˘≈⟩ ⊓-cong₂ ⨾-assoc ⟩
      R ⊓ S ⨾ ρ ˘ ⨾ π
    ≈⟨ ⊓-cong₂ (⨾-cong₂ ρ˘⨾π) ⟩
      R ⊓ S ⨾ Q ˘
    □
  fork⨾π-⊑ : {Z : Obj} {R : Mor Z A} {S : Mor Z B} → fork R S ⨾ π ⊑ R
  fork⨾π-⊑ = fork⨾π ⟨≈⊑⟩ ⊓-lower₁
    -- Initial part of (Bird and de Moor, 1997, (5.7))
  fork⨾ρ : {Z : Obj} {R : Mor Z A} {S : Mor Z B} → fork R S ⨾ ρ ≈ R ⨾ Q ⊓ S
  fork⨾ρ {Z} {R} {S} = ≈-begin
      fork R S ⨾ ρ
    ≈⟨ ⨾-cong₁ fork-def ⟩
```

```
        (R ⨾ π ˘ ⊓ S ⨾ ρ ˘) ⨾ ρ
      ≈⟨ modal₂unival ρ.univalent ⟨≈˘≈⟩ ⊓-cong₁ ⨾-assoc ⟩
        R ⨾ π ˘ ⨾ ρ ⊓ S
      ≈⟨ ⊓-cong₁ (⨾-cong₂ π˘⨾ρ) ⟩
        R ⨾ Q ⊓ S
      □
    fork⨾ρ-⊑ : {Z : Obj} {R : Mor Z A} {S : Mor Z B} → fork R S ⨾ ρ ⊑ S
    fork⨾ρ-⊑ = fork⨾ρ ⟨≈⊑⟩ ⊓-lower₂
    ▽Id⨾π : {R : Mor B A} → fork R Id ⨾ π ≈ R ⊓ Q ˘
    ▽Id⨾π {R} = ≈-begin
        fork R Id ⨾ π
      ≈⟨ fork⨾π ⟩
        R ⊓ Id ⨾ Q ˘
      ≈⟨ ⊓-cong₂ leftId ⟩
        R ⊓ Q ˘
      □
    Id▽⨾ρ : {S : Mor A B} → fork Id S ⨾ ρ ≈ Q ⊓ S
    Id▽⨾ρ {S} = ≈-begin
        fork Id S ⨾ ρ
      ≈⟨ fork⨾ρ ⟩
        Id ⨾ Q ⊓ S
      ≈⟨ ⊓-cong₁ leftId ⟩
        Q ⊓ S
      □

record Tabulation {A B : Obj} (Q : Mor A B) : Set (i ⊍ j ⊍ k₁) where
  field
    obj : Obj
    π : Mor obj A
    ρ : Mor obj B
    isTabulation : IsTabulation Q π ρ
  open IsTabulation isTabulation public

module Default-par₀ {B₁ B₂ C₁ C₂ : Obj} {Q₁ : Mor B₁ B₂} {Q₂ : Mor C₁ C₂}
                    (Q₁-tab : Tabulation Q₁)
                    (Q₂-tab : Tabulation Q₂) where
  private
    module Q₁ = Tabulation Q₁-tab
    module Q₂ = Tabulation Q₂-tab
  par : Mor B₁ C₁ → Mor B₂ C₂ → Mor Q₁.obj Q₂.obj
  par R S = Q₂.fork (Q₁.π ⨾ R) (Q₁.ρ ⨾ S)
  par-def : {R : Mor B₁ C₁} {S : Mor B₂ C₂}
          → par R S ≈ Q₂.fork (Q₁.π ⨾ R) (Q₁.ρ ⨾ S)
  par-def = ≈-refl

module Default-par (tabulate : {A B : Obj} (Q : Mor A B) → Tabulation Q) where
  module _ {A B : Obj} (Q : Mor A B) where
    open Tabulation (tabulate Q) public
  par : {A₁ A₂ B₁ B₂ : Obj} (Q₁ : Mor A₁ A₂) (Q₂ : Mor B₁ B₂) → Mor A₁ B₁ → Mor A₂ B₂ → Mor (obj Q₁) (obj Q₂)
  par Q₁ Q₂ R S = fork Q₂ (π Q₁ ⨾ R) (ρ Q₁ ⨾ S)
  par-def : {A₁ A₂ B₁ B₂ : Obj} {Q₁ : Mor A₁ A₂} {Q₂ : Mor B₁ B₂} {R : Mor A₁ B₁} {S : Mor A₂ B₂}
          → par Q₁ Q₂ R S ≈ fork Q₂ (π Q₁ ⨾ R) (ρ Q₁ ⨾ S)
  par-def = ≈-refl

module TwoTabulations {B₁ B₂ C₁ C₂ : Obj} {Q₁ : Mor B₁ B₂} {Q₂ : Mor C₁ C₂}
                      (Q₁-tab : Tabulation Q₁)
```

```
                         (Q₂-tab : Tabulation Q₂) where
private
  module Q₁ = Tabulation Q₁-tab
  module Q₂ = Tabulation Q₂-tab
record TabulatedParComp : Set (j ⊍ k₁) where
  field
    par : Mor B₁ C₁ → Mor B₂ C₂ → Mor Q₁.obj Q₂.obj
    par-def : {R : Mor B₁ C₁} {S : Mor B₂ C₂} → par R S ≈ Q₂.fork (Q₁.π ⨾ R) (Q₁.ρ ⨾ S)
  par-def′   : {R : Mor B₁ C₁} {S : Mor B₂ C₂} → par R S ≈ Q₁.π ⨾ R ⨾ Q₂.π ˘ ⊓ Q₁.ρ ⨾ S ⨾ Q₂.ρ ˘
  par-def′ = par-def ⟨≈≈⟩ Q₂.fork-def ⟨≈≈⟩ ⊓-cong ⨾-assoc ⨾-assoc
  ⊗-monotone : {R₁ R₂ : Mor B₁ C₁} {S₁ S₂ : Mor B₂ C₂} → R₁ ⊑ R₂ → S₁ ⊑ S₂ → par R₁ S₁ ⊑ par R₂ S₂
  ⊗-monotone R₁⊑R₂ S₁⊑S₂ = par-def ⟨≈⊑⟩ Q₂.fork-monotone (⨾-monotone₂ R₁⊑R₂) (⨾-monotone₂ S₁⊑S₂) ⟨⊑≈˘⟩ par-def
  ⊗-cong  :  {R₁ R₂ : Mor B₁ C₁} {S₁ S₂ : Mor B₂ C₂} → R₁ ≈ R₂ → S₁ ≈ S₂ → par R₁ S₁ ≈ par R₂ S₂
  ⊗-cong R₁≈R₂ S₁≈S₂ = ⊑-antisym (⊗-monotone (⊑-reflexive R₁≈R₂) (⊑-reflexive S₁≈S₂))
    (⊗-monotone (⊑-reflexive′ R₁≈R₂) (⊑-reflexive′ S₁≈S₂))
  ⊗-cong₁ :  {R₁ R₂ : Mor B₁ C₁} {S : Mor B₂ C₂} → R₁ ≈ R₂ → par R₁ S ≈ par R₂ S
  ⊗-cong₁ R₁≈R₂ = ⊗-cong R₁≈R₂ ≈-refl
  ⊗-cong₂ :  {R : Mor B₁ C₁} {S₁ S₂ : Mor B₂ C₂} → S₁ ≈ S₂ → par R S₁ ≈ par R S₂
  ⊗-cong₂ = ⊗-cong ≈-refl
  ⊗⨾π-⊑ : {R₁ : Mor B₁ C₁} {R₂ : Mor B₂ C₂} → par R₁ R₂ ⨾ Q₂.π ⊑ Q₁.π ⨾ R₁
  ⊗⨾π-⊑ {R₁} {R₂} = ⊑-begin
      par R₁ R₂ ⨾ Q₂.π
    ≈⟨ ⨾-cong₁ par-def′ ⟩
      (Q₁.π ⨾ R₁ ⨾ Q₂.π ˘ ⊓ Q₁.ρ ⨾ R₂ ⨾ Q₂.ρ ˘) ⨾ Q₂.π
    ⊑⟨ ⨾-monotone₁ ⊓-lower₁ ⟨⊑≈⟩ ⨾-assoc₃₊₁ ⟩
      Q₁.π ⨾ R₁ ⨾ Q₂.π ˘ ⨾ Q₂.π
    ⊑⟨ ⨾-monotone₂ (proj₂ Q₂.π.univalent) ⟩
      Q₁.π ⨾ R₁
    □
  ⊗⨾ρ-⊑ : {R₁ : Mor B₁ C₁} {R₂ : Mor B₂ C₂} → par R₁ R₂ ⨾ Q₂.ρ ⊑ Q₁.ρ ⨾ R₂
  ⊗⨾ρ-⊑ {R₁} {R₂} = ⊑-begin
      par R₁ R₂ ⨾ Q₂.ρ
    ≈⟨ ⨾-cong₁ par-def′ ⟩
      (Q₁.π ⨾ R₁ ⨾ Q₂.π ˘ ⊓ Q₁.ρ ⨾ R₂ ⨾ Q₂.ρ ˘) ⨾ Q₂.ρ
    ⊑⟨ ⨾-monotone₁ ⊓-lower₂ ⟨⊑≈⟩ ⨾-assoc₃₊₁ ⟩
      Q₁.ρ ⨾ R₂ ⨾ Q₂.ρ ˘ ⨾ Q₂.ρ
    ⊑⟨ ⨾-monotone₂ (proj₂ Q₂.ρ.univalent) ⟩
      Q₁.ρ ⨾ R₂
    □
  fork⨾⊗-⊑ :  {A : Obj} {R₁ : Mor A B₁} {R₂ : Mor A B₂} {S₁ : Mor B₁ C₁} {S₂ : Mor B₂ C₂}
          → Q₁.fork R₁ R₂ ⨾ par S₁ S₂ ⊑ Q₂.fork (R₁ ⨾ S₁) (R₂ ⨾ S₂)
  fork⨾⊗-⊑ {A} {R₁} {R₂} {S₁} {S₂} = ⊑-begin
      Q₁.fork R₁ R₂ ⨾ par S₁ S₂
    ≈⟨ ⨾-cong₂ par-def′ ⟩
      Q₁.fork R₁ R₂ ⨾ (Q₁.π ⨾ S₁ ⨾ Q₂.π ˘ ⊓ Q₁.ρ ⨾ S₂ ⨾ Q₂.ρ ˘)
    ⊑⟨ ⨾-⊓-subdistribR ⟨⊑≈⟩ ⊓-cong ⨾-assocL ⨾-assocL ⟩
      (Q₁.fork R₁ R₂ ⨾ Q₁.π) ⨾ S₁ ⨾ Q₂.π ˘ ⊓ (Q₁.fork R₁ R₂ ⨾ Q₁.ρ) ⨾ S₂ ⨾ Q₂.ρ ˘
    ⊑⟨ ⊓-monotone (⨾-monotone₁ Q₁.fork⨾π-⊑ ⟨⊑≈⟩ ⨾-assocL) (⨾-monotone₁ Q₁.fork⨾ρ-⊑ ⟨⊑≈⟩ ⨾-assocL) ⟩
      (R₁ ⨾ S₁) ⨾ Q₂.π ˘ ⊓ (R₂ ⨾ S₂) ⨾ Q₂.ρ ˘
    ≈˘⟨ Q₂.fork-def ⟩
      Q₂.fork (R₁ ⨾ S₁) (R₂ ⨾ S₂)
    □
  ⊗-⊓-⊗ : {R₁ S₁ : Mor B₁ C₁} {R₂ S₂ : Mor B₂ C₂}
          → par R₁ R₂ ⊓ par S₁ S₂ ≈ par (R₁ ⊓ S₁) (R₂ ⊓ S₂)
```

```
  ⊗-⊓-⊗ {R₁} {S₁} {R₂} {S₂} = ≈-begin
      par R₁ R₂ ⊓ par S₁ S₂
    ≈⟨ ⊓-cong par-def′ par-def′ ⟩
      (Q₁.π ⨾ R₁ ⨾ Q₂.π ˘ ⊓ Q₁.ρ ⨾ R₂ ⨾ Q₂.ρ ˘) ⊓ (Q₁.π ⨾ S₁ ⨾ Q₂.π ˘ ⊓ Q₁.ρ ⨾ S₂ ⨾ Q₂.ρ ˘)
    ≈⟨ ⊓-transpose₂ ⟩
      (Q₁.π ⨾ R₁ ⨾ Q₂.π ˘ ⊓ Q₁.π ⨾ S₁ ⨾ Q₂.π ˘) ⊓ (Q₁.ρ ⨾ R₂ ⨾ Q₂.ρ ˘ ⊓ Q₁.ρ ⨾ S₂ ⨾ Q₂.ρ ˘)
    ≈˘⟨ ⊓-cong (⨾-⊓-distribR Q₁.π.univalent) (⨾-⊓-distribR Q₁.ρ.univalent) ⟩
      Q₁.π ⨾ (R₁ ⨾ Q₂.π ˘ ⊓ S₁ ⨾ Q₂.π ˘) ⊓ Q₁.ρ ⨾ (R₂ ⨾ Q₂.ρ ˘ ⊓ S₂ ⨾ Q₂.ρ ˘)
    ≈˘⟨ ⊓-cong (⨾-cong₂ (⨾˘-⊓-distribL Q₂.π.univalent)) (⨾-cong₂ (⨾˘-⊓-distribL Q₂.ρ.univalent)) ⟩
      Q₁.π ⨾ (R₁ ⊓ S₁) ⨾ Q₂.π ˘ ⊓ Q₁.ρ ⨾ (R₂ ⊓ S₂) ⨾ Q₂.ρ ˘
    ≈˘⟨ par-def′ ⟩
      par (R₁ ⊓ S₁) (R₂ ⊓ S₂)
    □
open TwoTabulations public using (TabulatedParComp; module TabulatedParComp)
```

[**WK:** *Add diagrams in many places here!*]

```
module TwoTabulations-IdL {B₁ B₂ C₂ : Obj} {Q₁ : Mor B₁ B₂} {Q₂ : Mor B₁ C₂}
                          {Q₁-tab : Tabulation Q₁}
                          {Q₂-tab : Tabulation Q₂}
                          (parComp : TabulatedParComp Q₁-tab Q₂-tab) where
  private
    module Q₁ = Tabulation Q₁-tab
    module Q₂ = Tabulation Q₂-tab
  open TabulatedParComp parComp

  Id⊗₀⨾ρ : {R₂ : Mor B₂ C₂} → par Id R₂ ⨾ Q₂.ρ ≈ Q₁.π ⨾ Q₂ ⊓ Q₁.ρ ⨾ R₂
  Id⊗₀⨾ρ {R₂} = ≈-begin
      par Id R₂ ⨾ Q₂.ρ
    ≈⟨ ⨾-cong₁ (par-def ⟨≈≈⟩ Q₂.fork-cong₁ rightId) ⟩
      Q₂.fork Q₁.π (Q₁.ρ ⨾ R₂) ⨾ Q₂.ρ
    ≈⟨ Q₂.fork⨾ρ ⟩
      Q₁.π ⨾ Q₂ ⊓ Q₁.ρ ⨾ R₂
    □


module TwoTabulations-IdR {B₁ B₂ C₁ : Obj} {Q₁ : Mor B₁ B₂} {Q₂ : Mor C₁ B₂}
                          {Q₁-tab : Tabulation Q₁}
                          {Q₂-tab : Tabulation Q₂}
                          (parComp : TabulatedParComp Q₁-tab Q₂-tab) where
  private
    module Q₁ = Tabulation Q₁-tab
    module Q₂ = Tabulation Q₂-tab
  open TabulatedParComp parComp

  ⊗₀Id⨾π : {R₁ : Mor B₁ C₁} → par R₁ Id ⨾ Q₂.π ≈ Q₁.π ⨾ R₁ ⊓ Q₁.ρ ⨾ Q₂ ˘
  ⊗₀Id⨾π {R₁} = ≈-begin
      par R₁ Id ⨾ Q₂.π
    ≈⟨ ⨾-cong₁ (par-def ⟨≈≈⟩ Q₂.fork-cong₂ rightId) ⟩
      Q₂.fork (Q₁.π ⨾ R₁) (Q₁.ρ) ⨾ Q₂.π
    ≈⟨ Q₂.fork⨾π ⟩
      Q₁.π ⨾ R₁ ⊓ Q₁.ρ ⨾ Q₂ ˘
    □


module TwoTabulationsBidir {B₁ B₂ C₁ C₂ : Obj} {Q₁ : Mor B₁ B₂} {Q₂ : Mor C₁ C₂}
                           {Q₁-tab : Tabulation Q₁}
                           {Q₂-tab : Tabulation Q₂}
```

```
                                  (parComp₁→₂ : TabulatedParComp Q₁-tab Q₂-tab)
                                  (parComp₂→₁ : TabulatedParComp Q₂-tab Q₁-tab) where
  private
    module Q₁ = Tabulation Q₁-tab
    module Q₂ = Tabulation Q₂-tab
    module Q₁→Q₂ = TabulatedParComp parComp₁→₂
    module Q₂→Q₁ = TabulatedParComp parComp₂→₁

  ⊗-˘ : {R₁ : Mor B₁ C₁} {R₂ : Mor B₂ C₂} → (Q₁→Q₂.par R₁ R₂) ˘ ≈ Q₂→Q₁.par (R₁ ˘) (R₂ ˘)
  ⊗-˘ {R₁} {R₂} = ≈-begin
      Q₁→Q₂.par R₁ R₂ ˘
    ≈⟨ ˘-cong Q₁→Q₂.par-def ⟩
      Q₂.fork (Q₁.π ⨾ R₁) (Q₁.ρ ⨾ R₂) ˘
    ≈⟨ ˘-cong Q₂.fork-def ⟩
      ((Q₁.π ⨾ R₁) ⨾ Q₂.π ˘ ⊓ (Q₁.ρ ⨾ R₂) ⨾ Q₂.ρ ˘) ˘
    ≈⟨ ˘-⊓-distrib ⟨≈≈⟩ ⊓-cong ⨾˘-˘ ⨾˘-˘ ⟩
      Q₂.π ⨾ (Q₁.π ⨾ R₁) ˘ ⊓ Q₂.ρ ⨾ (Q₁.ρ ⨾ R₂) ˘
    ≈⟨ ⊓-cong (⨾-cong₂ ⨾-˘ ⟨≈≈⟩ ⨾-assocL) (⨾-cong₂ ⨾-˘ ⟨≈≈⟩ ⨾-assocL) ⟩
      (Q₂.π ⨾ R₁ ˘) ⨾ Q₁.π ˘ ⊓ (Q₂.ρ ⨾ R₂ ˘) ⨾ Q₁.ρ ˘
    ≈˘⟨ Q₁.fork-def ⟩
      Q₁.fork (Q₂.π ⨾ R₁ ˘) (Q₂.ρ ⨾ R₂ ˘)
    ≈˘⟨ Q₂→Q₁.par-def ⟩
      Q₂→Q₁.par (R₁ ˘) (R₂ ˘)
    □

module TwoTabulationsBidir-IdL {B₁ B₂ C₂ : Obj} {Q₁ : Mor B₁ B₂} {Q₂ : Mor B₁ C₂}
                                  {Q₁-tab : Tabulation Q₁}
                                  {Q₂-tab : Tabulation Q₂}
                                  (parComp₁→₂ : TabulatedParComp Q₁-tab Q₂-tab)
                                  (parComp₂→₁ : TabulatedParComp Q₂-tab Q₁-tab) where
  private
    module Q₁ = Tabulation Q₁-tab
    module Q₂ = Tabulation Q₂-tab
    module Q₁→Q₂ = TabulatedParComp parComp₁→₂
    module Q₂→Q₁ = TabulatedParComp parComp₂→₁
  open TwoTabulationsBidir parComp₂→₁ parComp₁→₂
  open TwoTabulations-IdL parComp₂→₁

  ρ˘⨾Id⊗ : {R₂ : Mor B₂ C₂} → Q₁.ρ ˘ ⨾ Q₁→Q₂.par Id R₂ ≈ Q₁ ˘ ⨾ Q₂.π ˘ ⊓ R₂ ⨾ Q₂.ρ ˘
  ρ˘⨾Id⊗ {R₂} = ≈-begin
      Q₁.ρ ˘ ⨾ Q₁→Q₂.par Id R₂
    ≈⟨ ⨾-cong₂ (Q₁→Q₂.⊗-cong Id˘ ˘˘ ⟨≈˘≈˘⟩ ⊗-˘) ⟩
      Q₁.ρ ˘ ⨾ (Q₂→Q₁.par Id (R₂ ˘)) ˘
    ≈˘⟨ ⨾-˘ ⟩
      (Q₂→Q₁.par Id (R₂ ˘) ⨾ Q₁.ρ) ˘
    ≈⟨ ˘-cong Id⊗₀⨾ρ ⟩
      (Q₂.π ⨾ Q₁ ⊓ Q₂.ρ ⨾ R₂ ˘) ˘
    ≈⟨ ˘-⊓-distrib ⟨≈≈⟩ ⊓-cong ⨾-˘ ⨾˘-˘ ⟩
      Q₁ ˘ ⨾ Q₂.π ˘ ⊓ R₂ ⨾ Q₂.ρ ˘
    □

    -- Generalisation of (Bird and de Moor, 1997, (5.5))
  fork⨾Id⊗-⊒ : {A : Obj} {R₁ : Mor A B₁} {R₂ : Mor A B₂} {S₂ : Mor B₂ C₂}
             → Q₂ ⨾ S₂ ˘ ⊑ Q₁
             → Q₂.fork R₁ (R₂ ⨾ S₂) ⊑ Q₁.fork R₁ R₂ ⨾ Q₁→Q₂.par Id S₂
  fork⨾Id⊗-⊒ {A} {R₁} {R₂} {S₂} Q₂⨾S₂˘⊑Q₁ = ⊑-begin
      Q₂.fork R₁ (R₂ ⨾ S₂)
    ≈⟨ Q₂.fork-def ⟨≈≈⟩ ⊓-cong₂ ⨾-assoc ⟩
      R₁ ⨾ Q₂.π ˘ ⊓ R₂ ⨾ S₂ ⨾ Q₂.ρ ˘
```

```
  ≈˘⟨ ⊓-cong₂ (⨾-cong₂ (⊑-to-⊓₂ (swap-⨾-⊑-total Q₂.π.total (⊑-begin
          (S₂ ⨾ Q₂.ρ ˘) ⨾ Q₂.π
        ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ Q₂.ρ˘⨾π ⟩
          S₂ ⨾ Q₂ ˘
        ⊑⟨ ⊑-˘-swap (⨾˘-˘ ⟨≈⊑⟩ Q₂⨾S₂˘⊑Q₁) ⟩
          Q₁ ˘
        □)))) ⟩
    R₁ ⨾ Q₂.π ˘ ⊓ R₂ ⨾ (Q₁ ˘ ⨾ Q₂.π ˘ ⊓ S₂ ⨾ Q₂.ρ ˘)
  ≈⟨ ⊓-cong₂ (⨾-cong₂ ρ˘⨾Id⊗ ⟨≈˘≈⟩ ⨾-assocL) ⟩
    R₁ ⨾ Q₂.π ˘ ⊓ (R₂ ⨾ Q₁.ρ ˘) ⨾ Q₁→Q₂.par Id S₂
  ⊑⟨ modal₂″ ⟨⊑≈⟩ ⨾-cong₁ (⊓-cong₁ ⨾-assoc) ⟩
    (R₁ ⨾ Q₂.π ˘ ⨾ (Q₁→Q₂.par Id S₂) ˘ ⊓ R₂ ⨾ Q₁.ρ ˘) ⨾ Q₁→Q₂.par Id S₂
  ⊑⟨ ⨾-monotone₁ (⊓-monotone₁ (⨾-monotone₂ (⊑-begin
      Q₂.π ˘ ⨾ (Q₁→Q₂.par Id S₂) ˘
    ≈˘⟨ ⨾-˘ ⟩
      (Q₁→Q₂.par Id S₂ ⨾ Q₂.π) ˘
    ⊑⟨ ˘-monotone (Q₁→Q₂.⊗⨾π-⊑ ⟨⊑≈⟩ rightId) ⟩
      Q₁.π ˘
    □))) ⟩
    (R₁ ⨾ Q₁.π ˘ ⊓ R₂ ⨾ Q₁.ρ ˘) ⨾ Q₁→Q₂.par Id S₂
  ≈˘⟨ ⨾-cong₁ Q₁.fork-def ⟩
    Q₁.fork R₁ R₂ ⨾ Q₁→Q₂.par Id S₂
  □

module TwoTabulationsBidir-IdR {B₁ B₂ C₁ : Obj} {Q₁ : Mor B₁ B₂} {Q₂ : Mor C₁ B₂}
                               {Q₁-tab : Tabulation Q₁}
                               {Q₂-tab : Tabulation Q₂}
                               (parComp₁→₂ : TabulatedParComp Q₁-tab Q₂-tab)
                               (parComp₂→₁ : TabulatedParComp Q₂-tab Q₁-tab) where
  private
    module Q₁ = Tabulation Q₁-tab
    module Q₂ = Tabulation Q₂-tab
    module Q₁→Q₂ = TabulatedParComp parComp₁→₂
    module Q₂→Q₁ = TabulatedParComp parComp₂→₁
  open TwoTabulationsBidir parComp₂→₁ parComp₁→₂
  open TwoTabulations-IdR parComp₂→₁

  π˘⨾⊗Id : {R₁ : Mor B₁ C₁} → Q₁.π ˘ ⨾ Q₁→Q₂.par R₁ Id ≈ R₁ ⨾ Q₂.π ˘ ⊓ Q₁ ⨾ Q₂.ρ ˘
  π˘⨾⊗Id {R₁} = ≈-begin
      Q₁.π ˘ ⨾ Q₁→Q₂.par R₁ Id
    ≈⟨ ⨾-cong₂ (Q₁→Q₂.⊗-cong ˘˘ Id˘ ⟨≈˘≈⟩ ⊗-˘) ⟩
      Q₁.π ˘ ⨾ (Q₂→Q₁.par (R₁ ˘) Id) ˘
    ≈˘⟨ ⨾-˘ ⟩
      ((Q₂→Q₁.par (R₁ ˘) Id) ⨾ Q₁.π) ˘
    ≈⟨ ˘-cong ⊗₀Id⨾π ⟩
      (Q₂.π ⨾ R₁ ˘ ⊓ Q₂.ρ ⨾ Q₁ ˘) ˘
    ≈⟨ ˘-⊓-distrib ⟨≈≈⟩ ⊓-cong ⨾˘-˘ ⨾˘-˘ ⟩
      R₁ ⨾ Q₂.π ˘ ⊓ Q₁ ⨾ Q₂.ρ ˘
    □

    -- Generalisation of (Bird and de Moor, 1997, (5.4))
  fork⨾⊗Id-⊒ :     {A : Obj} {R₁ : Mor A B₁} {R₂ : Mor A B₂} {S₁ : Mor B₁ C₁}
               → S₁ ⨾ Q₂ ⊑ Q₁
               → Q₂.fork (R₁ ⨾ S₁) R₂ ⊑ Q₁.fork R₁ R₂ ⨾ Q₁→Q₂.par S₁ Id
  fork⨾⊗Id-⊒ {A} {R₁} {R₂} {S₁} S₁⨾Q₂⊑Q₁ = ⊑-begin
      Q₂.fork (R₁ ⨾ S₁) R₂
    ≈⟨ Q₂.fork-def ⟨≈≈⟩ ⊓-cong₁ ⨾-assoc ⟩
      R₁ ⨾ S₁ ⨾ Q₂.π ˘ ⊓ R₂ ⨾ Q₂.ρ ˘
```

```
      ≈˘⟨ ⊓-cong₁ (⨾-cong₂ (⊑-to-⊓₁ (swap-⨾-⊑-total Q₂.ρ.total (⊑-begin
              (S₁ ⨾ Q₂.π ˘) ⨾ Q₂.ρ
            ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ Q₂.π˘⨾ρ ⟩
              S₁ ⨾ Q₂
            ⊑⟨ S₁⨾Q₂⊑Q₁ ⟩
              Q₁
            □)))) ⟩
        R₁ ⨾ (S₁ ⨾ Q₂.π ˘ ⊓ Q₁ ⨾ Q₂.ρ ˘) ⊓ R₂ ⨾ Q₂.ρ ˘
      ≈⟨ ⊓-cong₁ (⨾-cong₂ π˘⨾⊗Id ⟨≈˘≈⟩ ⨾-assocL) ⟩
        (R₁ ⨾ Q₁.π ˘) ⨾ Q₁→Q₂.par S₁ Id ⊓ R₂ ⨾ Q₂.ρ ˘
      ⊑⟨ modal₂ ⟨⊑≈⟩ ⨾-cong₁ (⊓-cong₂ ⨾-assoc) ⟩
        (R₁ ⨾ Q₁.π ˘ ⊓ R₂ ⨾ Q₂.ρ ˘ ⨾ (Q₁→Q₂.par S₁ Id) ˘) ⨾ (Q₁→Q₂.par S₁ Id)
      ⊑⟨ ⨾-monotone₁ (⊓-monotone₂ (⨾-monotone₂ (⊑-begin
          Q₂.ρ ˘ ⨾ (Q₁→Q₂.par S₁ Id) ˘
        ≈˘⟨ ⨾-˘ ⟩
          (Q₁→Q₂.par S₁ Id ⨾ Q₂.ρ) ˘
        ⊑⟨ ˘-monotone (Q₁→Q₂.⊗⨾ρ-⊑ ⟨⊑≈⟩ rightId) ⟩
          Q₁.ρ ˘
        □))) ⟩
        (R₁ ⨾ Q₁.π ˘ ⊓ R₂ ⨾ Q₁.ρ ˘) ⨾ Q₁→Q₂.par S₁ Id
      ≈˘⟨ ⨾-cong₁ Q₁.fork-def ⟩
        Q₁.fork R₁ R₂ ⨾ Q₁→Q₂.par S₁ Id
      □


module ThreeTabulationsBidir-IdL-IdR {B₁ B₂ C₁ C₂ : Obj} {Q₂ : Mor B₁ B₂} {Q₂₃ : Mor B₁ C₂} {Q₃ : Mor C₁ C₂}
                                     {Q₂-tab : Tabulation Q₂}
                                     {Q₂₃-tab : Tabulation Q₂₃}
                                     {Q₃-tab : Tabulation Q₃}
                                     (parComp₂→₂₃ : TabulatedParComp Q₂-tab Q₂₃-tab)
                                     (parComp₂₃→₂ : TabulatedParComp Q₂₃-tab Q₂-tab)
                                     (parComp₂₃→₃ : TabulatedParComp Q₂₃-tab Q₃-tab)
                                     (parComp₃→₂₃ : TabulatedParComp Q₃-tab Q₂₃-tab)
                                     (parComp₂→₃ : TabulatedParComp Q₂-tab Q₃-tab) where
  private
    module Q₂ = Tabulation Q₂-tab
    module Q₂₃ = Tabulation Q₂₃-tab
    module Q₃ = Tabulation Q₃-tab
    module Q₂→Q₂₃ = TabulatedParComp parComp₂→₂₃
    module Q₂₃→Q₃ = TabulatedParComp parComp₂₃→₃
    module Q₂→Q₃ = TabulatedParComp parComp₂→₃
  open TwoTabulationsBidir parComp₂₃→₂ parComp₂→₂₃ using ()
  open TwoTabulations-IdL parComp₂₃→₂ using ()
  open TwoTabulationsBidir-IdR parComp₂₃→₃ parComp₃→₂₃ using (fork⨾⊗Id-⊒)
  open TwoTabulationsBidir-IdL parComp₂→₂₃ parComp₂₃→₂ using (fork⨾Id⊗-⊒)

  Id⊗-⨾-⊗Id-⊑ : {R₁ : Mor B₁ C₁} {R₂ : Mor B₂ C₂}
              → Q₂→Q₂₃.par Id R₂ ⨾ Q₂₃→Q₃.par R₁ Id ⊑ Q₂→Q₃.par R₁ R₂
  Id⊗-⨾-⊗Id-⊑ {R₁} {R₂} = ⊑-begin
      Q₂→Q₂₃.par Id R₂ ⨾ Q₂₃→Q₃.par R₁ Id
    ≈⟨ ⨾-cong Q₂→Q₂₃.par-def Q₂₃→Q₃.par-def′ ⟩
      Q₂₃.fork (Q₂.π ⨾ Id) (Q₂.ρ ⨾ R₂) ⨾ ((Q₂₃.π ⨾ R₁ ⨾ Q₃.π ˘) ⊓ (Q₂₃.ρ ⨾ Id ⨾ Q₃.ρ ˘))
    ≈⟨ ⨾-cong (Q₂₃.fork-cong₁ rightId) (⊓-cong₂ (⨾-cong₂ leftId)) ⟩
      Q₂₃.fork Q₂.π (Q₂.ρ ⨾ R₂) ⨾ ((Q₂₃.π ⨾ R₁ ⨾ Q₃.π ˘) ⊓ (Q₂₃.ρ ⨾ Q₃.ρ ˘))
    ⊑⟨ ⨾-⊓-subdistribR ⟨⊑≈⟩ ⊓-cong ⨾-assocL ⨾-assocL ⟩
      (Q₂₃.fork Q₂.π (Q₂.ρ ⨾ R₂) ⨾ Q₂₃.π) ⨾ R₁ ⨾ Q₃.π ˘ ⊓ (Q₂₃.fork Q₂.π (Q₂.ρ ⨾ R₂) ⨾ Q₂₃.ρ) ⨾ Q₃.ρ ˘
    ⊑⟨ ⊓-monotone (⨾-monotone₁ Q₂₃.fork⨾π-⊑) (⨾-monotone₁ Q₂₃.fork⨾ρ-⊑ ⟨⊑≈⟩ ⨾-assoc) ⟩
      Q₂.π ⨾ R₁ ⨾ Q₃.π ˘ ⊓ Q₂.ρ ⨾ R₂ ⨾ Q₃.ρ ˘
```

```
    ≈˘⟨ Q2→Q3.par-def′ ⟩
      Q2→Q3.par R1 R2
    □

  fork⨾⊗-⊒ :  {A : Obj} {R1 : Mor A B1} {R2 : Mor A B2} {S1 : Mor B1 C1} {S2 : Mor B2 C2}
           →  S1 ⨾ Q3 ⊑ Q23
           →  Q23 ⨾ S2 ˘ ⊑ Q2
           →  Q3.fork (R1 ⨾ S1) (R2 ⨾ S2) ⊑ Q2.fork R1 R2 ⨾ Q2→Q3.par S1 S2
  fork⨾⊗-⊒ {A} {R1} {R2} {S1} {S2} S1⨾Q3⊑Q23 Q23⨾S2˘⊑Q2 = ⊑-begin
      Q3.fork (R1 ⨾ S1) (R2 ⨾ S2)
    ⊑⟨ fork⨾⊗Id-⊒ S1⨾Q3⊑Q23 ⟩
      Q23.fork R1 (R2 ⨾ S2) ⨾ Q23→Q3.par S1 Id
    ⊑⟨ ⨾-monotone1 (fork⨾Id⊗-⊒ Q23⨾S2˘⊑Q2) ⟨⊑≈⟩ ⨾-assoc ⟩
      Q2.fork R1 R2 ⨾ Q2→Q23.par Id S2 ⨾ Q23→Q3.par S1 Id
    ⊑⟨ ⨾-monotone2 Id⊗-⨾-⊗Id-⊑ ⟩
      Q2.fork R1 R2 ⨾ Q2→Q3.par S1 S2
    □

    -- Generalisation of (Bird and de Moor, 1997, (5.3))
  fork⨾⊗ : {A : Obj} {R1 : Mor A B1} {R2 : Mor A B2} {S1 : Mor B1 C1} {S2 : Mor B2 C2}
         → S1 ⨾ Q3 ⊑ Q23
         → Q23 ⨾ S2 ˘ ⊑ Q2
         → Q2.fork R1 R2 ⨾ Q2→Q3.par S1 S2 ≈ Q3.fork (R1 ⨾ S1) (R2 ⨾ S2)
  fork⨾⊗ S1⨾Q3⊑Q23 Q23⨾S2˘⊑Q2 = ⊑-antisym Q2→Q3.fork⨾⊗-⊑ (fork⨾⊗-⊒ S1⨾Q3⊑Q23 Q23⨾S2˘⊑Q2)

  module TabulatedParCompFunctoriality {A1 A2 : Obj} {Q1 : Mor A1 A2}
                                       {Q1-tab : Tabulation Q1}
                                       (parComp1→2 : TabulatedParComp Q1-tab Q2-tab)
                                       (parComp1→3 : TabulatedParComp Q1-tab Q3-tab) where
    private
      module Q1 = Tabulation Q1-tab
      module Q1→Q2 = TabulatedParComp parComp1→2
      module Q1→Q3 = TabulatedParComp parComp1→3

    ⨾-⊗-⨾ :   {R1 : Mor A1 B1} {R2 : Mor A2 B2} {S1 : Mor B1 C1} {S2 : Mor B2 C2}
        → S1 ⨾ Q3 ⊑ Q23
        → Q23 ⨾ S2 ˘ ⊑ Q2
        → Q1→Q3.par (R1 ⨾ S1) (R2 ⨾ S2) ≈ Q1→Q2.par R1 R2 ⨾ Q2→Q3.par S1 S2
    ⨾-⊗-⨾ {R1} {R2} {S1} {S2} S1⨾Q3⊑Q23 Q23⨾S2˘⊑Q2 = ≈-begin
        Q1→Q3.par (R1 ⨾ S1) (R2 ⨾ S2)
      ≈⟨ Q1→Q3.par-def ⟩
        Q3.fork (Q1.π ⨾ R1 ⨾ S1) (Q1.ρ ⨾ R2 ⨾ S2)
      ≈⟨ Q3.fork-cong ⨾-assocL ⨾-assocL ⟨≈≈˘⟩ fork⨾⊗ S1⨾Q3⊑Q23 Q23⨾S2˘⊑Q2 ⟩
        Q2.fork (Q1.π ⨾ R1) (Q1.ρ ⨾ R2) ⨾ Q2→Q3.par S1 S2
      ≈˘⟨ ⨾-cong1 Q1→Q2.par-def ⟩
        Q1→Q2.par R1 R2 ⨾ Q2→Q3.par S1 S2
      □


record HasTabulations : Set (i ⊍ j ⊍ k1) where
  field
    tabulate : {A B : Obj} (Q : Mor A B) → Tabulation Q
  open module Tab {A B : Obj} (Q : Mor A B) = Tabulation (tabulate Q)
    using (obj; fork; π; ρ) public
  open module Tab′ {A B : Obj} {Q : Mor A B} = Tabulation (tabulate Q)
    hiding (obj; fork; π; ρ) public
  field
    par : {A1 A2 B1 B2 : Obj} (Q1 : Mor A1 A2) (Q2 : Mor B1 B2) → Mor A1 B1 → Mor A2 B2 → Mor (obj Q1) (obj Q2)
    par-def : {A1 A2 B1 B2 : Obj} {Q1 : Mor A1 A2} {Q2 : Mor B1 B2} {R : Mor A1 B1} {S : Mor A2 B2}
            → par Q1 Q2 R S ≈ fork Q2 (π Q1 ⨾ R) (ρ Q1 ⨾ S)
```

```
  module _ {B₁ B₂ C₁ C₂ : Obj} (Q₁ : Mor B₁ B₂) (Q₂ : Mor C₁ C₂) where
    tabulatedParComp : TabulatedParComp (tabulate Q₁) (tabulate Q₂)
    tabulatedParComp = record {par = par Q₁ Q₂; par-def = par-def}
  module _ {B₁ B₂ C₁ C₂ : Obj} {Q₁ : Mor B₁ B₂} {Q₂ : Mor C₁ C₂} where
    open TabulatedParComp (tabulatedParComp Q₁ Q₂) public hiding (par; par-def)
  module _ {B₁ B₂ C₂ : Obj} {Q₁ : Mor B₁ B₂} {Q₂ : Mor B₁ C₂} where
    open TwoTabulations-IdL (tabulatedParComp Q₁ Q₂) public
    open TwoTabulationsBidir-IdL (tabulatedParComp Q₁ Q₂) (tabulatedParComp Q₂ Q₁) public
  module _ {B₁ B₂ C₁ : Obj} {Q₁ : Mor B₁ B₂} {Q₂ : Mor C₁ B₂} where
    open TwoTabulations-IdR (tabulatedParComp Q₁ Q₂) public
    open TwoTabulationsBidir-IdR (tabulatedParComp Q₁ Q₂) (tabulatedParComp Q₂ Q₁) public
  module _ {B₁ B₂ C₁ C₂ : Obj} {Q₁ : Mor B₁ B₂} {Q₂ : Mor C₁ C₂} where
    open TwoTabulationsBidir (tabulatedParComp Q₁ Q₂) (tabulatedParComp Q₂ Q₁) public
  module _ {B₁ B₂ C₁ C₂ : Obj} {Q₂ : Mor B₁ B₂} {Q₂₃ : Mor B₁ C₂} {Q₃ : Mor C₁ C₂} where
    open ThreeTabulationsBidir-IdL-IdR (tabulatedParComp Q₂ Q₂₃) (tabulatedParComp Q₂₃ Q₂)
                                       (tabulatedParComp Q₂₃ Q₃) (tabulatedParComp Q₃ Q₂₃)
                                       (tabulatedParComp Q₂ Q₃) public
    module _ {A₁ A₂ : Obj} {Q₁ : Mor A₁ A₂} where
      open TabulatedParCompFunctoriality (tabulatedParComp Q₁ Q₂) (tabulatedParComp Q₁ Q₃) public
  Id-⊗-Id : {A₁ A₂ : Obj} {Q₁ : Mor A₁ A₂} → par Q₁ Q₁ Id Id ≈ Id
  Id-⊗-Id {A₁} {A₂} {Q₁} = ≈-begin
      par Q₁ Q₁ Id Id
    ≈⟨ par-def ⟨≈≈⟩ fork-cong rightId rightId ⟩
      fork Q₁ (π Q₁) (ρ Q₁)
    ≈⟨ fork-def ⟩
      π Q₁ ⨾ π Q₁ ˘ ⊓ ρ Q₁ ⨾ ρ Q₁ ˘
    ≈⟨ extensionality ⟩
      Id
    □
```

4.4 Categoric.Allegory.DirectProduct

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import Categoric.Allegory
import Categoric.Allegory.Tabulation
open import Categoric.TopMor
```

We introduce direct products as a special case of tabulations, and re-use the development from Categoric.Allegory.Tabulation (Sect. 4.3).

```
module Categoric.Allegory.DirectProduct
  {i j k₁ k₂ : Level} {Obj : Set i}
  (𝒜 : Allegory j k₁ k₂ Obj)
  (let open Allegory 𝒜)
  (hasTopMors : HasTopMors orderedSemigroupoid)
  where
open HasTopMors hasTopMors
open import Categoric.Allegory.TopMor 𝒜 hasTopMors
module Tab = Categoric.Allegory.Tabulation 𝒜
open Tab hiding (module DefaultFork)

record IsDirectProduct (A B P : Obj) : Set (i ⊍ j ⊍ k₁) where
  infixr 5 _▽_
```

```
field
  π : Mor P A
  ρ : Mor P B
  tabulates-⊤ : IsTabulation ⊤ π ρ
⊤-tabulation : Tabulation (⊤ {A} {B})
⊤-tabulation = record {obj = P; π = π; ρ = ρ; isTabulation = tabulates-⊤}
private module T = IsTabulation tabulates-⊤
open T using (fork; fork⨾π; fork⨾ρ)
open T public hiding (fork; ▽Id⨾π; Id▽⨾ρ; ρ˘⨾π) renaming
  (fork-def to ▽-def
  ; extensionality to ⊠-extensionality
  ; fork-cong to ▽-cong
  ; fork-cong₁ to ▽-cong₁
  ; fork-cong₂ to ▽-cong₂
  ; fork-monotone to ▽-monotone
  ; fork-monotone₁ to ▽-monotone₁
  ; fork-monotone₂ to ▽-monotone₂
  ; fork⨾π-⊑ to ▽⨾π-⊑   -- (R ▽ S) ⨾ π ⊑ R
  ; fork⨾ρ-⊑ to ▽⨾ρ-⊑   -- (R ▽ S) ⨾ ρ ⊑ S
  )
ρ˘⨾π : ρ ˘ ⨾ π ≈ ⊤
ρ˘⨾π = T.ρ˘⨾π ⟨≈≈⟩ ⊤˘
_▽_ : {Z : Obj} → Mor Z A → Mor Z B → Mor Z P
_▽_ = fork
  -- (Bird and de Moor, 1997, (5.6))
▽⨾π : {Z : Obj} {R : Mor Z A} {S : Mor Z B} → (R ▽ S) ⨾ π ≈ dom S ⨾ R
▽⨾π {Z} {R} {S} = ≈-begin
    (R ▽ S) ⨾ π
  ≈⟨ fork⨾π ⟨≈≈⟩ ⊓-cong₂ (⨾-cong₂ ⊤˘) ⟩
    R ⊓ S ⨾ ⊤
  ≈˘⟨ dom⨾-≈-⊓-⨾⊤ ⟩
    dom S ⨾ R
  □
  -- (Bird and de Moor, 1997, (5.7))
▽⨾ρ : {Z : Obj} {R : Mor Z A} {S : Mor Z B} → (R ▽ S) ⨾ ρ ≈ dom R ⨾ S
▽⨾ρ {Z} {R} {S} = ≈-begin
    (R ▽ S) ⨾ ρ
  ≈⟨ fork⨾ρ ⟩
    R ⨾ ⊤ ⊓ S
  ≈⟨ ⊓-commutative ⟨≈≈˘⟩ dom⨾-≈-⊓-⨾⊤ ⟩
    dom R ⨾ S
  □
▽Id⨾π : {R : Mor B A} → (R ▽ Id) ⨾ π ≈ R
▽Id⨾π {R} = ≈-begin
    (R ▽ Id) ⨾ π
  ≈⟨ ▽⨾π ⟩
    dom Id ⨾ R
  ≈⟨ ⨾-cong₁ dom-Id ⟨≈≈⟩ leftId ⟩
    R
  □
Id▽⨾ρ : {S : Mor A B} → (Id ▽ S) ⨾ ρ ≈ S
Id▽⨾ρ {S} = ≈-begin
    (Id ▽ S) ⨾ ρ
  ≈⟨ ▽⨾ρ ⟩
    dom Id ⨾ S
```

```
  ≈⟨ ⨾-cong₁ dom-Id ⟨≈≈⟩ leftId ⟩
    S
  □
```

Since DefaultFork will be needed inside the IsTabulation record for tabulates-⊤, we should not rename there.

```
module DefaultFork = Tab.DefaultFork   -- renaming (fork to _▽_; fork-def to ▽-def)


record HasDirectProducts : Set (i ⊍ j ⊍ k₁) where
  infixr 3 _⊠_
  field
    _⊠_ : Obj → Obj → Obj
    isDirectProduct : (A B : Obj) → IsDirectProduct A B (A ⊠ B)
  open module IsDirProd {A B : Obj} = IsDirectProduct (isDirectProduct A B) public
  infixr 5 _⊗_
  field
    _⊗_ : {A₁ A₂ B₁ B₂ : Obj} → Mor A₁ B₁ → Mor A₂ B₂ → Mor (A₁ ⊠ A₂) (B₁ ⊠ B₂)
    ⊗-def : {A₁ A₂ B₁ B₂ : Obj} {R : Mor A₁ B₁} {S : Mor A₂ B₂} → R ⊗ S ≈ (π ⨾ R) ▽ (ρ ⨾ S)
  module _ {B₁ B₂ C₁ C₂ : Obj} where
    parComp : TabulatedParComp (⊤-tabulation {B₁} {B₂}) (⊤-tabulation {C₁} {C₂})
    parComp = record {par = _⊗_; par-def = ⊗-def}
    open TabulatedParComp parComp public
      hiding (par; par-def)
      renaming (par-def′ to ⊗-def′; fork⨾⊗-⊑ to ▽⨾⊗-⊑)
  module _ {B₁ B₂ C₂ : Obj} where
    open TwoTabulations-IdL {B₁} {B₂} {C₂} parComp public
    Id⊗⨾ρ : {R₂ : Mor B₂ C₂} → (Id {B₁} ⊗ R₂) ⨾ ρ ≈ ρ ⨾ R₂
    Id⊗⨾ρ = Id⊗₀⨾ρ ⟨≈≈⟩ (⊓-cong₁ (total⨾⊤ π.total) ⟨≈≈⟩ ⊤-⊓)

    open TwoTabulationsBidir-IdL {B₁} {B₂} {C₂} parComp parComp public
    ▽⨾Id⊗-⊒ :  {A : Obj} {R₁ : Mor A B₁} {R₂ : Mor A B₂} {S₂ : Mor B₂ C₂}
               → R₁ ▽ (R₂ ⨾ S₂) ⊑ (R₁ ▽ R₂) ⨾ (Id ⊗ S₂)
    ▽⨾Id⊗-⊒ = fork⨾Id⊗-⊒ ⊑-⊤
  module _ {B₁ B₂ C₁ : Obj} where
    open TwoTabulations-IdR {B₁} {B₂} {C₁} parComp public
    ⊗Id⨾π : {R₁ : Mor B₁ C₁} → (R₁ ⊗ Id {B₂}) ⨾ π ≈ π ⨾ R₁
    ⊗Id⨾π = ⊗₀Id⨾π ⟨≈≈⟩ (⊓-cong₂ (⨾-cong₂ ⊤˘ ⟨≈≈⟩ total⨾⊤ ρ.total) ⟨≈≈⟩ ⊓-⊤)

    open TwoTabulationsBidir-IdR {B₁} {B₂} {C₁} parComp parComp public
    ▽⨾⊗Id-⊒ :  {A : Obj} {R₁ : Mor A B₁} {R₂ : Mor A B₂} {S₁ : Mor B₁ C₁}
               → (R₁ ⨾ S₁) ▽ R₂ ⊑ (R₁ ▽ R₂) ⨾ (S₁ ⊗ Id)
    ▽⨾⊗Id-⊒ = fork⨾⊗Id-⊒ ⊑-⊤
  module _ {B₁ B₂ C₁ C₂ : Obj} where
    open TwoTabulationsBidir {B₁} {B₂} {C₁} {C₂} parComp parComp public
    open ThreeTabulationsBidir-IdL-IdR {B₁} {B₂} {C₁} {C₂} parComp parComp parComp parComp parComp public
    ▽⨾⊗-⊒ : {A : Obj} {R₁ : Mor A B₁} {R₂ : Mor A B₂} {S₁ : Mor B₁ C₁} {S₂ : Mor B₂ C₂}
            → (R₁ ⨾ S₁) ▽ (R₂ ⨾ S₂) ⊑ (R₁ ▽ R₂) ⨾ (S₁ ⊗ S₂)
    ▽⨾⊗-⊒ = fork⨾⊗-⊒ ⊑-⊤ ⊑-⊤
    ▽⨾⊗ :  {A : Obj} {R₁ : Mor A B₁} {R₂ : Mor A B₂} {S₁ : Mor B₁ C₁} {S₂ : Mor B₂ C₂}
         → (R₁ ▽ R₂) ⨾ (S₁ ⊗ S₂) ≈ (R₁ ⨾ S₁) ▽ (R₂ ⨾ S₂)
    ▽⨾⊗ = fork⨾⊗ ⊑-⊤ ⊑-⊤

    module _ {A₁ A₂ : Obj} where
      ⨾-⊗-⨾  : {R₁ : Mor A₁ B₁} {R₂ : Mor A₂ B₂} {S₁ : Mor B₁ C₁} {S₂ : Mor B₂ C₂}
              → (R₁ ⨾ S₁) ⊗ (R₂ ⨾ S₂) ≈ (R₁ ⊗ R₂) ⨾ (S₁ ⊗ S₂)
      ⨾-⊗-⨾  = TabulatedParCompFunctoriality.⨾-⊗-⨾ parComp parComp ⊑-⊤ ⊑-⊤
  Id-⊗-Id : {A₁ A₂ : Obj} → Id {A₁} ⊗ Id {A₂} ≈ Id
  Id-⊗-Id {A₁} {A₂} = ≈-begin
```

```
    Id ⊗ Id
  ≈⟨ ⊗-def ⟨≈≈⟩ ▽-cong rightId rightId ⟩
    π ▽ ρ
  ≈⟨ ▽-def ⟩
    π ⨾ π ˘ ⊓ ρ ⨾ ρ ˘
  ≈⟨ ⊠-extensionality ⟩
    Id
  □
```

```
module Default-⊗ (_⊠_ : Obj → Obj → Obj)
                 (isDirectProduct : (A B : Obj) → IsDirectProduct A B (A ⊠ B)) where
  module _ {A B : Obj} where open IsDirectProduct (isDirectProduct A B) public
  _⊗_ : {A₁ A₂ B₁ B₂ : Obj} → Mor A₁ B₁ → Mor A₂ B₂ → Mor (A₁ ⊠ A₂) (B₁ ⊠ B₂)
  R ⊗ S = (π ⨾ R) ▽ (ρ ⨾ S)
  ⊗-def : {A₁ A₂ B₁ B₂ : Obj} {R : Mor A₁ B₁} {S : Mor A₂ B₂} → R ⊗ S ≈ (π ⨾ R) ▽ (ρ ⨾ S)
  ⊗-def = ≈-refl
```

4.5 Categoric.Functor.Retract

```
open import RATH.Level
open import Categoric.Category
open import Categoric.Functor
```

```
module Categoric.Functor.Retract {i j k : Level} {Obj : Set i} (𝒞 : Category j k Obj) where
open Category 𝒞
```

```
retractFunctor : {i₂ : Level} {Obj₂ : Set i₂} (F : Obj₂ → Obj)
               → Functor (retractCategory F 𝒞) 𝒞
retractFunctor F = record
  {obj = F
  ; mor = λ f → f
  ; mor-cong = λ f≈g → f≈g
  ; mor-⨾    = ≈-refl
  ; mor-Id   = ≈-refl
  }
```

```
module Retract²Functor
        {i₂ j₂   : Level} {Obj₂ : Set i₂} {Mor₂ : Obj₂ → Obj₂ → Set j₂}
        (Id₂     : {A : Obj₂} → Mor₂ A A)
        (_⨾₂_    : {A B C : Obj₂} → Mor₂ A B → Mor₂ B C → Mor₂ A C)
        (FO      : Obj₂ → Obj)
        (FM      : {A B : Obj₂} → Mor₂ A B → Mor (FO A) (FO B))
        (FM-Id₂  : {A : Obj₂} → FM Id₂ ≈ Id {FO A})
        (FM-⨾₂   : {A B C : Obj₂} {f : Mor₂ A B} {g : Mor₂ B C}
                 → FM (f ⨾₂ g) ≈ FM f ⨾ FM g)
  where
  𝒞₂ = retract²Category 𝒞 Id₂ _⨾₂_ FO FM FM-Id₂ FM-⨾₂
  private
    module 𝒞₂ = Category 𝒞₂
  open Category₂ 𝒞₂ hiding (Id₂; _⨾₂_)
```

```
  open CatF using (ReflectsMonos; ReflectsEpis)
  retract²Functor : Functor 𝒞₂ 𝒞
  retract²Functor = record
    {obj = FO
    ; mor = FM
    ; mor-cong = λ f≈g → f≈g
    ; mor-⨾    = FM-⨾₂
    ; mor-Id   = FM-Id₂
    }
  retract²Functor-reflectsMonos : ReflectsMonos retract²Functor
  retract²Functor-reflectsMonos {A} {B} {F} F-isMono {Z} {R} {S} ℱ-RF≈ℱ-SF
    = F-isMono {FO Z} {FM R} {FM S} (≈-begin
        FM R ⨾ FM F
      ≈˘⟨ FM-⨾₂ ⟩
        FM (R ⨾₂ F)
      ≈⟨ ℱ-RF≈ℱ-SF ⟩
        FM (S ⨾₂ F)
      ≈⟨ FM-⨾₂ ⟩
        FM S ⨾ FM F
      □)
  retract²Functor-reflectsEpis : ReflectsEpis retract²Functor
  retract²Functor-reflectsEpis {A} {B} {F} F-isEpi {Z} {R} {S} ℱ-FR≈ℱ-FS
    = F-isEpi {FO Z} {FM R} {FM S} (≈-begin
        FM F ⨾ FM R
      ≈˘⟨ FM-⨾₂ ⟩
        FM (F ⨾₂ R)
      ≈⟨ ℱ-FR≈ℱ-FS ⟩
        FM (F ⨾₂ S)
      ≈⟨ FM-⨾₂ ⟩
        FM F ⨾ FM S
      □)
open Retract²Functor public hiding (𝒞₂)
```

4.6 Categoric.Product.ConvOp

```
module Categoric.Product.ConvOp where
open import RATH.Level
open import Categoric.Semigroupoid
open import Categoric.ConvSemigroupoid
open import Categoric.Product.Semigroupoid
open import RATH.Data.Product
```

```
module ProdConvOp
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} {𝒮₁ : Semigroupoid {i₁} j₁ k₁ Obj₁} (convOp₁ : ConvOp 𝒮₁)
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} {𝒮₂ : Semigroupoid {i₂} j₂ k₂ Obj₂} (convOp₂ : ConvOp 𝒮₂)
  where
  open Semigroupoid₁ 𝒮₁
  open Semigroupoid₂ 𝒮₂
  private
    𝒮₁×𝒮₂ = ProductSemigroupoid 𝒮₁ 𝒮₂
    Obj = Obj₁ × Obj₂
```

```
      module convOp₁ = ConvOp convOp₁
      module convOp₂ = ConvOp convOp₂
      module 𝒮₁ = Semigroupoid 𝒮₁
      module 𝒮₂ = Semigroupoid 𝒮₂
    open convOp₁ using () renaming (_˘ to _˘₁)
    open convOp₂ using () renaming (_˘ to _˘₂)
    open Semigroupoid 𝒮₁×𝒮₂
    _˘ : {A B : Obj} → Mor A B → Mor B A
    (R , S) ˘ = R ˘₁ , S ˘₂
    ˘-cong : {A B : Obj} {R S : Mor A B} → R ≈ S → R ˘ ≈ S ˘
    ˘-cong (R₁≈S₁ , R₂≈S₂) = convOp₁.˘-cong R₁≈S₁ , convOp₂.˘-cong R₂≈S₂
    ˘˘ : {A B : Obj} {R : Mor A B} → (R ˘) ˘ ≈ R
    ˘˘ = convOp₁.˘˘ , convOp₂.˘˘
    ⨾-˘ : {A B C : Obj} {R : Mor A B} {S : Mor B C} → (R ⨾ S) ˘ ≈ S ˘ ⨾ R ˘
    ⨾-˘ = convOp₁.⨾-˘ , convOp₂.⨾-˘
    ProductConvOp : ConvOp 𝒮₁×𝒮₂
    ProductConvOp = record
      {_˘ = _˘
      ;˘-cong = ˘-cong
      ;˘˘ = ˘˘
      ;⨾-˘ = ⨾-˘
      }

open ProdConvOp public using (ProductConvOp)
```

4.7 Categoric.Product.LocOrd

```
module Categoric.Product.LocOrd where
open import RATH.Level
open import Categoric.LEPGraph
open import Categoric.Semigroupoid
open import Categoric.OrderedSemigroupoid
open import Categoric.Product.Semigroupoid
open import RATH.Data.Product
open import Relation.Binary.Poset.Constructions using (_×-poset_)


module _
    {i₁ j₁ k₁₁ k₂₁ : Level} {Obj₁ : Set i₁} (Hom₁ : LocalEdgePoset Obj₁ j₁ k₁₁ k₂₁)
    {i₂ j₂ k₁₂ k₂₂ : Level} {Obj₂ : Set i₂} (Hom₂ : LocalEdgePoset Obj₂ j₂ k₁₂ k₂₂)
    where
    ProductLEP : LocalEdgePoset (Obj₁ × Obj₂) (j₁ ⊍ j₂) (k₁₁ ⊍ k₁₂) (k₂₁ ⊍ k₂₂)
    ProductLEP (A₁ , A₂) (B₁ , B₂) = Hom₁ A₁ B₁ ×-poset Hom₂ A₂ B₂


module ProdLocOrd
    {i₁ j₁ k₁₁ k₂₁ : Level} {Obj₁ : Set i₁} {Hom₁ : LocalEdgePoset Obj₁ j₁ k₁₁ k₂₁} {compOp₁ : CompOp (LEP≈ Hom₁)} (locOrd₁ : Lo
    {i₂ j₂ k₁₂ k₂₂ : Level} {Obj₂ : Set i₂} {Hom₂ : LocalEdgePoset Obj₂ j₂ k₁₂ k₂₂} {compOp₂ : CompOp (LEP≈ Hom₂)} (locOrd₂ : Lo
    where
    open LocalEdgePoset Hom₁ using () renaming (Hom≈ to Hom₁≈; _⊑_ to _⊑₁_)
    𝒮₁ : Semigroupoid j₁ k₁₁ Obj₁
```

```
  S₁ = record {Hom = Hom₁≈; compOp = compOp₁}
  open Semigroupoid₁ S₁ hiding (Hom₁; compOp₁)
  open LocalEdgePoset Hom₂ using () renaming (Hom≈ to Hom₂≈; _⊑_ to _⊑₂_)
  S₂ : Semigroupoid j₂ k₁₂ Obj₂
  S₂ = record {Hom = Hom₂≈; compOp = compOp₂}
  open Semigroupoid₂ S₂ hiding (Hom₂; compOp₂)
  private
    S₁×S₂ = ProductSemigroupoid S₁ S₂
    Obj = Obj₁ × Obj₂
    module locOrd₁ = LocOrd locOrd₁
    module locOrd₂ = LocOrd locOrd₂
    module S₁ = Semigroupoid S₁
    module S₂ = Semigroupoid S₂
    Hom = ProductLEP Hom₁ Hom₂
  open locOrd₁ using () renaming (_⊑_ to _⊑₁_)
  open locOrd₂ using () renaming (_⊑_ to _⊑₂_)
  open Semigroupoid S₁×S₂ hiding (Hom)
  open LocalEdgePoset Hom
  ⨾-monotone :   {A B C : Obj} {P Q : Mor A B} {R S : Mor B C}
               → P ⊑ Q → R ⊑ S → (P ⨾ R) ⊑ (Q ⨾ S)
  ⨾-monotone (P₁⊑Q₁ , P₂⊑Q₂) (R₁⊑S₁ , R₂⊑S₂) = locOrd₁.⨾-monotone P₁⊑Q₁ R₁⊑S₁ , locOrd₂.⨾-monotone P₂⊑Q₂ R₂⊑S₂
  ProductLocOrd : LocOrd Hom compOp
  ProductLocOrd = record {⨾-monotone = ⨾-monotone}

open ProdLocOrd public using (ProductLocOrd)
```

4.8 Categoric.Product.OrderedSemigroupoid

```
module Categoric.Product.OrderedSemigroupoid where
open import RATH.Level
open import Categoric.Semigroupoid
open import Categoric.OrderedSemigroupoid
open import Categoric.Product.Semigroupoid
open import Categoric.Product.LocOrd
open import RATH.Data.Product

module ProdOSG
  {i₁ j₁ k₁₁ k₂₁ : Level} {Obj₁ : Set i₁} (C₁ : OrderedSemigroupoid {i₁} j₁ k₁₁ k₂₁ Obj₁)
  {i₂ j₂ k₁₂ k₂₂ : Level} {Obj₂ : Set i₂} (C₂ : OrderedSemigroupoid {i₂} j₂ k₁₂ k₂₂ Obj₂)
  where
  private
    module C₁ = OrderedSemigroupoid C₁
    module C₂ = OrderedSemigroupoid C₂
    C₁×C₂ = ProductSemigroupoid C₁.semigroupoid C₂.semigroupoid
  ProductOSG : OrderedSemigroupoid (j₁ ⊍ j₂) (k₁₁ ⊍ k₁₂) (k₂₁ ⊍ k₂₂) (Obj₁ × Obj₂)
  ProductOSG = record
    {Hom = ProductLEP C₁.Hom C₂.Hom
    ; compOp = Semigroupoid.compOp C₁×C₂
    ; locOrd = ProductLocOrd C₁.locOrd C₂.locOrd
    }

open ProdOSG public
```

4.9 Categoric.Product.OCC

```
module Categoric.Product.OCC where
open import RATH.Level
open import Categoric.Semigroupoid
open import Categoric.ConvSemigroupoid
open import Categoric.OrderedSemigroupoid
open import Categoric.OrderedSemigroupoid.Lattice using (module LocOrdJoin)
open import Categoric.Category
open import Categoric.OCC
open import Categoric.Relator.OCC
open import Categoric.Category.FinColimits
open import Categoric.Product.Semigroupoid
open import Categoric.Product.Category using (ProductCategory)
open import Categoric.Product.ConvOp
open import Categoric.Product.LocOrd
open import RATH.Data.Product
```

```
module ProdOCC
  {i₁ j₁ k₁₁ k₂₁ : Level} {Obj₁ : Set i₁} (𝒞₁ : OCC {i₁} j₁ k₁₁ k₂₁ Obj₁)
  {i₂ j₂ k₁₂ k₂₂ : Level} {Obj₂ : Set i₂} (𝒞₂ : OCC {i₂} j₂ k₁₂ k₂₂ Obj₂)
  where
  private
    module 𝒞₁ = OCC 𝒞₁
    module 𝒞₂ = OCC 𝒞₂
  open Category₁ 𝒞₁.category
  open Category₂ 𝒞₂.category
  private
    𝒞₁×𝒞₂ = ProductCategory 𝒞₁.category 𝒞₂.category
    Obj = Obj₁ × Obj₂
    locOrd = ProductLocOrd 𝒞₁.locOrd 𝒞₂.locOrd
    convOp = ProductConvOp 𝒞₁.convOp 𝒞₂.convOp
  open Category 𝒞₁×𝒞₂
  open LocOrd locOrd
  open ConvOp convOp
  ˘-monotone : {A B : Obj} {R S : Mor A B} → R ⊑ S → (R ˘) ⊑ (S ˘)
  ˘-monotone (R₁⊑S₁ , R₂⊑S₂) = 𝒞₁.˘-monotone R₁⊑S₁ , 𝒞₂.˘-monotone R₂⊑S₂
  ProductOCC : OCC (j₁ ⊍ j₂) (k₁₁ ⊍ k₁₂) (k₂₁ ⊍ k₂₂) Obj
  ProductOCC = record {OCC_Base = record
    {osgc = record {OSGC_Base = record
      {orderedSemigroupoid = record
        {Hom = ProductLEP 𝒞₁.Hom 𝒞₂.Hom
        ; compOp = compOp
        ; locOrd = locOrd
        }
      ; convOp = convOp
      ; ˘-monotone = ˘-monotone
      }}
    ; idOp = idOp
    }}
```

```
Proj₁ : Relator ProductOCC 𝒞₁
Proj₁ = record
```

```
  {obj = proj₁
  ; mor = proj₁
  ; monotone = proj₁
  ; mor-⨾ = ≈₁-refl
  ; mor-Id = ≈₁-refl
  ; mor-˘ = ≈₁-refl
  }

Proj₂ : Relator ProductOCC 𝒞₂
Proj₂ = record
  {obj = proj₂
  ; mor = proj₂
  ; monotone = proj₂
  ; mor-⨾ = ≈₂-refl
  ; mor-Id = ≈₂-refl
  ; mor-˘ = ≈₂-refl
  }

open ProdOCC public

module _
  {i₁ j₁ k₁₁ k₂₁ : Level} {Obj₁ : Set i₁} {𝒞₁ : OCC j₁ k₁₁ k₂₁ Obj₁}
  {i₂ j₂ k₁₂ k₂₂ : Level} {Obj₂ : Set i₂} {𝒞₂ : OCC j₂ k₁₂ k₂₂ Obj₂}
    where
  private
    module 𝒞₁ = OCC 𝒞₁
    module 𝒞₂ = OCC 𝒞₂
  ProductUnit : 𝒞₁.HasUnit → 𝒞₂.HasUnit → OCC.HasUnit (ProductOCC 𝒞₁ 𝒞₂)
  ProductUnit hasUnit₁ hasUnit₂ = record
    {𝟙 = U₁.𝟙 , U₂.𝟙
    ; 𝟙-isUnit = record
      {Id-isTop = U₁.Id-isTop , U₂.Id-isTop
      ; toUnit = U₁.toUnit , U₂.toUnit
      ; toUnit-isTotal = U₁.toUnit-isTotal , U₂.toUnit-isTotal
      }
    }
    where
      module U₁ = 𝒞₁.HasUnit hasUnit₁
      module U₂ = 𝒞₂.HasUnit hasUnit₂

module _
  {i₁ j₁ k₁₁ k₂₁ : Level} {Obj₁ : Set i₁} {𝒞₁ : OCC j₁ k₁₁ k₂₁ Obj₁}
  {i₂ j₂ k₁₂ k₂₂ : Level} {Obj₂ : Set i₂} {𝒞₂ : OCC j₂ k₁₂ k₂₂ Obj₂}
  {i₃ j₃ k₁₃ k₂₃ : Level} {Obj₃ : Set i₃} {𝒞₃ : OCC j₃ k₁₃ k₂₃ Obj₃}
    where
  private
    module 𝒞₁ = OCC 𝒞₁
    module 𝒞₂ = OCC 𝒞₂
    module 𝒞₃ = OCC 𝒞₃
  open Category₁ 𝒞₁.category
  open Category₂ 𝒞₂.category
  open Category₃ 𝒞₃.category
  infixr 4 _▼_
  _▼_ : Relator 𝒞₃ 𝒞₁ → Relator 𝒞₃ 𝒞₂ → Relator 𝒞₃ (ProductOCC 𝒞₁ 𝒞₂)
```

```
F ▼ G = record
  {obj = λ A → F.obj A , G.obj A
  ; mor = λ f → F.mor f , G.mor f
  ; monotone = λ f⊑g → F.monotone f⊑g , G.monotone f⊑g
  ; mor-⨾ = F.mor-⨾ , G.mor-⨾
  ; mor-Id = F.mor-Id , G.mor-Id
  ; mor-˘ = F.mor-˘ , G.mor-˘
  }
  where
    module F = Relator F
    module G = Relator G


module _ (F : Relator (ProductOCC 𝒞₁ 𝒞₂) 𝒞₃) where
  private module F = Relator F
  infixl 9 _at₁_
  _at₁_ : Obj₁ → Relator 𝒞₂ 𝒞₃
  _at₁_ A = record
    {obj = λ B → F.obj (A , B)
    ; mor = λ f → F.mor (𝒞₁.Id , f)
    ; monotone = λ f⊑g → F.monotone (𝒞₁.⊑-refl , f⊑g)
    ; mor-⨾ = F.mor-cong (𝒞₁.leftId , 𝒞₂.≈-refl) 𝒞₃.⟨≈˘≈⟩ F.mor-⨾
    ; mor-Id = F.mor-Id
    ; mor-˘ = F.mor-cong (𝒞₁.Id˘ , 𝒞₂.≈-refl) 𝒞₃.⟨≈˘≈⟩ F.mor-˘
    }


  infixl 9 _at₂_
  _at₂_ : Obj₂ → Relator 𝒞₁ 𝒞₃
  _at₂_ B = record
    {obj = λ A → F.obj (A , B)
    ; mor = λ f → F.mor (f , 𝒞₂.Id)
    ; monotone = λ f⊑g → F.monotone (f⊑g , 𝒞₂.⊑-refl)
    ; mor-⨾ = F.mor-cong (𝒞₁.≈-refl , 𝒞₂.leftId) 𝒞₃.⟨≈˘≈⟩ F.mor-⨾
    ; mor-Id = F.mor-Id
    ; mor-˘ = F.mor-cong (𝒞₁.≈-refl , 𝒞₂.Id˘) 𝒞₃.⟨≈˘≈⟩ F.mor-˘
    }


  birelator-Id-commute : {A₁ B₁ : Obj₁} {A₂ B₂ : Obj₂} {f₁ : Mor₁ A₁ B₁} {f₂ : Mor₂ A₂ B₂}
                       → F.mor (f₁ , Id₂) ⨾₃ F.mor (Id₁ , f₂)
                       ≈₃ F.mor (Id₁ , f₂) ⨾₃ F.mor (f₁ , Id₂)
  birelator-Id-commute {A₁} {B₁} {A₂} {B₂} {f₁} {f₂} = ≈₃-begin
      F.mor (f₁ , Id₂) ⨾₃ F.mor (Id₁ , f₂)
    ≈₃˘⟨ F.mor-⨾ ⟩
      F.mor (f₁ ⨾₁ Id₁ , Id₂ ⨾₂ f₂)
    ≈₃⟨ F.mor-cong ((rightId₁ ⟨≈₁≈˘⟩ leftId₁) , (leftId₂ ⟨≈₂≈˘⟩ rightId₂)) ⟩
      F.mor (Id₁ ⨾₁ f₁ , f₂ ⨾₂ Id₂)
    ≈₃⟨ F.mor-⨾ ⟩
      F.mor (Id₁ , f₂) ⨾₃ F.mor (f₁ , Id₂)
    □₃


module _ {i₄ j₄ k₁₄ k₂₄ : Level} {Obj₄ : Set i₄} {𝒞₄ : OCC j₄ k₁₄ k₂₄ Obj₄}
         (F : Relator 𝒞₁ 𝒞₃) (G : Relator 𝒞₂ 𝒞₄)
    where
  private
    module F = Relator F
```

```
    module G = Relator G
  ProductRelator : Relator (ProductOCC 𝒞₁ 𝒞₂) (ProductOCC 𝒞₃ 𝒞₄)
  ProductRelator = record
    {obj = λ {(A₁ , A₂) → F.obj A₁ , G.obj A₂}
    ; mor = λ {(f₁ , f₂) → F.mor f₁ , G.mor f₂}
    ; monotone = λ {(f₁⊑g₁ , f₂⊑g₂) → F.monotone f₁⊑g₁ , G.monotone f₂⊑g₂}
    ; mor-⨾ = F.mor-⨾ , G.mor-⨾
    ; mor-Id = F.mor-Id , G.mor-Id
    ; mor-˘ = F.mor-˘ , G.mor-˘
    }
```

4.10 Categoric.Product.MeetOp

```
module Categoric.Product.MeetOp where
open import RATH.Level
open import Categoric.Semigroupoid
open import Categoric.OrderedSemigroupoid
open import Categoric.OrderedSemigroupoid.Lattice
open import Categoric.LSLSemigroupoid
open import Categoric.Product.LocOrd
open import Categoric.Product.OrderedSemigroupoid
open import RATH.Data.Product
```

```
module ProdMeetOp
  {i₁ j₁ k₁₁ k₂₁ : Level} {Obj₁ : Set i₁} {𝒮₁ : OrderedSemigroupoid {i₁} j₁ k₁₁ k₂₁ Obj₁} (meetOp₁ : MeetOp 𝒮₁)
  {i₂ j₂ k₁₂ k₂₂ : Level} {Obj₂ : Set i₂} {𝒮₂ : OrderedSemigroupoid {i₂} j₂ k₁₂ k₂₂ Obj₂} (meetOp₂ : MeetOp 𝒮₂)
  where
  private
    module 𝒮₁ = OrderedSemigroupoid 𝒮₁
    module 𝒮₂ = OrderedSemigroupoid 𝒮₂
    𝒮₁×𝒮₂ = ProductOSG 𝒮₁ 𝒮₂
    Obj = Obj₁ × Obj₂
    locOrd = ProductLocOrd 𝒮₁.locOrd 𝒮₂.locOrd
    module meetOp₁ = MeetOp meetOp₁
    module meetOp₂ = MeetOp meetOp₂
  open Semigroupoid₁ 𝒮₁.semigroupoid
  open Semigroupoid₂ 𝒮₂.semigroupoid
  open OrderedSemigroupoid 𝒮₁×𝒮₂
  open LocOrdMeet Hom
  open meetOp₁ using () renaming (_⊓_ to _⊓₁_)
  open meetOp₂ using () renaming (_⊓_ to _⊓₂_)
  _⊓_ : {A B : Obj} → Mor A B → Mor A B → Mor A B
  (R₁ , R₂) ⊓ (S₁ , S₂) = R₁ ⊓₁ S₁ , R₂ ⊓₂ S₂
  ⊓-lower₁ : {A B : Obj} {R S : Mor A B} → R ⊓ S ⊑ R
  ⊓-lower₁ = meetOp₁.⊓-lower₁ , meetOp₂.⊓-lower₁
  ⊓-lower₂ : {A B : Obj} {R S : Mor A B} → R ⊓ S ⊑ S
  ⊓-lower₂ = meetOp₁.⊓-lower₂ , meetOp₂.⊓-lower₂
  ⊓-universal : {A B : Obj} {R S X : Mor A B} → X ⊑ R → X ⊑ S → X ⊑ R ⊓ S
  ⊓-universal (X₁⊑R₁ , X₂⊑R₂) (X₁⊑S₁ , X₂⊑S₂) = meetOp₁.⊓-universal X₁⊑R₁ X₁⊑S₁ , meetOp₂.⊓-universal X₂⊑R₂ X₂⊑S₂
  meet : {A B : Obj} → (R S : Mor A B) → Meet R S
  meet R S = record {value = R ⊓ S; proof = record {bound₁ = ⊓-lower₁; bound₂ = ⊓-lower₂; universal = ⊓-universal}}
```

```
  ProductMeetOp : MeetOp 𝒮₁×𝒮₂
  ProductMeetOp = record {meet = meet}

open ProdMeetOp public using (ProductMeetOp)
```

4.11 Categoric.Product.Allegory

```
module Categoric.Product.Allegory where
open import RATH.Level
open import Categoric.Category
open import Categoric.OCC
open import Categoric.LSLSemigroupoid
open import Categoric.Allegory
open import Categoric.Relator.OCC
open import Categoric.Product.OCC
open import Categoric.Product.MeetOp
open import RATH.Data.Product

module ProdAllegory
  {i₁ j₁ k₁₁ k₂₁ : Level} {Obj₁ : Set i₁} (𝒜₁ : Allegory {i₁} j₁ k₁₁ k₂₁ Obj₁)
  {i₂ j₂ k₁₂ k₂₂ : Level} {Obj₂ : Set i₂} (𝒜₂ : Allegory {i₂} j₂ k₁₂ k₂₂ Obj₂)
  where
  private
    module 𝒜₁ = Allegory 𝒜₁
    module 𝒜₂ = Allegory 𝒜₂
  open Category₁ 𝒜₁.category
  open Category₂ 𝒜₂.category
  private
    𝒜₁×𝒜₂ = ProductOCC 𝒜₁.occ 𝒜₂.occ
    Obj = Obj₁ × Obj₂
    meetOp = ProductMeetOp 𝒜₁.meetOp 𝒜₂.meetOp
  open OCC 𝒜₁×𝒜₂
  open MeetOp meetOp
  Dedekind : {A B C : Obj} {Q : Mor A B} {R : Mor B C} {S : Mor A C}
           → (Q ⨾ R ⊓ S) ⊑ (Q ⊓ S ⨾ R˘) ⨾ (R ⊓ Q˘ ⨾ S)
  Dedekind = 𝒜₁.Dedekind , 𝒜₂.Dedekind
  ProductAllegory : Allegory (j₁ ⊍ j₂) (k₁₁ ⊍ k₁₂) (k₂₁ ⊍ k₂₂) Obj
  ProductAllegory = record {occ = 𝒜₁×𝒜₂; meetOp = meetOp; Dedekind = Dedekind}

open import Categoric.Relator.OCC
open import Categoric.Relator.Allegory
module _ where
  private
    ℛ : Relator (Allegory.occ ProductAllegory) 𝒜₁.occ
    ℛ = Proj₁ 𝒜₁.occ 𝒜₂.occ
    module ℛ = Relator ℛ
  Proj₁-mor-⊓-⊒ : MeetPres.PreservesMeets-⊒ ProductAllegory 𝒜₁ ℛ
  Proj₁-mor-⊓-⊒ {A} {B} {R} {S} = 𝒜₁.⊑-refl
module _ where
  private
```

```
    𝓡 : Relator (Allegory.occ ProductAllegory) 𝒜₂.occ
    𝓡 = Proj₂ 𝒜₁.occ 𝒜₂.occ
    module 𝓡 = Relator 𝓡

  Proj₂-mor-⊓-⊒ : MeetPres.PreservesMeets-⊒ ProductAllegory 𝒜₂ 𝓡
  Proj₂-mor-⊓-⊒ {A} {B} {R} {S} = 𝒜₂.⊑-refl
module _
  {i₃ j₃ k₁₃ k₂₃ : Level} {Obj₃ : Set i₃} (𝒜₃ : Allegory {i₃} j₃ k₁₃ k₂₃ Obj₃)
  where

  private
    module 𝒜₃ = Allegory 𝒜₃
  open Category₃ 𝒜₃.category

  module _ (𝓡₁ : Relator 𝒜₃.occ 𝒜₁.occ)
           (𝓡₂ : Relator 𝒜₃.occ 𝒜₂.occ)
           (𝓡₁-mor-⊓-⊒ : MeetPres.PreservesMeets-⊒ 𝒜₃ 𝒜₁ 𝓡₁)
           (𝓡₂-mor-⊓-⊒ : MeetPres.PreservesMeets-⊒ 𝒜₃ 𝒜₂ 𝓡₂)
           where
    private
      𝓡 : Relator 𝒜₃.occ (ProductOCC 𝒜₁.occ 𝒜₂.occ)
      𝓡 = 𝓡₁ ▼ 𝓡₂
      module 𝓡 = Relator 𝓡

    ▼-mor-⊓-⊒ : MeetPres.PreservesMeets-⊒ 𝒜₃ ProductAllegory 𝓡
    ▼-mor-⊓-⊒ {A} {B} {R} {S} = 𝓡₁-mor-⊓-⊒ , 𝓡₂-mor-⊓-⊒
  module _ (𝓡₁ : Relator 𝒜₁.occ 𝒜₂.occ)
           (𝓡₂ : Relator 𝒜₂.occ 𝒜₃.occ)
           (𝓡₁-mor-⊓-⊒ : MeetPres.PreservesMeets-⊒ 𝒜₁ 𝒜₂ 𝓡₁)
           (𝓡₂-mor-⊓-⊒ : MeetPres.PreservesMeets-⊒ 𝒜₂ 𝒜₃ 𝓡₂)
           where
    private
      𝓡 : Relator 𝒜₁.occ 𝒜₃.occ
      𝓡 = 𝓡₁ ⨾ 𝓡₂
      module 𝓡 = Relator 𝓡
      module 𝓡₁ = Relator 𝓡₁
      module 𝓡₂ = Relator 𝓡₂

    ⨾-mor-⊓-⊒ : MeetPres.PreservesMeets-⊒ 𝒜₁ 𝒜₃ 𝓡
    ⨾-mor-⊓-⊒ {A} {B} {R} {S} = 𝓡₂-mor-⊓-⊒ 𝒜₃.⟨⊑⊑⟩ 𝓡₂.monotone 𝓡₁-mor-⊓-⊒

open ProdAllegory public
```

Bibliography

Richard S. Bird and Oege de Moor. *Algebra of Programming*, volume 100 of *International Series in Computer Science*. Prentice Hall, 1997.

Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation, part I: Basic concepts and double pushout approach. In Grzegorz Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*, chapter 3, pages 163–245. World Scientific, Singapore, 1997. ISBN 9810228848.

Nils Anders Danielsson et al. Agda standard library, version 0.7, January 2013. http://tinyurl.com/AgdaStdlib.

Wolfram Kahl. Dependently-typed formalisation of relation-algebraic abstractions. In Harrie de Swart, editor, *Relational and Algebraic Methods in Computer Science, RAMiCS 2011*, volume 6663 of *LNCS*, pages 230–247. Springer, 2011. doi: 10.1007/978-3-642-21070-9_18.

Wolfram Kahl. Towards certifiable implementation of graph transformation via relation categories. In Wolfram Kahl and Timothy G. Griffin, editors, *Relational and Algebraic Methods in Computer Science, RAMiCS 2012*, volume 7560 of *LNCS*, pages 82–97. Springer, 2012. ISBN 978-3-642-33314-9. doi: 10.1007/978-3-642-33314-9_6.

Wolfram Kahl. Relation-Algebraic Theories in Agda — RATH-Agda-2.0.1. Mechanically checked Agda theories, with 456 pages literate document output. http://relmics.mcmaster.ca/RATH-Agda/, February 2014.

Yasuo Kawahara. Notes on the universality of relational functors. *Mem. Fac. Sci. Kyushu Univ. Ser. A*, 27(2): 275–289, 1973.

Michael Löwe. Algebraic approach to graph transformation based on single pushout derivations. Technical Report 90/05, TU Berlin, 1990.

Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007. See also `http://wiki.portal.chalmers.se/agda/pmwiki.php`.