Categories of Coalgebras with Semi-Monadic Homomorphisms

MonCoAlg-0.1

Wolfram Kahl

kahl@cas.mcmaster.ca

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

1 February 2014

Abstract

We present the Agda theories resulting from a first exploration of coalgebras where the functor factors over a monad, and coalgebra homomorphisms are Kleisli homomorphisms satisfying an appropriate commutativity condition.

The resulting categories do not coincide with coalgebra categories over the Kleisli category, but encompass many common graph structures that are usually treated with ad-hoc extensions of the algebraic approach to graph transformation.

As examples for our coalgebraic approach, we show symbolically edge-attributed graphs, for which pushouts in the underlying Kleisli category extend to pushouts in the coalgebra category, and a simple formalisation of term graphs.

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).

Overview

Chapter 2 contains the basic definitions of monads and their Kleisli categories, and product monads.

Conventional coalgebras are defined in Chapter 3, and we explore moving between coalgebras over the Kleisli category of a monad and coalgebras over the underlying category.

In Chapter 4 we present our version of "monadic coalgebras", the operation of which is in general *not* a Kleisli arrow. We instantiate this monadic coalgebra concept for symbolically edge-attributed graphs in Chapter 5, and for simple term graphs in Chapter 6.

Chapter 7 presents a specialisation of the monadic coalgebras of Chapter 4 to a product category, in a way that is a generalisation of both the symbolically edge-attributed graphs of Chapter 5 and the simple term graphs of Chapter 6.

Since for term graphs, it is easy to see that a pushout in the Kleisli category does not in general extend to a pushout of term graphs, we present a further specialisation of the product category setting in Chapter 8, where we restrict the monad to not depend on "the other component"; this is still a generalisation of the symbolically edge-attributed graphs of Chapter 5, and does have the property that pushouts in the Kleisli category of the underlying monad (from the point of view of Chapter 4) extend to a pushout in the coalgebra category.

Each section of these chapters is the document view of a literate Agda module file, processed by `lhs2TeX -agda`.

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

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

1.1 MonCoAlg0All

module MonCoAlg0All **where**

This module contains entry points from which all modules that are part of the package MonCoAlg are reached. The

commented-out modules contain holes that cannot be filled, and are included in the project for documentation purposes.

```
import Categoric.Coalgebra.Category
import Categoric.Coalgebra.Monadic
  -- import Categoric.Coalgebra.Monadic.GfromF
  -- import Categoric.Coalgebra.Monadic.GfromFM
import Categoric.Monad.DepProduct
import Categoric.Monad.KleisliEndoFunctor
import Categoric.MonCoAlg.SEAG.Cat2.Equiv
  -- import Categoric.MonCoAlg.SEAG.FMNonProps
import Categoric.MonCoAlg.SEAG.Cat2.Pushout3
import Categoric.MonCoAlg.TG.Cat2.Equiv
import Categoric.MonCoAlg.P.Equiv
import Categoric.MonCoAlg.P.Pushout2
```

The following refers to a 2011 machine running Linux on a six-core 3.2GHZ AMD Phenom II with 16GB of RAM. On this machine, I have been able to type-check the current module from scratch (with the standard library already type-checked) in about 40 minutes using the following incantation (with the RATH-Agda (Kahl, 2014) sources installed in the same directory):

```
STDLIB=/usr/local/packages/Agda-2.3.3.9/build/lib/src  # adapt to your installation!

agda +RTS -K128M -S -M13G -H13G -RTS -i . -i -i $STDLIB  MonCoAlg0.lagda
```

For checking this module as a whole, and probably also for some of the indirectly imported modules, much less than 9GB of heap will not work.

Chapter 2

Monads

2.1 Categoric.Monad

```
module Categoric.Monad where

open import RATH.Level
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
open import Categoric.Semigroupoid
import Categoric.Semigroupoid.FinColimits as SGFinColimits
open import Categoric.Category
open import Categoric.Category.FinColimits
open import Categoric.Functor
open import Categoric.IdOp
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
```

Since we want the base category $\mathcal{C}$ to be an explicit argument of **record** Monad, but an implicit argument of **module** Monad, we cannot make it a parameter of the current module Categoric.Monad.

```
module Functor-M {i j k : Level} {Obj : Set i} {𝒞 : Category j k Obj} (M : Functor 𝒞 𝒞) where
  open Functor M public using () renaming
    (obj to M-obj
    ; mor to M-mor
    ; mor-cong to M-cong
    ; mor-⨟ to M-mor-⨟
    ; mor-Id to M-Id
    )
```

The type MonadTrafos M collects the natural transformations, together with their properties, that turn functor M into a monad.

contains following record

```
record MonadTrafos {i j k : Level} {Obj : Set i} {𝒞 : Category j k Obj} (M : Functor 𝒞 𝒞) : Set (i ⊔ j ⊔ k) where
  field
    M-return : NatTrans {C₂ = 𝒞} (Identity _) M
    M-join   : NatTrans {C₂ = 𝒞} (M ⨟⨟ M) M
  open Category 𝒞
  open Functor-M M
  module M-return = NatTrans M-return
  module M-join   = NatTrans M-join
```

For the sake of explicitly stating the types and naturality properties, we restate them instead of just extracting them from the modules via **renaming** — this also has the advantage that we do not have to duplicate those **open** directives with **public** later:

```
  return     : {A : Obj} → Mor A (M-obj A)
  return     = M-return.indmor
  join       : {A : Obj} → Mor (M-obj (M-obj A)) (M-obj A)
  join       = M-join.indmor
  return-naturality : {A B : Obj} {f : Mor A B} → f ⨟ return ≈ return ⨟ M-mor f
  return-naturality = M-return.naturality
  join-naturality   : {A B : Obj} {f : Mor A B} → M-mor (M-mor f) ⨟ join ≈ join ⨟ M-mor f
  join-naturality   = M-join.naturality
```

Now the monad laws:

```
  field
    leftUnit  : {A : Obj} → return {M-obj A} ⨟ join {A} ≈ Id {M-obj A}
    rightUnit : {A : Obj} → M-mor (return {A}) ⨟ join {A} ≈ Id {M-obj A}
    assoc     : {A : Obj} → join {M-obj A} ⨟ join {A} ≈ M-mor (join {A}) ⨟ join {A}
```

Kleisli morphisms:

```
  KHom : LocalSetoid Obj j k
  KHom A B = Hom A (M-obj B)
```

After the last **field**, we can now **open public**:

```
  open LocalEdgeSetoid KHom public using () renaming (Edge to KMor)
```

Lifting of Kleisli morphisms:

```
  ↼ : {A B : Obj} → KMor A B → KMor (M-obj A) B
  ↼ f = M-mor f ⨟ join


  ↼-cong : {A B : Obj} {f g : KMor A B} → f ≈ g → ↼ f ≈ ↼ g
  ↼-cong f≈g = ⨟-cong₁ (M-cong f≈g)


  ↼⨟ : {A B C : Obj} {f : Mor A B} {g : KMor B C} → ↼ (f ⨟ g) ≈ M-mor f ⨟ ↼ g
  ↼⨟ {f = f} {g} = ≈-begin
      ↼ (f ⨟ g)
    ≈⟨ ≈-refl ⟩
      M-mor (f ⨟ g) ⨟ join
    ≈⟨ ⨟-cong₁ M-mor-⨟ ⟨≈≈⟩ ⨟-assoc ⟩
      M-mor f ⨟ M-mor g ⨟ join
    ≈⟨ ≈-refl ⟩
      M-mor f ⨟ ↼ g
    □


  ↼-⨟return : {A B : Obj} {f : Mor A B} → ↼ (f ⨟ return) ≈ M-mor f
  ↼-⨟return {f = f} = ≈-begin
      ↼ (f ⨟ return)
    ≈⟨ ↼⨟ ⟩
      M-mor f ⨟ M-mor return ⨟ join
    ≈⟨ ⨟-cong₂ rightUnit ⟩
      M-mor f ⨟ Id
    ≈⟨ rightId ⟩
      M-mor f
    □


  ↼-⨟M-mor  : {A B C : Obj} {f : KMor A B} {g : Mor B C} → ↼ (f ⨟ M-mor g) ≈ ↼ f ⨟ M-mor g
  ↼-⨟M-mor {f = f} {g} = ≈-begin
```

```
      ↼ (f ⨟ M-mor g)
    ≈⟨ ↼⨟ ⟩
      M-mor f ⨟ M-mor (M-mor g) ⨟ join
    ≈⟨ ⨟-cong₂ join-naturality ⟩
      M-mor f ⨟ join ⨟ M-mor g
    ≈⟨ ⨟-assocL ⟩
      ↼ f ⨟ M-mor g
    □

  return-⨟-↼ : {A B : Obj} {f : KMor A B} → return ⨟ ↼ f ≈ f
  return-⨟-↼ {f = f} = ≈-begin
      return ⨟ ↼ f
    ≈⟨ ≈-refl ⟩
      return ⨟ M-mor f ⨟ join
    ≈˘⟨ on-⨟-assocL (⨟-cong₁ return-naturality) ⟩
      f ⨟ return ⨟ join
    ≈⟨ ⨟-cong₂ leftUnit ⟨≈≈⟩ rightId ⟩
      f
    □

  ↼-⨟-↼ : {A B C : Obj} {f : KMor A B} {g : KMor B C} → ↼ f ⨟ ↼ g ≈ ↼ (f ⨟ ↼ g)
  ↼-⨟-↼ {f = f} {g} = ≈-begin
      ↼ f ⨟ ↼ g
    ≈⟨ ⨟-assoc ⟨≈≈˘⟩ ⨟-cong₂ (on-⨟-assocL (⨟-cong₁ join-naturality)) ⟩
      M-mor f ⨟ M-mor (M-mor g) ⨟ join ⨟ join
    ≈⟨ ⨟-assocL ⟨≈≈˘⟩ ⨟-cong₁ M-mor-⨟ ⟩
      M-mor (f ⨟ M-mor g) ⨟ join ⨟ join
    ≈⟨ ⨟-cong₂ assoc ⟨≈≈⟩ ⨟-assocL ⟨≈≈⟩ ⨟-cong₁ (M-mor-⨟ ⟨≈˘≈⟩ M-cong ⨟-assoc) ⟩
      M-mor (f ⨟ M-mor g ⨟ join) ⨟ join
    ≈⟨ ≈-refl ⟩
      ↼ (f ⨟ ↼ g)
    □

  ↼return : {A : Obj} → ↼ (return {A}) ≈ Id
  ↼return = ≈-begin
      ↼ return
    ≈⟨ ≈-refl ⟩
      M-mor return ⨟ join
    ≈⟨ rightUnit ⟩
      Id
    □

  isMono-⇐M : {A B : Obj} {f : Mor A B} → isMono (M-mor f) → isMono (return {A}) → isMono f
  isMono-⇐M {A} {B} {f} Mf-isMono return-isMono {Z} {g} {h} g⨟f≈h⨟f
    = return-isMono {Z} {g} {h}
      (Mf-isMono {Z} {g ⨟ return} {h ⨟ return}
        (≈-begin
            (g ⨟ return) ⨟ M-mor f
          ≈⟨ ⨟-assoc ⟨≈≈˘⟩ ⨟-cong₂ return-naturality ⟩
            g ⨟ f ⨟ return
          ≈⟨ ⨟-cong₁&₂₁ g⨟f≈h⨟f ⟩
            h ⨟ f ⨟ return
          ≈⟨ ⨟-cong₂ return-naturality ⟨≈≈⟩ ⨟-assocL ⟩
            (h ⨟ return) ⨟ M-mor f
          □))

record Monad {i j k : Level} {Obj : Set i} (𝒞 : Category j k Obj) : Set (i ⊍ j ⊍ k) where
  field M : Functor 𝒞 𝒞
```

```
      trafos : MonadTrafos M
  open MonadTrafos trafos public


module Monad-M {i j k : Level} {Obj : Set i} {𝒞 : Category j k Obj} (ℳ : Monad 𝒞) where
  open Monad ℳ public
  open Functor-M M public


module Kleisli {i j k : Level} {Obj : Set i} {𝒞 : Category j k Obj} (ℳ : Monad 𝒞) where
  open Category 𝒞
  open Monad-M ℳ

  infixr 9 _⨟∘_

  _⨟∘_ : {A B C : Obj} → KMor A B → KMor B C → KMor A C
  f ⨟∘ g = f ⨟ η ∙ g

  ⨟∘-cong      : {A B C : Obj} {f₁ f₂ : KMor A B} {g₁ g₂ : KMor B C}
                 → f₁ ≈ f₂ → g₁ ≈ g₂ → f₁ ⨟∘ g₁ ≈ f₂ ⨟∘ g₂
  ⨟∘-cong f₁≈f₂ g₁≈g₂ = ⨟-cong f₁≈f₂ (⨟-cong₁ (M-cong g₁≈g₂))

  ⨟∘-cong₁ : {A B C : Obj} {f₁ f₂ : KMor A B} {g : KMor B C}
             → f₁ ≈ f₂ → f₁ ⨟∘ g ≈ f₂ ⨟∘ g
  ⨟∘-cong₁ = ⨟-cong₁

  ⨟∘-cong₂ : {A B C : Obj} {f : KMor A B} {g₁ g₂ : KMor B C}
             → g₁ ≈ g₂ → f ⨟∘ g₁ ≈ f ⨟∘ g₂
  ⨟∘-cong₂ g₁≈g₂ = ⨟-cong₂₁ (M-cong g₁≈g₂)

  ⨟∘-⨟-assoc       : {A B C D : Obj} {f : KMor A B} {g : KMor B C} {h : Mor C D}
                     → (f ⨟∘ g) ⨟ M-mor h ≈ f ⨟∘ (g ⨟ M-mor h)
  ⨟∘-⨟-assoc {A} {B} {C} {D} {f} {g} {h} = ≈-begin
                (f ⨟∘ g) ⨟ M-mor h
        ≈⟨ ≈-refl ⟩
                (f ⨟ M-mor g ⨟ join) ⨟ M-mor h
        ≈⟨ ⨟-assoc ⟨≈≈⟩ ⨟-cong₂ (⨟-assoc ⟨≈≈˘⟩ ⨟-cong₂ join-naturality) ⟩
                f ⨟ M-mor g ⨟ M-mor (M-mor h) ⨟ join
        ≈⟨ ⨟-cong₂ (⨟-assocL ⟨≈≈˘⟩ ⨟-cong₁ M-mor-⨟) ⟩
                f ⨟ M-mor (g ⨟ M-mor h) ⨟ join
        ≈⟨ ≈-refl ⟩
                f ⨟∘ (g ⨟ M-mor h)
        □

  ⨟∘-⨟-assocL : {A B C D : Obj} {f : KMor A B} {g : KMor B C} {h : Mor C D} → f ⨟∘ (g ⨟ M-mor h) ≈ (f ⨟∘ g) ⨟ M-mor h
  ⨟∘-⨟-assocL = ≈-sym ⨟∘-⨟-assoc

  ⨟-⨟∘-assoc       : {A B C D : Obj} {f : Mor A B} {g : KMor B C} {h : KMor C D}
                     → (f ⨟ g) ⨟∘ h ≈ f ⨟ (g ⨟∘ h)
  ⨟-⨟∘-assoc {A} {B} {C} {D} {f} {g} {h} = ≈-begin
                (f ⨟ g) ⨟∘ h
        ≈⟨ ≈-refl ⟩
                (f ⨟ g) ⨟ M-mor h ⨟ join
        ≈⟨ ⨟-assoc ⟩
                f ⨟ (g ⨟∘ h)
        □

  ⨟-⨟∘-assocL    : {A B C D : Obj} {f : Mor A B} {g : KMor B C} {h : KMor C D}
                   → f ⨟ (g ⨟∘ h) ≈ (f ⨟ g) ⨟∘ h
  ⨟-⨟∘-assocL = ≈-sym ⨟-⨟∘-assoc

  ⨟∘-assoc : {A B C D : Obj} {f : KMor A B} {g : KMor B C} {h : KMor C D}
             → (f ⨟∘ g) ⨟∘ h ≈ f ⨟∘ g ⨟∘ h
  ⨟∘-assoc {A} {B} {C} {D} {f} {g} {h} = ≈-begin
                (f ⨟∘ g) ⨟∘ h
        ≈⟨ ≈-refl ⟩
                (f ⨟ M-mor g ⨟ join) ⨟ M-mor h ⨟ join
        ≈⟨ ⨟-assoc ⟨≈≈⟩ ⨟-cong₂ (⨟-assocL ⟨≈≈⟩ ⨟-cong₁ ⨟-assoc) ⟩
```

```
            f ⨟ (M-mor g ⨟ join ⨟ M-mor h) ⨟ join
      ≈⟨ ⨟-cong₂₁ (⨟-cong₂ join-naturality ⟨≈˘≈⟩ ⨟-assocL) ⟩
            f ⨟ ((M-mor g ⨟ M-mor (M-mor h)) ⨟ join) ⨟ join
      ≈⟨ ⨟-cong₂ (⨟-assoc ⟨≈≈⟩ ⨟-cong₂ assoc ⟨≈≈⟩ ⨟-assocL) ⟩
            f ⨟ ((M-mor g ⨟ M-mor (M-mor h)) ⨟ M-mor join) ⨟ join
      ≈⟨ ⨟-cong₂₁ ((⨟-assoc ⟨≈≈˘⟩ ⨟-cong₂ M-mor-⨟) ⟨≈≈˘⟩ M-mor-⨟) ⟩
            f ⨟ M-mor (g ⨟ M-mor h ⨟ join) ⨟ join
      ≈⟨ ≈-refl ⟩
            f ⨟∘ g ⨟∘ h
      □
⨟∘-assocL       : {A B C D : Obj} {f : KMor A B} {g : KMor B C} {h : KMor C D}
          → f ⨟∘ g ⨟∘ h ≈ (f ⨟∘ g) ⨟∘ h
⨟∘-assocL = ≈-sym ⨟∘-assoc
KleisliCompOp : CompOp KHom
KleisliCompOp = record
  { _⨟_ = _⨟∘_
  ; ⨟-cong = ⨟∘-cong
  ; ⨟-assoc = ⨟∘-assoc
  }
KleisliSG : Semigroupoid j k Obj
KleisliSG = record
  {Hom = KHom
  ; compOp = KleisliCompOp
  }
return-⨟∘ : {A B : Obj} {f : KMor A B} → return ⨟∘ f ≈ f
return-⨟∘ {A} {B} {f} = ≈-begin
            return ⨟∘ f
      ≈⟨ ≈-refl ⟩
            return ⨟ M-mor f ⨟ join
      ≈⟨ ⨟-assocL ⟩
            (return ⨟ M-mor f) ⨟ join
      ≈⟨ ⨟-cong₁ (≈-sym return-naturality) ⟩
            (f ⨟ return) ⨟ join
      ≈⟨ ⨟-assoc ⟩
            f ⨟ return ⨟ join
      ≈⟨ ≈Id-isRightIdentity leftUnit ⟩
            f
      □
⨟return-⨟∘ : {A B C : Obj} {f : Mor A B} {g : KMor B C} → (f ⨟ return) ⨟∘ g ≈ f ⨟ g
⨟return-⨟∘ = ⨟-⨟∘-assoc ⟨≈≈⟩ ⨟-cong₂ return-⨟∘


isMono-⇒ : {A B : Obj} {f : KMor A B} → Semigroupoid.isMono KleisliSG f → isMono (return {A}) → isMono f
isMono-⇒ {A} {B} {f} f-isMonoK returnA-isMono {Z} {g} {h} g⨟f≈h⨟f
      =  returnA-isMono {Z} {g} {h}
            (f-isMonoK {Z} {g ⨟ return} {h ⨟ return}
              (≈-begin
                  (g ⨟ return) ⨟∘ f
                ≈⟨ ⨟return-⨟∘ ⟩
                  g ⨟ f
                ≈⟨ g⨟f≈h⨟f ⟩
                  h ⨟ f
                ≈˘⟨ ⨟return-⨟∘ ⟩
                  (h ⨟ return) ⨟∘ f
                □))


isMono-⇐ : {A B : Obj} {f : Mor A B} → isMono (M-mor f) → Semigroupoid.isMono KleisliSG (f ⨟ return)
isMono-⇐ {A} {B} {f} Mf-isMono {Z} {g} {h} g⨟∘f≈h⨟∘f = Mf-isMono {Z} {g} {h}
```

```
      (≈-begin
           g ⨟ M-mor f
        ≈˘⟨ ⨟-cong₂ (≈Id-isRightIdentity rightUnit) ⟩
           g ⨟ M-mor f ⨟ M-mor return ⨟ join
        ≈˘⟨ ⨟-cong₂ (⨟-cong₁ M-mor-⨟ ⟨≈≈⟩ ⨟-assoc) ⟩
           g ⨟ M-mor (f ⨟ return) ⨟ join
        ≈⟨   g⨟∘f≈h⨟∘f ⟩
           h ⨟ M-mor (f ⨟ return) ⨟ join
        ≈⟨ ⨟-cong₂ (⨟-cong₁ M-mor-⨟ ⟨≈≈⟩ ⨟-assoc) ⟩
           h ⨟ M-mor f ⨟ M-mor return ⨟ join
        ≈⟨ ⨟-cong₂ (≈Id-isRightIdentity rightUnit) ⟩
           h ⨟ M-mor f
        □)


isEpi-⇐ : {A B : Obj} {f : Mor A B} → isEpi f → Semigroupoid.isEpi KleisliSG (f ⨟ return)
isEpi-⇐ {A} {B} {f} f-isEpi {Z} {g} {h} f⨟∘g≈f⨟∘h = f-isEpi {M-obj Z} {g} {h}
  (
    (≈-begin
           f ⨟ g
        ≈˘⟨ ⨟-cong₂ (≈Id-isRightIdentity leftUnit) ⟩
           f ⨟ g ⨟ return ⨟ join
        ≈˘⟨ ⨟-assoc ⟨≈≈˘⟩ ⨟-cong₂ (⨟-cong₁&₂₁ return-naturality) ⟩
           (f ⨟ return) ⨟ M-mor g ⨟ join
        ≈⟨   f⨟∘g≈f⨟∘h ⟩
           (f ⨟ return) ⨟ M-mor h ⨟ join
        ≈⟨ ⨟-assoc ⟨≈≈˘⟩ ⨟-cong₂ (⨟-cong₁&₂₁ return-naturality) ⟩
           f ⨟ h ⨟ return ⨟ join
        ≈⟨ ⨟-cong₂ (≈Id-isRightIdentity leftUnit) ⟩
           f ⨟ h
        □)
  )


⨟∘-return : {A B : Obj} {f : KMor A B} → f ⨟∘ return ≈ f
⨟∘-return {A} {B} {f} = ≈-begin
           f ⨟∘ return
    ≈⟨ ≈-refl ⟩
           f ⨟ M-mor return ⨟ join
    ≈⟨ ≈Id-isRightIdentity rightUnit ⟩
           f
    □


KleisliIdOp : IdOp KHom _⨟∘_
KleisliIdOp = record
  {Id = return
  ; leftId = return-⨟∘
  ; rightId = ⨟∘-return
  }
KleisliCat : Category j k Obj
KleisliCat = record
  {semigroupoid = KleisliSG
  ; idOp = KleisliIdOp
  }
open CatFinColimits 𝒞


KleisliIsCoproduct      : {A B S : Obj} {ι : Mor A S} {κ : Mor B S}
                        → (isCoproduct : IsCoproduct ι κ)
```

```
                        → SGFinColimits.IsCoproduct KleisliSG (ι ⨟ return) (κ ⨟ return)
  KleisliIsCoproduct {A} {B} {S} {ι} {κ} isCoproduct {C} F G = let
      ι′ = ι ⨟ return
      κ′ = κ ⨟ return
      open IsCoproduct isCoproduct
    in record
      {univMor = F ▵ G
      ; univMor-factors-left = ≈-begin
                  (ι ⨟ return) ⨟∘ (F ▵ G)
                ≈⟨ ⨟return-⨟∘ ⟩
                  ι ⨟ (F ▵ G)
                ≈⟨ ι⨟▵ ⟩
                  F
                □
      ; univMor-factors-right = ≈-begin
                  (κ ⨟ return) ⨟∘ (F ▵ G)
                ≈⟨ ⨟return-⨟∘ ⟩
                  κ ⨟ (F ▵ G)
                ≈⟨ κ⨟▵ ⟩
                  G
                □
      ; univMor-unique = λ {V} (eq₁ : (ι ⨟ return) ⨟∘ V ≈ F)
                               (eq₂ : (κ ⨟ return) ⨟∘ V ≈ G)
          → ≈-begin
                  V
                ≈⟨ Id⊞-isLeftIdentity ⟨≈˘≈⟩ ▵-⨟ ⟩
                  (ι ⨟ V) ▵ (κ ⨟ V)
                ≈⟨ ▵-cong (⨟return-⨟∘ ⟨≈˘≈⟩ eq₁) (⨟return-⨟∘ ⟨≈˘≈⟩ eq₂) ⟩
                  F ▵ G
                □
      }

  KleisliHasCoproducts : HasCoproducts → SGFinColimits.HasCoproducts KleisliSG
  KleisliHasCoproducts hasCoproducts = let open HasCoproducts hasCoproducts
    in record {_⊞_ = _⊞_; ι = ι ⨟ return; κ = κ ⨟ return; isCoproduct = KleisliIsCoproduct isCoproduct}


module Monad′ {i j k : Level} {Obj : Set i} {𝒞 : Category j k Obj} (ℳ : Monad 𝒞) where
  open Monad ℳ public
  open Kleisli ℳ public
  open Functor M public


module Monad₁ {i j k : Level} {Obj : Set i} {𝒞 : Category j k Obj} (ℳ₁ : Monad 𝒞) where
  open Monad-M ℳ₁ public using () renaming
    (M                 to M₁
    ; M-return         to M₁-return
    ; M-join           to M₁-join
    ; M-obj            to M₁-obj
    ; M-mor            to M₁-mor
    ; M-cong           to M₁-cong
    ; M-mor-⨟          to M₁-mor-⨟
    ; M-Id             to M₁-Id
    ; return           to return₁
    ; return-naturality to return₁-naturality
    ; join             to join₁
    ; join-naturality  to join₁-naturality
    ; leftUnit         to leftUnit₁
    ; rightUnit        to rightUnit₁
```

```
    ; assoc              to assoc₁
    ; KHom               to KHom₁
    ; KMor               to KMor₁
    ; ↼                  to ↼₁
    ; isMono-⇐M          to isMono-⇐M₁
    )
  open Kleisli 𝓜₁ public using () renaming
    (_⨟∘_                to _⨟∘₁_
    ; ⨟∘-cong            to ⨟∘₁-cong
    ; ⨟∘-cong₁           to ⨟∘₁-cong₁
    ; ⨟∘-cong₂           to ⨟∘₁-cong₂
    ; ⨟∘-⨟-assoc         to ⨟∘₁-⨟-assoc
    ; ⨟∘-⨟-assocL        to ⨟∘₁-⨟-assocL
    ; ⨟-⨟∘-assoc         to ⨟-⨟∘₁-assoc
    ; ⨟-⨟∘-assocL        to ⨟-⨟∘₁-assocL
    ; ⨟∘-assoc           to ⨟∘₁-assoc
    ; ⨟∘-assocL          to ⨟∘₁-assocL
    ; return-⨟∘          to return-⨟∘₁
    ; ⨟return-⨟∘         to ⨟return-⨟∘₁
    ; ⨟∘-return          to ⨟∘₁-return
    )

module Monad₂ {i j k : Level} {Obj : Set i} {𝒞 : Category j k Obj} (𝓜₂ : Monad 𝒞) where
  open Monad-M 𝓜₂ public using () renaming
    (M                   to M₂
    ; M-return           to M₂-return
    ; M-join             to M₂-join
    ; M-obj              to M₂-obj
    ; M-mor              to M₂-mor
    ; M-cong             to M₂-cong
    ; M-mor-⨟            to M₂-mor-⨟
    ; M-Id               to M₂-Id
    ; return             to return₂
    ; return-naturality  to return₂-naturality
    ; join               to join₂
    ; join-naturality    to join₂-naturality
    ; leftUnit           to leftUnit₂
    ; rightUnit          to rightUnit₂
    ; assoc              to assoc₂
    ; KHom               to KHom₂
    ; KMor               to KMor₂
    ; ↼                  to ↼₂
    ; isMono-⇐M          to isMono-⇐M₂
    )
  open Kleisli 𝓜₂ public using () renaming
    (_⨟∘_                to _⨟∘₂_
    ; ⨟∘-cong            to ⨟∘₂-cong
    ; ⨟∘-cong₁           to ⨟∘₂-cong₁
    ; ⨟∘-cong₂           to ⨟∘₂-cong₂
    ; ⨟∘-⨟-assoc         to ⨟∘₂-⨟-assoc
    ; ⨟∘-⨟-assocL        to ⨟∘₂-⨟-assocL
    ; ⨟-⨟∘-assoc         to ⨟-⨟∘₂-assoc
    ; ⨟-⨟∘-assocL        to ⨟-⨟∘₂-assocL
    ; ⨟∘-assoc           to ⨟∘₂-assoc
    ; ⨟∘-assocL          to ⨟∘₂-assocL
    ; return-⨟∘          to return-⨟∘₂
    ; ⨟return-⨟∘         to ⨟return-⨟∘₂
    ; ⨟∘-return          to ⨟∘₂-return
    )
```

```
module Monad₃ {i j k : Level} {Obj : Set i} {𝒞 : Category j k Obj} (ℳ₃ : Monad 𝒞) where
  open Monad-M ℳ₃ public using () renaming
    (M                  to M₃
    ; M-return          to M₃-return
    ; M-join            to M₃-join
    ; M-obj             to M₃-obj
    ; M-mor             to M₃-mor
    ; M-cong            to M₃-cong
    ; M-mor-⨟           to M₃-mor-⨟
    ; M-Id              to M₃-Id
    ; return            to return₃
    ; return-naturality to return₃-naturality
    ; join              to join₃
    ; join-naturality   to join₃-naturality
    ; leftUnit          to leftUnit₃
    ; rightUnit         to rightUnit₃
    ; assoc             to assoc₃
    ; KHom              to KHom₃
    ; KMor              to KMor₃
    ; ↼                 to ↼₃
    ; isMono-⇐M         to isMono-⇐M₃
    )
  open Kleisli ℳ₃ public using () renaming
    (_⨟∘_               to _⨟∘₃_
    ; ⨟∘-cong           to ⨟∘₃-cong
    ; ⨟∘-cong₁          to ⨟∘₃-cong₁
    ; ⨟∘-cong₂          to ⨟∘₃-cong₂
    ; ⨟∘-⨟-assoc        to ⨟∘₃-⨟-assoc
    ; ⨟∘-⨟-assocL       to ⨟∘₃-⨟-assocL
    ; ⨟-⨟∘-assoc        to ⨟-⨟∘₃-assoc
    ; ⨟-⨟∘-assocL       to ⨟-⨟∘₃-assocL
    ; ⨟∘-assoc          to ⨟∘₃-assoc
    ; ⨟∘-assocL         to ⨟∘₃-assocL
    ; return-⨟∘         to return-⨟∘₃
    ; ⨟return-⨟∘        to ⨟return-⨟∘₃
    ; ⨟∘-return         to ⨟∘₃-return
    )


Id-ℳ : {i j k : Level} {Obj : Set i} {𝒞 : Category j k Obj} → Monad 𝒞
Id-ℳ {𝒞 = 𝒞} = let open Category 𝒞 in record
  {M = Identity 𝒞
  ; trafos = record
    {M-return = IdTrans (Identity 𝒞)
    ; M-join = NatLeftId ≅ₙ IdTrans (Identity 𝒞)
    ; leftUnit = leftId ⟨≈≈⟩ leftId
    ; rightUnit = leftId ⟨≈≈⟩ leftId
    ; assoc = ≈-refl
    }
  }
```

2.2 Categoric.Monad.FunctorMonad

```
open import RATH.Level
open import Categoric.LESGraph using (LocalSetoid; module LocalSetoidCalc₃)
```

```
open import Categoric.Category
open import Categoric.Functor
open import Categoric.Monad
open import Categoric.IdOp
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
```

A FunctorMonad on two categories $\mathcal{C}_1$ and $\mathcal{C}_2$ is a bifunctor $\mathcal{M}$ from the product category of $\mathcal{C}_1$ and $\mathcal{C}_2$ to $\mathcal{C}_2$ that can be understood as a functor from $\mathcal{C}_1$ to the category of monads over $\mathcal{C}_2$.

return and join need to be natural transformations to $\mathcal{M}$, starting from the product category, with the resulting naturality conditions FM-return and FM-join.

```
module Categoric.Monad.FunctorMonad
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} {𝒞₁ : Category {i₁} j₁ k₁ Obj₁}
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} {𝒞₂ : Category {i₂} j₂ k₂ Obj₂} where
open Category₁ 𝒞₁
open Category₂ 𝒞₂
private
  module 𝒞₁ = Category 𝒞₁
  module 𝒞₂ = Category 𝒞₂
record FunctorMonad : Set (i₁ ⊎ i₂ ⊎ j₁ ⊎ j₂ ⊎ k₁ ⊎ k₂) where
  field
    bifunctor : Bifunctor 𝒞₁ 𝒞₂ 𝒞₂
  private
    module ℳ = Bifunctor bifunctor
  field
    monadTrafos : {A : Obj₁} → MonadTrafos (ℳ.functor₂ {A})
  ℳ : {A : Obj₁} → Monad 𝒞₂
  ℳ {A} = record {M = ℳ.functor₂ {A}; trafos = monadTrafos {A}}
  open module ℳ′ {A : Obj₁} = Monad-M (ℳ {A})
  field
    FM-return : {A₁ A₂ : Obj₁} {B₁ B₂ : Obj₂} {f : Mor₁ A₁ A₂} {g : Mor₂ B₁ B₂}
              → return {A₁} {B₁} ⨟₂ ℳ.mor f g ≈₂ g ⨟₂ return {A₂} {B₂}
    FM-join   : {A₁ A₂ : Obj₁} {B₁ B₂ : Obj₂} {f : Mor₁ A₁ A₂} {g : Mor₂ B₁ B₂}
              → join {A₁} {B₁} ⨟₂ ℳ.mor f g ≈₂ ℳ.mor f (ℳ.mor f g) ⨟₂ join {A₂} {B₂}
```

After the last **field**, **public** re-exports:

```
  open ℳ public
  open ℳ′ public
  open module K ℳ {A : Obj₁} = Kleisli (ℳ {A}) public


  ObjPair : Set (i₁ ⊎ i₂)
  ObjPair = Obj₁ × Obj₂
  module ObjPair (op : ObjPair) where
    Carrier₁ : Obj₁
    Carrier₁ = proj₁ op
    Carrier₂ : Obj₂
    Carrier₂ = proj₂ op
    Carrier′ : Obj₂
    Carrier′ = ℳ.obj Carrier₁ Carrier₂


  FMMor : (A B : ObjPair) → Set (j₁ ⊎ j₂)
  FMMor (A₁ , A₂) (B₁ , B₂) = Mor₁ A₁ B₁ × Mor₂ A₂ (ℳ.obj B₁ B₂)


  module FMMor {A B : ObjPair} (f : FMMor A B) where
    open ObjPair A renaming (Carrier₁ to A₁; Carrier₂ to A₂; Carrier′ to A′)
```

```
  open ObjPair B renaming (Carrier₁ to B₁; Carrier₂ to B₂; Carrier′ to B′)
  mor₁ : Mor₁ A₁ B₁
  mor₁ = proj₁ f
  mor₂ : Mor₂ A₂ B′
  mor₂ = proj₂ f
  mor′ : Mor₂ A′ B′
  mor′ = 𝓜.mor mor₁ mor₂ ⨟₂ join


FMHom : LocalSetoid ObjPair (j₁ ⊎ j₂) (k₁ ⊎ k₂)
FMHom A B = let open FMMor in record
  {Carrier = FMMor A B
  ; _≈_ = λ F G → mor₁ F ≈₁ mor₁ G × mor₂ F ≈₂ mor₂ G
  ; isEquivalence = record
      {refl = ≈₁-refl, ≈₂-refl
      ; sym = λ { (F₁≈G₁, F₂≈G₂) → ≈₁-sym F₁≈G₁, ≈₂-sym F₂≈G₂}
      ; trans = λ { (F₁≈G₁, F₂≈G₂) (G₁≈H₁, G₂≈H₂)
                  → ≈₁-trans F₁≈G₁ G₁≈H₁, ≈₂-trans F₂≈G₂ G₂≈H₂}
      }
  }
open LocalSetoidCalc₃ FMHom
mor′-cong : {A B : ObjPair} {F G : FMMor A B} (open FMMor)
          → mor₁ F ≈₁ mor₁ G × mor₂ F ≈₂ mor₂ G → mor′ F ≈₂ mor′ G
mor′-cong {A} {B} {F} {G} (F₁≈G₁, F₂≈G₂) = 𝒞₂.⨟-cong₁ (𝓜.mor-cong F₁≈G₁ F₂≈G₂)


module FMMor-Comp {A B C : ObjPair} (F : FMMor A B) (G : FMMor B C) where
  open ObjPair A public renaming (Carrier₁ to A₁; Carrier₂ to A₂; Carrier′ to A′)
  open ObjPair B public renaming (Carrier₁ to B₁; Carrier₂ to B₂; Carrier′ to B′)
  open ObjPair C public renaming (Carrier₁ to C₁; Carrier₂ to C₂; Carrier′ to C′)
  module F = FMMor F
  module G = FMMor G
  FMMor-comp : FMMor A C
  FMMor-comp = (F.mor₁ ⨟₁ G.mor₁), (F.mor₂ ⨟₂ G.mor′)
  module H = FMMor FMMor-comp


  FMMor-comp′ : F.mor′ ⨟₂ G.mor′ ≈₂ H.mor′
  FMMor-comp′ = ≈₂-begin
      (𝓜.mor F.mor₁ F.mor₂ ⨟₂ join {B₁}) ⨟₂ 𝓜.mor G.mor₁ G.mor₂ ⨟₂ join {C₁}
    ≈₂⟨ 𝒞₂.⨟-cong₂ (𝒞₂.⨟-cong₁ 𝓜.mor-₂⨟₁ ⟨≈₂≈⟩ 𝒞₂.⨟-assoc) ⟨≈₂≈⟩ 𝒞₂.⨟-assoc ⟩
      𝓜.mor F.mor₁ F.mor₂ ⨟₂ join {B₁} ⨟₂ M-mor G.mor₂ ⨟₂ 𝓜.mor G.mor₁ Id₂ ⨟₂ join {C₁}
    ≈₂˘⟨ 𝒞₂.⨟-cong₂ (𝒞₂.⨟-cong₁&₂₁ join-naturality) ⟩
      𝓜.mor F.mor₁ F.mor₂ ⨟₂ M-mor (M-mor G.mor₂) ⨟₂ join {B₁}
                                  ⨟₂ 𝓜.mor G.mor₁ Id₂ ⨟₂ join {C₁}
    ≈₂⟨ 𝒞₂.⨟-cong₂₂ (𝒞₂.⨟-cong₁&₂₁ FM-join ⟨≈₂≈⟩ 𝒞₂.⨟-cong₂ assoc) ⟩
      𝓜.mor F.mor₁ F.mor₂ ⨟₂ M-mor (M-mor G.mor₂) ⨟₂
      𝓜.mor G.mor₁ (𝓜.mor G.mor₁ Id₂) ⨟₂ M-mor join ⨟₂ join
    ≈₂˘⟨ 𝒞₂.⨟-cong₂₂ (𝒞₂.⨟-cong₁ 𝓜.mor-₁⨟ ⟨≈₂≈⟩ 𝒞₂.⨟-assoc) ⟩
      𝓜.mor F.mor₁ F.mor₂ ⨟₂ M-mor (M-mor G.mor₂) ⨟₂
      𝓜.mor G.mor₁ (𝓜.mor G.mor₁ Id₂ ⨟₂ join) ⨟₂ join
    ≈₂˘⟨ 𝒞₂.⨟-cong₂ (𝒞₂.⨟-cong₁ 𝓜.mor-⨟₁ ⟨≈₂≈⟩ 𝒞₂.⨟-assoc) ⟩
      𝓜.mor F.mor₁ F.mor₂ ⨟₂ 𝓜.mor G.mor₁ (M-mor G.mor₂ ⨟₂
      𝓜.mor G.mor₁ Id₂ ⨟₂ join) ⨟₂ join
    ≈₂˘⟨ 𝒞₂.⨟-cong₂₁ (𝓜.mor-cong₂ (𝒞₂.⨟-cong₁ 𝓜.mor-₂⨟₁ ⟨≈₂≈⟩ 𝒞₂.⨟-assoc)) ⟩
      𝓜.mor F.mor₁ F.mor₂ ⨟₂ 𝓜.mor G.mor₁ (𝓜.mor G.mor₁ G.mor₂ ⨟₂ join) ⨟₂ join
    ≈₂⟨ 𝒞₂.⨟-assocL ⟨≈₂≈˘⟩ 𝒞₂.⨟-cong₁ 𝓜.mor-⨟ ⟩
      𝓜.mor (F.mor₁ ⨟₁ G.mor₁) (F.mor₂ ⨟₂ 𝓜.mor G.mor₁ G.mor₂ ⨟₂ join) ⨟₂ join
    □₂
```

```
  open FMMor-Comp public using () renaming (FMMor-comp to comp; FMMor-comp′ to comp′)


  module FMMor-Comp3 {A B C D : ObjPair}
                     (F : FMMor A B) (G : FMMor B C) (H : FMMor C D) where
    module F = FMMor F
    module G = FMMor G
    module H = FMMor H
    FG = comp F G
    GH = comp G H
    module FG = FMMor FG
    module GH = FMMor GH
    module FG-H = FMMor (comp FG H)
    module F-GH = FMMor (comp F GH)
    FM-assoc : comp FG H ≈₃ comp F GH
    FM-assoc = (≈₁-begin
                  (F.mor₁ ⨟₁ G.mor₁) ⨟₁ H.mor₁
                ≈₁⟨ 𝒞₁.⨟-assoc ⟩
                  F.mor₁ ⨟₁ G.mor₁ ⨟₁ H.mor₁
                □₁)
                ,(≈₂-begin
                  (F.mor₂ ⨟₂ G.mor′) ⨟₂ H.mor′
                ≈₂⟨ 𝒞₂.⨟-assoc ⟨≈₂≈⟩ 𝒞₂.⨟-cong₂ (comp′ G H) ⟩
                  F.mor₂ ⨟₂ GH.mor′
                □₂)


  FMCategory : Category (j₁ ⊎ j₂) (k₁ ⊎ k₂) ObjPair
  FMCategory = record
    {semigroupoid = record
        {Hom = FMHom
        ;compOp = record
          {_⨟_ = comp
          ;⨟-cong = FMcomp-cong
          ;⨟-assoc = λ {A} {B} {C} {D} {F} {G} {H} → FMMor-Comp3.FM-assoc F G H
          }
        }
    ;idOp = record
        {Id = Id₁, return
        ;leftId = leftId₁, (𝒞₂.⨟-cong₁&₂₁ FM-return ⟨≈₂≈⟩ (𝒞₂.⨟-cong₂ leftUnit ⟨≈₂≈⟩ rightId₂))
        ;rightId = rightId₁, (𝒞₂.⨟-cong₂ rightUnit ⟨≈₂≈⟩ rightId₂)
        }
    }
    where
        FMcomp-cong : {A B C : ObjPair} {F₁ F₂ : FMMor A B} {G₁ G₂ : FMMor B C}
                      → F₁ ≈₃ F₂ → G₁ ≈₃ G₂ → comp F₁ G₁ ≈₃ comp F₂ G₂
        FMcomp-cong {A} {B} {C} {F₁₁, F₁₂} {F₂₁, F₂₂} {G₁₁, G₁₂} {G₂₁, G₂₂}
                    (F₁₁≈F₂₁, F₁₂≈F₂₂) (G₁₁≈G₂₁, G₁₂≈G₂₂)
          = 𝒞₁.⨟-cong F₁₁≈F₂₁ G₁₁≈G₂₁
          ,(≈₂-begin
              F₁₂ ⨟₂ mor G₁₁ G₁₂ ⨟₂ join
            ≈₂⟨ 𝒞₂.⨟-cong F₁₂≈F₂₂ (𝒞₂.⨟-cong₁ (mor-cong G₁₁≈G₂₁ G₁₂≈G₂₂)) ⟩
              F₂₂ 𝒞₂.⨟ mor G₂₁ G₂₂ 𝒞₂.⨟ join
            □₂)


ConstFunctorMonad : Monad 𝒞₂ → FunctorMonad
ConstFunctorMonad ℳ = record
  {bifunctor = ConstBifunctor ℳ.M
  ;monadTrafos = record
```

```
    {M-return = record {indmor = 𝓜.return; naturality = 𝓜.return-naturality}
    ; M-join = record {indmor = 𝓜.join; naturality = 𝓜.join-naturality}
    ; leftUnit = 𝓜.leftUnit
    ; rightUnit = 𝓜.rightUnit
    ; assoc = 𝓜.assoc
    }
  ; FM-return = ≈₂-sym 𝓜.return-naturality
  ; FM-join = ≈₂-sym 𝓜.join-naturality
  }
  where
    module 𝓜 = Monad′ 𝓜
```

2.3 Categoric.Monad.FunctorMonad.Monad

```
open import RATH.Level
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid; module LocalSetoidCalc₃)
open import Categoric.Semigroupoid
open import Categoric.Category
open import Categoric.Product.Category
open import Categoric.Functor
open import Categoric.Monad
open import Categoric.Monad.FunctorMonad
open import Categoric.IdOp
  -- open import Relation.Binary using (Setoid ; module Setoid)
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
```

A FunctorMonad on two categories $\mathcal{C}_1$ and $\mathcal{C}_2$ gives rise to a monad on the product category $\mathcal{C}_1 \times \mathcal{C}_2$:

```
module Categoric.Monad.FunctorMonad.Monad
    {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (𝒞₁ : Category {i₁} j₁ k₁ Obj₁)
    {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (𝒞₂ : Category {i₂} j₂ k₂ Obj₂)
    (𝓜 : FunctorMonad {𝒞₁ = 𝒞₁} {𝒞₂ = 𝒞₂}) where
𝒞₀ = ProductCategory 𝒞₁ 𝒞₂
open Category₀ 𝒞₀
open Category₁ 𝒞₁
open Category₂ 𝒞₂
private
  module 𝒞₀ = Category 𝒞₀
  module 𝒞₁ = Category 𝒞₁
  module 𝒞₂ = Category 𝒞₂
module 𝓜 = FunctorMonad 𝓜
FM-Monad : Monad 𝒞₀
FM-Monad = record
  {M = record
    {obj = λ {(A, B) → A, 𝓜.obj A B}
    ; mor = λ {(F, G) → F, 𝓜.mor F G}
    ; mor-cong = λ {(F₁≈F₂, G₁≈G₂) → F₁≈F₂, 𝓜.mor-cong F₁≈F₂ G₁≈G₂}
    ; mor-⨟ = ≈₁-refl, 𝓜.mor-⨟
    ; mor-Id = ≈₁-refl, 𝓜.mor-Id
    }
  ; trafos = record
    {M-return = record
      {indmor     = λ {{A, B} → Id₁, 𝓜.return {A} {B}}
      ; naturality = λ {{_} {_} {F, G} → (rightId₁ ⟨≈₁≈˘⟩ leftId₁), ≈₂-sym 𝓜.FM-return}
      }
```

```
    ; M-join = record
        {indmor     = λ {{A, B} → Id₁, 𝓜.join {A} {B}}
        ; naturality = λ {{_} {_} {F, G} → (rightId₁ ⟨≈₁≈˘⟩ leftId₁), ≈₂-sym 𝓜.FM-join}
        }
    ; leftUnit   = leftId₁, 𝓜.leftUnit
    ; rightUnit  = leftId₁, 𝓜.rightUnit
    ; assoc      = ≈₁-refl, 𝓜.assoc
    }
  }
```

2.4 Categoric.Monad.FunctorMonad.Coproduct

```
module Categoric.Monad.FunctorMonad.Coproduct where

open import RATH.Level

open import Categoric.Category using (Category; module Category)
open import Categoric.Category.FinColimits using (module CatFinColimits)
open import Categoric.Functor using (Bifunctor; module Bifunctor)
open import Categoric.Functor.Coproduct using (CoproductBifunctor)
open import Categoric.Monad.FunctorMonad using (FunctorMonad)


CoproductFM : {i j k : Level} {Obj : Set i}
  → (C : Category j k Obj)
  → CatFinColimits.HasCoproducts C
  → FunctorMonad {𝓒₁ = C} {𝓒₂ = C}
CoproductFM {Obj = Obj} C hasCoproducts = let
    open Category C
    open CatFinColimits C using (module HasCoproducts)
    open HasCoproducts hasCoproducts
    ⊕join : {A B : Obj} → Mor (A ⊞ (A ⊞ B)) (A ⊞ B)
    ⊕join {A} {B} = ι ◬ Id {A ⊞ B}
  in record
    {bifunctor = CoproductBifunctor C hasCoproducts
    ; monadTrafos = record
      {M-return = record
        {indmor = κ
        ; naturality = λ {A} {B} {f} → ≈-begin
              f ⨟ κ
            ≈˘⟨ κ⨟⊕ ⟩
              κ ⨟ (Id ⊕ f)
            □}
      ; M-join = record
        {indmor = ⊕join
        ; naturality = λ {A} {B} {f} → ≈-begin
              (Id ⊕ (Id ⊕ f)) ⨟ (ι ◬ Id)
            ≈⟨ ⊕-⨟-◬ ⟨≈≈⟩ ◬-cong leftId rightId ⟩
              ι ◬ (Id ⊕ f)
            ≈˘⟨ ◬-⨟ ⟨≈≈⟩ ◬-cong (ι⨟⊕ ⟨≈≈⟩ leftId) leftId ⟩
              (ι ◬ Id) ⨟ (Id ⊕ f)
            □
        }
      ; leftUnit = λ {A} → ≈-begin
              κ ⨟ (ι ◬ Id)
            ≈⟨ κ⨟◬ ⟩
              Id
            □
```

```
    ; rightUnit = λ {A} → ≈-begin
            (Id ⊕ κ) ⨟ (ι ⧋ Id)
          ≈⟨ ⊕-⨟-⧋ ⟨≈≈⟩ ⧋-cong leftId rightId ⟨≈≈⟩ Id⊞≈Id ⟩
            Id
          □
    ; assoc = λ {A} → ≈-begin
            (ι ⧋ Id) ⨟ (ι ⧋ Id)
          ≈⟨ ⧋-⨟ ⟨≈≈⟩ ⧋-cong ι⨟⧋ leftId ⟩
            ι ⧋ (ι ⧋ Id)
          ≈˘⟨ ⊕-⨟-⧋ ⟨≈≈⟩ ⧋-cong leftId rightId ⟩
            (Id ⊕ (ι ⧋ Id)) ⨟ (ι ⧋ Id)
          □
    }
  ; FM-return = λ {A₁} {A₂} {B₁} {B₂} {f} {g} → ≈-begin
            κ ⨟ (f ⊕ g)
          ≈⟨ κ⨟⊕ ⟩
            g ⨟ κ
          □
  ; FM-join = λ {A₁} {A₂} {B₁} {B₂} {f} {g} → ≈-begin
            (ι ⧋ Id) ⨟ (f ⊕ g)
          ≈⟨ ⧋-⨟ ⟨≈≈⟩ ⧋-cong ι⨟⊕ leftId ⟩
            f ⨟ ι ⧋ (f ⊕ g)
          ≈˘⟨ ⊕-⨟-⧋ ⟨≈≈⟩ ⧋-cong₂ rightId ⟩
            (f ⊕ (f ⊕ g)) ⨟ (ι ⧋ Id)
          □
  }
```

2.5 Categoric.Monad.Product

```
open import RATH.Level
open import Categoric.Category
open import Categoric.Semigroupoid
open import Categoric.Functor
open import Categoric.Monad
open import Categoric.Product.Category
open import Function using (id)
open import RATH.Data.Product
open import RATH.PropositionalEquality using (≡-refl)
```

Let two monads on the constituent categories of a product category be given:

```
module Categoric.Monad.Product
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} {𝒞₁ : Category {i₁} j₁ k₁ Obj₁} (ℳ₁ : Monad 𝒞₁)
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} {𝒞₂ : Category {i₂} j₂ k₂ Obj₂} (ℳ₂ : Monad 𝒞₂)
  where
private
  𝒞₃ = ProductCategory 𝒞₁ 𝒞₂
open Category
open Semigroupoid₁ (semigroupoid 𝒞₁)
open Semigroupoid₂ (semigroupoid 𝒞₂)
open Semigroupoid₃ (semigroupoid 𝒞₃)
open Monad₁ ℳ₁
open Monad₂ ℳ₂
```

This induces a monad on the product category:

```
ProductMonad : Monad 𝒞₃
ProductMonad = record
  {M = record
    {obj = map M₁-obj M₂-obj
    ; mor = map M₁-mor M₂-mor
    ; mor-cong = λ eq → M₁-cong (proj₁ eq), M₂-cong (proj₂ eq)
    ; mor-⨾ = M₁-mor-⨾, M₂-mor-⨾
    ; mor-Id = M₁-Id, M₂-Id
    }
  ; trafos = record
    {M-return = record
      {indmor = return₁, return₂
      ; naturality = return₁-naturality, return₂-naturality
      }
    ; M-join = record
      {indmor = join₁, join₂
      ; naturality = join₁-naturality, join₂-naturality
      }
    ; leftUnit = leftUnit₁, leftUnit₂
    ; rightUnit = rightUnit₁, rightUnit₂
    ; assoc = assoc₁, assoc₂
    }
  }
```

Furthermore, the Kleisli category of this product monad is isomorphic to the product category of the Kleisli categories of the two constituent monads:

```
open Kleisli using (KleisliCat)
open Monad₃ ProductMonad


𝒦₁ = KleisliCat ℳ₁
𝒦₂ = KleisliCat ℳ₂
𝒦₃ = KleisliCat ProductMonad


toProductKleisli : Functor (ProductCategory 𝒦₁ 𝒦₂) 𝒦₃
toProductKleisli = record
  {obj = id
  ; mor = id
  ; mor-cong = λ e → e
  ; mor-⨾ = ≈-refl 𝒞₁, ≈-refl 𝒞₂
  ; mor-Id = ≈-refl 𝒞₁, ≈-refl 𝒞₂
  }


toProductKleisliFF : CatF.IsFullAndFaithful toProductKleisli
toProductKleisliFF {A} {B}
  = record { _⟨$⟩_ = id; cong = λ e → e}
  , record   {left-inverse-of = λ _ → ≈-refl 𝒞₁, ≈-refl 𝒞₂
             ; right-inverse-of = λ _ → ≈-refl 𝒞₁, ≈-refl 𝒞₂}


fromProductKleisli : Functor 𝒦₃ (ProductCategory 𝒦₁ 𝒦₂)
fromProductKleisli = FFInverse toProductKleisli toProductKleisliFF id ≡-refl
```

2.6 Categoric.Monad.DepProduct

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

```
open import Categoric.Semigroupoid
open import Categoric.Functor
open import Categoric.Monad
open import Categoric.Monad.FunctorMonad
open import Categoric.Product.Category
open import Function using (id)
open import RATH.Data.Product
```

For two categories $\mathcal{C}_1$ and $\mathcal{C}_2$, let a monad $\mathcal{M}_1$ on $\mathcal{C}_1$ be given, and a monad $\mathcal{M}_2$ on $\mathcal{C}_2$ that is parameterised over $\mathcal{C}_1$:

```
module Categoric.Monad.DepProduct
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁}
  {𝒞₁ : Category {i₁} j₁ k₁ Obj₁} (ℳ₁ : Monad 𝒞₁)
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂}
  {𝒞₂ : Category {i₂} j₂ k₂ Obj₂} (ℳ₂ : FunctorMonad {𝒞₁ = 𝒞₁} {𝒞₂ = 𝒞₂})
  where
open Category₁ 𝒞₁
open Category₂ 𝒞₂
open Monad₁ ℳ₁
private
  module 𝒞₁ = Category 𝒞₁
  module 𝒞₂ = Category 𝒞₂
  module ℳ₂ = FunctorMonad ℳ₂
```

Together, $\mathcal{M}_1$ and $\mathcal{M}_2$ induce a monad on the product category of $\mathcal{C}_1$ and $\mathcal{C}_2$:

```
DepProductMonad : Monad (ProductCategory 𝒞₁ 𝒞₂)
DepProductMonad = record
  {M = record
    {obj = λ {(A₁ , A₂) → M₁-obj A₁ , ℳ₂.obj (M₁-obj A₁) A₂}
    ;mor = λ {{A₁ , A₂} {B₁ , B₂} (f₁ , f₂) → M₁-mor f₁ , ℳ₂.mor (M₁-mor f₁) f₂}
    ;mor-cong = λ {(f₁≈g₁ , f₂≈g₂) → M₁-cong f₁≈g₁ , ℳ₂.mor-cong (M₁-cong f₁≈g₁) f₂≈g₂}
    ;mor-⨟ = M₁-mor-⨟ , (ℳ₂.mor-cong₁ M₁-mor-⨟ ⟨≈₂≈⟩ ℳ₂.mor-⨟)
    ;mor-Id = M₁-Id , (ℳ₂.mor-cong₁ M₁-Id ⟨≈₂≈⟩ ℳ₂.mor-Id)
    }
  ;trafos = record
    {M-return = record
      {indmor = return₁ , ℳ₂.return
      ;naturality = return₁-naturality , ≈₂-sym ℳ₂.FM-return
      }
    ;M-join = record
      {indmor = join₁ , ℳ₂.mor join₁ Id₂ ⨟₂ ℳ₂.join
      ;naturality = λ {{_} {_} {f₁ , f₂} → join₁-naturality , (≈₂-begin
          ℳ₂.mor (M₁-mor (M₁-mor f₁)) (ℳ₂.mor (M₁-mor f₁) f₂) ⨟₂ ℳ₂.mor join₁ Id₂ ⨟₂ ℳ₂.join
        ≈₂⟨ 𝒞₂.⨟-assocL ⟨≈₂≈˘⟩ 𝒞₂.⨟-cong₁ ℳ₂.mor-⨟ ⟩
          ℳ₂.mor (M₁-mor (M₁-mor f₁) ⨟₁ join₁) (ℳ₂.mor (M₁-mor f₁) f₂ ⨟₂ Id₂) ⨟₂ ℳ₂.join
        ≈₂⟨ 𝒞₂.⨟-cong₁ (ℳ₂.mor-cong join₁-naturality rightId₂) ⟩
          ℳ₂.mor (join₁ ⨟₁ M₁-mor f₁) (ℳ₂.mor (M₁-mor f₁) f₂) ⨟₂ ℳ₂.join
        ≈₂⟨ 𝒞₂.⨟-cong₁ ℳ₂.mor-⨟₂ ⟨≈₂≈⟩ 𝒞₂.⨟-assoc ⟩
          ℳ₂.mor join₁ Id₂ ⨟₂ ℳ₂.mor (M₁-mor f₁) (ℳ₂.mor (M₁-mor f₁) f₂) ⨟₂ ℳ₂.join
        ≈₂⟨ 𝒞₂.⨟-cong₂ ℳ₂.FM-join ⟨≈₂˘≈⟩ 𝒞₂.⨟-assocL ⟩
          (ℳ₂.mor join₁ Id₂ ⨟₂ ℳ₂.join) ⨟₂ ℳ₂.mor (M₁-mor f₁) f₂
        □₂)}
      }
    ;leftUnit = leftUnit₁ , (≈₂-begin
          ℳ₂.return ⨟₂ ℳ₂.mor join₁ Id₂ ⨟₂ ℳ₂.join
        ≈₂⟨ 𝒞₂.⨟-cong₁&₂₁ ℳ₂.FM-return ⟩
```

```
      Id₂ ⨟₂ 𝓜₂.return ⨟₂ 𝓜₂.join
    ≈₂⟨ leftId₂ ⟨≈₂≈⟩ 𝓜₂.leftUnit ⟩
      Id₂
    □₂)
  ; rightUnit = rightUnit₁, (≈₂-begin
      𝓜₂.mor (M₁-mor return₁) 𝓜₂.return ⨟₂ 𝓜₂.mor join₁ Id₂ ⨟₂ 𝓜₂.join
    ≈₂⟨ 𝓒₂.⨟-assocL ⟨≈₂≈˘⟩ 𝓒₂.⨟-cong₁ 𝓜₂.mor-₂⨟ ⟩
      𝓜₂.mor (M₁-mor return₁ ⨟₁ join₁) 𝓜₂.return ⨟₂ 𝓜₂.join
    ≈₂⟨ 𝓒₂.⨟-cong₁ (𝓜₂.mor-cong₁ rightUnit₁) ⟨≈₂≈⟩ 𝓜₂.rightUnit ⟩
      Id₂
    □₂)
  ; assoc = assoc₁, (≈₂-begin
      (𝓜₂.mor join₁ Id₂ ⨟₂ 𝓜₂.join) ⨟₂ 𝓜₂.mor join₁ Id₂ ⨟₂ 𝓜₂.join
    ≈₂⟨ 𝓒₂.⨟-assoc ⟨≈₂≈⟩ 𝓒₂.⨟-cong₂ (𝓒₂.⨟-cong₁&₂₁ 𝓜₂.FM-join) ⟩
      𝓜₂.mor join₁ Id₂ ⨟₂ 𝓜₂.mor join₁ (𝓜₂.mor join₁ Id₂) ⨟₂ 𝓜₂.join ⨟₂ 𝓜₂.join
    ≈₂⟨ 𝓒₂.⨟-assocL ⟨≈₂≈⟩ 𝓒₂.⨟-cong (≈₂-sym 𝓜₂.mor-⨟₂) 𝓜₂.assoc ⟩
      𝓜₂.mor (join₁ ⨟₁ join₁) (𝓜₂.mor join₁ Id₂) ⨟₂ 𝓜₂.M-mor 𝓜₂.join ⨟₂ 𝓜₂.join
    ≈₂⟨ 𝓒₂.⨟-assocL ⟨≈₂≈˘⟩ 𝓒₂.⨟-cong₁ 𝓜₂.mor-₁⨟ ⟩
      𝓜₂.mor (join₁ ⨟₁ join₁) (𝓜₂.mor join₁ Id₂ ⨟₂ 𝓜₂.join) ⨟₂ 𝓜₂.join
    ≈₂⟨ 𝓒₂.⨟-cong₁ (𝓜₂.mor-cong₁ assoc₁ ⟨≈₂≈⟩ 𝓜₂.mor-₂⨟) ⟨≈₂≈⟩ 𝓒₂.⨟-assoc ⟩
      𝓜₂.mor (M₁-mor join₁) (𝓜₂.mor join₁ Id₂ ⨟₂ 𝓜₂.join) ⨟₂ 𝓜₂.mor join₁ Id₂ ⨟₂ 𝓜₂.join
    □₂)
  }
}
```

Chapter 3

Coalgebras

3.1 Categoric.Coalgebra.Semigroupoid

We base our initial study of coalgebras on semigroupoids, since we will need them in semigroupoids of finite presentations of functions or relations between infinite types, which was also the original motivation of Kahl (2008).

```
open import RATH.Level
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
open import Categoric.Semigroupoid
open import Categoric.SGFunctor
open import Categoric.IdOp


module Categoric.Coalgebra.Semigroupoid
  {i j k : Level} {Obj : Set i} {𝒞 : Semigroupoid {i} j k Obj} (ℱ : SGFunctor 𝒞 𝒞)
  where
open Semigroupoid 𝒞
private
  module ℱ = SGFunctor ℱ


record Coalgebra : Set (i ⊔ j) where
  field Carrier : Obj
        op : Mor Carrier (ℱ.obj Carrier)


record CAMor (A B : Coalgebra) : Set (j ⊔ k) 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
  mor′ : Mor (ℱ.obj A₀) (ℱ.obj B₀)
  mor′ = ℱ.mor mor
  field commutes : mor ⨟ B.op ≈ A.op ⨟ mor′
    -- [ WK: ] Having mor′ and commutes′ probably confuses more than it helps? [ ]
  commutes′ : mor′ ⨟ ℱ.mor B.op ≈ ℱ.mor (A.op ⨟ mor′)
  commutes′ = ≈-begin
      ℱ.mor mor ⨟ ℱ.mor B.op
    ≈˘⟨ ℱ.mor-⨟ ⟩
      ℱ.mor (mor ⨟ B.op)
    ≈⟨ ℱ.mor-cong commutes ⟩
```

```
      F.mor (A.op ⨟ mor′)
    □

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

  comp : CAMor A C
  comp = record
    {mor = F.mor ⨟ G.mor
    ;commutes = ≈-begin
        (F.mor ⨟ G.mor) ⨟ C.op
      ≈⟨ ⨟-assoc ⟨≈≈⟩ ⨟-cong₂ G.commutes ⟩
        F.mor ⨟ B.op ⨟ F.mor G.mor
      ≈⟨ ⨟-cong₁&₂₁ F.commutes ⟩
        A.op ⨟ F.mor F.mor ⨟ F.mor G.mor
      ≈˘⟨ ⨟-cong₂ F.mor-⨟ ⟩
        A.op ⨟ F.mor (F.mor ⨟ G.mor)
      □
    }
open CAMor-Comp public using (comp)

CASemigroupoid : Semigroupoid (j ⊔ k) k Coalgebra
CASemigroupoid = retract²Semigroupoid 𝒞
  (λ {A} {B} {C} F G → comp {A} {B} {C} F G)
  Carrier
  mor
  ≈-refl
  where open Coalgebra; open CAMor
```

3.2 Categoric.Coalgebra.Category

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

module Categoric.Coalgebra.Category
  {i j k : Level} {Obj : Set i} {𝒞 : Category {i} j k Obj} (ℱ : Functor 𝒞 𝒞)
  where
open Category 𝒞
private
  module ℱ = Functor ℱ
open import Categoric.Coalgebra.Semigroupoid (CatF.sgFunctor ℱ) public

CA-Id : {A : Coalgebra} → CAMor A A
CA-Id {A} = record
```

```
  {mor = Id {A.Carrier}
  ;commutes = ≈-begin
        Id ⨟ A.op
    ≈⟨ leftId ⟩
        A.op
    ≈˘⟨ rightId ⟩
        A.op ⨟ Id
    ≈˘⟨ ⨟-cong₂ 𝓕.mor-Id ⟩
        A.op ⨟ 𝓕.mor Id
    □
  }
  where
    module A = Coalgebra A


CACategory : Category (j ⊎ k) k Coalgebra
CACategory = retract²Category 𝒞
  (λ {A} → CA-Id {A})
  comp
  Carrier
  mor
  ≈-refl
  ≈-refl
  where open Coalgebra; open CAMor
```

3.3 Categoric.Coalgebra.Monadic

```
open import RATH.Level
open import RATH.PropositionalEquality using (_≡_; ≡-refl)
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
open import Categoric.Semigroupoid
open import Categoric.Category
open import Categoric.Category.FinColimits
open import Categoric.Functor
open import Categoric.Monad
open import Function using (_∘_)


module Categoric.Coalgebra.Monadic
  {i j k : Level} {Obj : Set i} {𝒞₀ : Category {i} j k Obj}
  (ℳ : Monad 𝒞₀)
  (ℱ : Functor 𝒞₀ 𝒞₀)
  where
open Category₀ 𝒞₀
open CatFinColimits 𝒞₀
private
  module 𝒞₀ = Category 𝒞₀
  module ℳ = Monad' ℳ
  module ℱ = Functor ℱ
  𝒦 = ℳ.KleisliCat
  module 𝒦 = Category 𝒦
  ℱℳ : Functor 𝒞₀ 𝒞₀
  ℱℳ = ℱ ⨟⨟ ℳ.M
  module ℱℳ = Functor ℱℳ
  ℳℱ : Functor 𝒞₀ 𝒞₀
```

```
  𝓜𝓕 = 𝓜.M ⨟⨟ 𝓕
  module 𝓜𝓕 = Functor 𝓜𝓕
open Category₁ 𝒦
open 𝓜 using (↼; _⨟∘_)
```

In order to be able to consider monadic coalgebras to be coalgebras over the Kleisli category $\mathcal{K}$, we need to derive a functor on $\mathcal{K}$ from $\mathcal{F}$.

Two seemingly plausible choices for this functor are starting from $\mathcal{F}$ or from $\mathcal{FM}$, see Categoric.Coalgebra.Monadic.GfromF (Sect. 3.4) and Categoric.Coalgebra.Monadic.GfromFM (Sect. 3.5).

Only the third choice, namely to start from $\mathcal{MF}$, is compatible with example applications, and also does not require additional hypotheses.

```
module MonCoAlg𝓜𝓕
  where
```

Here, we can actually define the natural transformation required for adapting $\mathcal{G}$.mor:

```
𝓜𝓕distr𝓜 : NatTrans (𝓜.M ⨟⨟ 𝓜𝓕) (𝓜𝓕 ⨟⨟ 𝓜.M)
𝓜𝓕distr𝓜 = NatIso.from (NatAssoc {F = 𝓜.M} {𝓜.M} {𝓕})
             ; 𝓜.M-join ◃⨟⨟ 𝓕
             ; NatIso.from (NatRightId {F = 𝓜𝓕})
             ; 𝓜𝓕 ⨟⨟▹ 𝓜.M-return
open NatTrans 𝓜𝓕distr𝓜 using () renaming (indmor to distr𝓜𝓕; naturality to distr𝓜𝓕-naturality)
```

We will use the following properties:

```
  distr𝓜𝓕-def : {A : Obj} → distr𝓜𝓕 {A} ≈₀ 𝓕.mor 𝓜.join ⨟₀ 𝓜.return
  distr𝓜𝓕-def {A} = leftId₀ ⟨≈₀≈⟩ 𝒞₀.⨟-cong₂ leftId₀
  join-distr𝓜𝓕 : {A : Obj} → 𝓜𝓕.mor (𝓜.join {A}) ⨟₀ distr𝓜𝓕 ≈₀ distr𝓜𝓕 ⨟∘ distr𝓜𝓕
  join-distr𝓜𝓕 {A} = ≈₀-begin
      𝓜𝓕.mor (𝓜.join {A}) ⨟₀ distr𝓜𝓕
    ≈₀⟨ 𝒞₀.⨟-cong₂ distr𝓜𝓕-def ⟩
      𝓜𝓕.mor (𝓜.join {A}) ⨟₀ 𝓕.mor 𝓜.join ⨟₀ 𝓜.return
    ≈₀⟨ 𝒞₀.⨟-cong₁&₂₁ (𝓕.mor-⨟ ⟨≈₀˘≈⟩ (𝓕.mor-cong 𝓜.assoc ⟨≈₀˘≈⟩ 𝓕.mor-⨟)) ⟩
      𝓕.mor 𝓜.join ⨟₀ 𝓕.mor 𝓜.join ⨟₀ 𝓜.return
    ≈₀˘⟨ 𝒞₀.⨟-cong₂ distr𝓜𝓕-def ⟩
      𝓕.mor 𝓜.join ⨟₀ distr𝓜𝓕
    ≈₀˘⟨ 𝒞₀.⨟-cong₂ (𝒞₀.⨟-cong₂ 𝓜.leftUnit ⟨≈₀≈⟩ rightId₀) ⟩
      𝓕.mor 𝓜.join ⨟₀ distr𝓜𝓕 ⨟₀ 𝓜.return ⨟₀ 𝓜.join
    ≈₀⟨ 𝒞₀.⨟-cong₂ (𝒞₀.⨟-cong₁&₂₁ 𝓜.return-naturality) ⟩
      𝓕.mor 𝓜.join ⨟₀ 𝓜.return ⨟₀ 𝓜.mor distr𝓜𝓕 ⨟₀ 𝓜.join
    ≈₀˘⟨ 𝒞₀.⨟-cong₁ distr𝓜𝓕-def ⟨≈₀≈⟩ 𝒞₀.⨟-assoc ⟩
      distr𝓜𝓕 ⨟₁ distr𝓜𝓕
    □₀
  distr𝓜𝓕-join : {A : Obj} → 𝓜.mor distr𝓜𝓕 ⨟₀ 𝓜.join {𝓕.obj (𝓜.obj A)} ≈₀ 𝓜.mor (𝓕.mor 𝓜.join)
  distr𝓜𝓕-join {A} = ≈₀-begin
      𝓜.mor distr𝓜𝓕 ⨟₀ 𝓜.join
    ≈₀⟨ 𝒞₀.⨟-cong₁ (𝓜.mor-cong distr𝓜𝓕-def ⟨≈₀≈⟩ 𝓜.mor-⨟) ⟨≈₀≈⟩ 𝒞₀.⨟-assoc ⟩
      𝓜.mor (𝓕.mor 𝓜.join) ⨟₀ 𝓜.mor 𝓜.return ⨟₀ 𝓜.join
    ≈₀⟨ 𝒞₀.⨟-cong₂ 𝓜.rightUnit ⟨≈₀≈⟩ 𝒞₀.rightId ⟩
      𝓜.mor (𝓕.mor 𝓜.join)
    □₀
  return-distr𝓜𝓕 : {A : Obj} → 𝓜𝓕.mor (𝓜.return {A}) ⨟₀ distr𝓜𝓕 ≈₀ 𝓜.return
  return-distr𝓜𝓕 {A} = ≈₀-begin
      𝓜𝓕.mor (𝓜.return {A}) ⨟₀ distr𝓜𝓕
```

```
    ≈₀⟨ 𝒞₀.⨟-cong₂ distr𝓜𝓕-def ⟩
      𝓜𝓕.mor (𝓜.return {A}) ⨟₀ ℱ.mor 𝓜.join ⨟₀ 𝓜.return
    ≈₀⟨ 𝒞₀.⨟-assocL ⟨≈₀≈⟩ 𝒞₀.⨟-cong₁ (ℱ.mor-⨟ ⟨≈₀˘≈⟩ ℱ.mor-cong 𝓜.rightUnit ⟨≈₀≈⟩ ℱ.mor-Id) ⟨≈₀≈⟩ leftId₀ ⟩
      𝓜.return
    □₀
```

For the functor underlying Data.SEAGraph, candidates for a right-inverse of $\mathcal{MF}$distr$\mathcal{M}$ would require various left-inverse properties for $\mathcal{T}$.return, see after Categoric.MonCoAlg2.SEAG.Pushout.$\mathcal{MF}$distr$\mathcal{M}$.

```
  𝒢 : Functor 𝒦 𝒦
  𝒢 = record
    {obj = 𝓜𝓕.obj
    ;mor = λ {A} {B} f → 𝓜𝓕.mor f ⨟₀ distr𝓜𝓕
    ;mor-cong = λ {A} {B} {f} {g} f≈g → 𝒞₀.⨟-cong₁ (𝓜𝓕.mor-cong f≈g)
    ;mor-⨟ = λ {A} {B} {C} {f} {g} → ≈₀-begin
        𝓜𝓕.mor (f ⨟₀ 𝓜.mor g ⨟₀ 𝓜.join) ⨟₀ distr𝓜𝓕
      ≈₀⟨ 𝒞₀.⨟-cong₁ 𝓜𝓕.mor-⨟⨟ ⟨≈₀≈⟩ 𝒞₀.⨟-assoc₃₊₁ ⟩
        𝓜𝓕.mor f ⨟₀ 𝓜𝓕.mor (𝓜.mor g) ⨟₀ 𝓜𝓕.mor 𝓜.join ⨟₀ distr𝓜𝓕
      ≈₀⟨ 𝒞₀.⨟-cong₂₂ join-distr𝓜𝓕 ⟩
        𝓜𝓕.mor f ⨟₀ 𝓜𝓕.mor (𝓜.mor g) ⨟₀ (distr𝓜𝓕 ⨟∘ distr𝓜𝓕)
      ≈₀⟨ 𝒞₀.⨟-cong₂ (𝒞₀.⨟-cong₁&₂₁ distr𝓜𝓕-naturality) ⟩
        𝓜𝓕.mor f ⨟₀ distr𝓜𝓕 ⨟₀ 𝓜.mor (𝓜𝓕.mor g) ⨟₀ 𝓜.mor distr𝓜𝓕 ⨟₀ 𝓜.join
      ≈₀˘⟨ 𝒞₀.⨟-cong₂ (𝒞₀.⨟-cong₁ 𝓜.mor-⨟ ⟨≈₀≈⟩ 𝒞₀.⨟-assoc) ⟨≈₀≈⟩ 𝒞₀.⨟-assoc ⟩
        (𝓜𝓕.mor f ⨟₀ distr𝓜𝓕) ⨟₀ 𝓜.mor (𝓜𝓕.mor g ⨟₀ distr𝓜𝓕) ⨟₀ 𝓜.join
      □₀
    ;mor-Id = λ {A} → ≈₀-begin
        𝓜𝓕.mor (𝓜.return {A}) ⨟₀ distr𝓜𝓕 {A}
      ≈₀⟨ return-distr𝓜𝓕 ⟩
        𝓜.return {𝓜𝓕.obj A}
      □₀
    }
  module 𝒢 = Functor 𝒢
  open import Categoric.Coalgebra.Category {i} {j} {k} {Obj} {𝒦} 𝒢
    public using () renaming
      (Coalgebra to MonCoAlg
      ;module Coalgebra to MonCoAlg
      ;CAMor to MCAMor
      ;module CAMor to MCAMor
      ;CACategory to MCACategory
      )
```

Note that A.op now has the type $\mathsf{Mor}_0\ \mathsf{A}_0\ (\mathcal{M}.\mathsf{obj}\ (\mathcal{F}.\mathsf{obj}\ (\mathcal{M}.\mathsf{obj}\ \mathsf{A}_0)))$.

```
  module _ {A B : MonCoAlg} where
    open module A = MonCoAlg A using () renaming (Carrier to A₀)
    open module B = MonCoAlg B using () renaming (Carrier to B₀)
    ↼𝒢mor : {f : Mor₁ A₀ B₀} → ↼ (𝒢.mor f) ≈₀ ℱ𝓜.mor (↼ f)
    ↼𝒢mor {f} = ≈₀-begin
        𝓜.mor (𝓜𝓕.mor f ⨟₀ distr𝓜𝓕) ⨟₀ 𝓜.join
      ≈₀⟨ 𝒞₀.⨟-cong₁ 𝓜.mor-⨟ ⟨≈₀≈⟩ 𝒞₀.⨟-assoc ⟨≈₀≈⟩ 𝒞₀.⨟-cong₂ distr𝓜𝓕-join ⟩
        𝓜.mor (𝓜𝓕.mor f) ⨟₀ 𝓜.mor (ℱ.mor 𝓜.join)
      ≈₀˘⟨ ℱ𝓜.mor-⨟ ⟩
        ℱ𝓜.mor (𝓜.mor f ⨟₀ 𝓜.join)
      □₀
```

3.4 Categoric.Coalgebra.Monadic.GfromF

```
open import RATH.Level
```

```
open import Categoric.Category
open import Categoric.Functor
open import Categoric.Monad
open import Function using (_∘_)


module Categoric.Coalgebra.Monadic.GfromF
  {i j k : Level} {Obj : Set i} {𝒞₀ : Category {i} j k Obj}
  (ℳ : Monad 𝒞₀)
  (ℱ : Functor 𝒞₀ 𝒞₀)
  where
open Category₀ 𝒞₀
private
  module 𝒞₀ = Category 𝒞₀
  module ℳ = Monad′ ℳ
  module ℱ = Functor ℱ
  𝒦 = ℳ.KleisliCat
  module 𝒦 = Category 𝒦
  ℱℳ : Functor 𝒞₀ 𝒞₀
  ℱℳ = ℱ ⁘₀ ℳ.M
  module ℱℳ = Functor ℱℳ
  ℳℱ : Functor 𝒞₀ 𝒞₀
  ℳℱ = ℳ.M ⁘₀ ℱ
  module ℳℱ = Functor ℳℱ
open Category₁ 𝒦
open ℳ using (↼; _⁘∘_)
```

In order to be able to consider monadic coalgebras to be coalgebras over the Kleisli category $\mathcal{K}$, we need to derive a functor on $\mathcal{K}$ from $\mathcal{F}$.

One plausible choice is to start from $\mathcal{F}$:

```
module MonCoAlgℱ (ℱdistrℳ : NatTrans ℳℱ ℱℳ) where
  open NatTrans ℱdistrℳ using () renaming (indmor to distr; naturality to distr-naturality)
```

distr cannot be given for the functor underlying Data.SEAGraph, since it would require a morphism mapping terms to variables (Mor_2 (𝒯.obj GV) GV), see Categoric.MonCoAlg2.SEAG.Pushout.$\mathcal{M}$-$\mathcal{F}$-distr. (The opposite type, however, has a natural transformation Categoric.MonCoAlg2.SEAG.Pushout.$\mathcal{M}$-$\mathcal{F}$-undistr.)

```
  𝒢 : Functor 𝒦 𝒦
  𝒢 = record
    {obj = ℱ.obj
    ; mor = λ {A} {B} f → ℱ.mor f ⁘₀ distr
    ; mor-cong = λ {A} {B} {f} {g} f≈g → 𝒞₀.⁘-cong₁ (ℱ.mor-cong f≈g)
    ; mor-⁘ = λ {A} {B} {C} {f} {g} → ≈₀-begin
        ℱ.mor (f ⁘₀ ℳ.mor g ⁘₀ ℳ.join) ⁘₀ distr
      ≈₀⟨ 𝒞₀.⁘-cong₁ ℱ.mor-⁘⁘ ⟨≈₀≈⟩ 𝒞₀.⁘-assoc₃₊₁ ⟩
        ℱ.mor f ⁘₀ ℱ.mor (ℳ.mor g) ⁘₀ ℱ.mor ℳ.join ⁘₀ distr
      ≈₀⟨ 𝒞₀.⁘-cong₂₂ {!!} ⟩   -- ???
        ℱ.mor f ⁘₀ ℱ.mor (ℳ.mor g) ⁘₀ distr ⁘₀ ℳ.mor distr ⁘₀ ℳ.join
      ≈₀⟨ 𝒞₀.⁘-cong₂ (𝒞₀.⁘-cong₁&₂₁ distr-naturality) ⟩
        ℱ.mor f ⁘₀ distr ⁘₀ ℳ.mor (ℱ.mor g) ⁘₀ ℳ.mor distr ⁘₀ ℳ.join
      ≈₀˘⟨ 𝒞₀.⁘-cong₂ (𝒞₀.⁘-cong₁ ℳ.mor-⁘ ⟨≈₀≈⟩ 𝒞₀.⁘-assoc) ⟨≈₀≈⟩ 𝒞₀.⁘-assoc ⟩
        (ℱ.mor f ⁘₀ distr) ⁘₀ ℳ.mor (ℱ.mor g ⁘₀ distr) ⁘₀ ℳ.join
      □₀
    ; mor-Id = λ {A} → ≈₀-begin
        ℱ.mor ℳ.return ⁘₀ distr
```

```
      ≈₀⟨ {!!} ⟩   -- ???
        M.return
      □₀
    }
  open import Categoric.Coalgebra.Category {i} {j} {k} {Obj} {K} G
    public using () renaming
      (Coalgebra to MonCoAlg
      ; module Coalgebra to MonCoAlg
      ; CAMor to MCAMor
      ; module CAMor to MCAMor
      ; CACategory to MCACategory
      )
```

3.5 Categoric.Coalgebra.Monadic.GfromFM

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


module Categoric.Coalgebra.Monadic.GfromFM
  {i j k : Level} {Obj : Set i} {C₀ : Category {i} j k Obj}
  (M : Monad C₀)
  (F : Functor C₀ C₀)
  where
open Category₀ C₀
private
  module C₀ = Category C₀
  module M = Monad' M
  module F = Functor F
  K = M.KleisliCat
  module K = Category K
  FM : Functor C₀ C₀
  FM = F ⁘ M.M
  module FM = Functor FM
  MF : Functor C₀ C₀
  MF = M.M ⁘ F
  module MF = Functor MF
open Category₁ K
open M using (↼; _⁘∘_)
```

In order to be able to consider monadic coalgebras to be coalgebras over the Kleisli category $\mathcal{K}$, we need to derive a functor on $\mathcal{K}$ from $\mathcal{F}$.

One seemingly plausible choice is to start from $\mathcal{FM}$:

```
module MonCoAlgFM (FMdistrM : NatTrans (MF ⁘ M.M) (FM ⁘ M.M)) where
  open NatTrans FMdistrM using () renaming (indmor to distr; naturality to distr-naturality)
```

$\mathcal{FM}$distr$\mathcal{M}$ cannot be given for the functor underlying Data.SEAGraph, since it would require a morphism mapping terms to variables (Mor$_2$ ($\mathcal{T}$.obj GV) GV), see Categoric.MonCoAlg2.SEAG.Pushout.$\mathcal{FM}$distr$\mathcal{M}$.

```
  G : Functor K K
  G = record
```

```
    {obj = 𝓕𝓜.obj
    ;mor = λ {A} {B} f → 𝓕𝓜.mor f ⨟₀ distr
    ;mor-cong = λ {A} {B} {f} {g} f≈g → 𝒞₀.⨟-cong₁ (𝓕𝓜.mor-cong f≈g)
    ;mor-⨟ = λ {A} {B} {C} {f} {g} → ≈₀-begin
        𝓕𝓜.mor (f ⨟₀ 𝓜.mor g ⨟₀ 𝓜.join) ⨟₀ distr
      ≈₀⟨ 𝒞₀.⨟-cong₁ 𝓕𝓜.mor-⨟⨟ ⟨≈₀≈⟩ 𝒞₀.⨟-assoc₃₊₁ ⟩
        𝓕𝓜.mor f ⨟₀ 𝓕𝓜.mor (𝓜.mor g) ⨟₀ 𝓕𝓜.mor 𝓜.join ⨟₀ distr
      ≈₀⟨ 𝒞₀.⨟-cong₂₂ {!!} ⟩   -- ???
        𝓕𝓜.mor f ⨟₀ 𝓕𝓜.mor (𝓜.mor g) ⨟₀ distr ⨟₀ 𝓜.mor distr ⨟₀ 𝓜.join
      ≈₀⟨ 𝒞₀.⨟-cong₂ (𝒞₀.⨟-cong₁&₂₁ distr-naturality) ⟩
        𝓕𝓜.mor f ⨟₀ distr ⨟₀ 𝓜.mor (𝓕𝓜.mor g) ⨟₀ 𝓜.mor distr ⨟₀ 𝓜.join
      ≈₀˘⟨ 𝒞₀.⨟-cong₂ (𝒞₀.⨟-cong₁ 𝓜.mor-⨟ ⟨≈₀≈⟩ 𝒞₀.⨟-assoc) ⟨≈₀≈⟩ 𝒞₀.⨟-assoc ⟩
        (𝓕𝓜.mor f ⨟₀ distr) ⨟₀ 𝓜.mor (𝓕𝓜.mor g ⨟₀ distr) ⨟₀ 𝓜.join
      □₀
    ;mor-Id = λ {A} → ≈₀-begin
        𝓕𝓜.mor 𝓜.return ⨟₀ distr
      ≈₀⟨ {!!} ⟩   -- ???
        𝓜.return
      □₀
    }
  open import Categoric.Coalgebra.Category {i} {j} {k} {Obj} {𝒦} 𝒢
    public using () renaming
      (Coalgebra to MonCoAlg
      ;module Coalgebra to MonCoAlg
      ;CAMor to MCAMor
      ;module CAMor to MCAMor
      ;CACategory to MCACategory
      )
```

3.6 Categoric.Monad.KleisliEndoFunctor

```
open import RATH.Level
open import Categoric.Category
open import Categoric.Functor
open import Categoric.Monad
open import RATH.PropositionalEquality using (_≡_; ≡-refl)
```

Let a monad $\mathcal{M}$ on a category $\mathcal{C}_0$ be given:

```
module Categoric.Monad.KleisliEndoFunctor
  {ℓi₀ ℓj₀ ℓk₀ : Level} {Obj₀ : Set ℓi₀} {𝒞₀ : Category {ℓi₀} ℓj₀ ℓk₀ Obj₀} (ℳ : Monad 𝒞₀)
  where
private
  module 𝒞₀ = Category 𝒞₀
  module ℳ = Monad′ ℳ
  𝒞₁ = ℳ.KleisliCat
  module 𝒞₁ = Category 𝒞₁
open ℳ using (_⨟∘_)
open Category₀ 𝒞₀
open Category₁ 𝒞₁
```

We characterise endofunctors on the Kleisli category $\mathcal{C}_1$ in terms of the base category $\mathcal{C}_0$:

```
record KleisliEndoFunctor : Set (ℓi₀ ⊎ ℓj₀ ⊎ ℓk₀) where
  field
```

```
    obj : Obj₀ → Obj₀
    mor : {A B : Obj₀} → 𝓜.KMor A B → 𝓜.KMor (obj A) (obj B)
    mor-cong : {A B : Obj₀} {f g : 𝓜.KMor A B} → f ≈₀ g → mor f ≈₀ mor g
    mor-⨟ : {A B C : Obj₀} {f : 𝓜.KMor A B} {g : 𝓜.KMor B C}
          → mor (f ⨟∘ g) ≈₀ mor f ⨟∘ mor g
    mor-Id : {A : Obj₀} → mor (𝓜.return {A}) ≈₀ 𝓜.return {obj A}
```

This may be useful if it can serve to avoid explicit construction of $\mathcal{C}_1$ at typechecking time, and possibly also at run-time, but making this argument will require running appropriate benchmarks.

The KleisliEndoFunctor characterisation precisely captures endofunctors on the Kleisli category, as witnessed by the following inverse conversions:

```
toKleisliEndoFunctor : (F : Functor 𝒞₁ 𝒞₁) → KleisliEndoFunctor
toKleisliEndoFunctor F = record {obj = F.obj; mor = F.mor; mor-cong = F.mor-cong; mor-⨟ = F.mor-⨟; mor-Id = F.mor-Id}
  where
    module F = Functor F


fromKleisliEndoFunctor : KleisliEndoFunctor → Functor 𝒞₁ 𝒞₁
fromKleisliEndoFunctor G = record {obj = G.obj; mor = G.mor; mor-cong = G.mor-cong; mor-⨟ = G.mor-⨟; mor-Id = G.mor-Id}
  where
    module G = KleisliEndoFunctor G


from-to-KleisliEndoFunctor : (F : Functor 𝒞₁ 𝒞₁) → fromKleisliEndoFunctor (toKleisliEndoFunctor F) ≡ F
from-to-KleisliEndoFunctor F = ≡-refl


to-from-KleisliEndoFunctor : (G : KleisliEndoFunctor) → toKleisliEndoFunctor (fromKleisliEndoFunctor G) ≡ G
to-from-KleisliEndoFunctor G = ≡-refl
```

More interesting is the questions which endofunctors on $\mathcal{C}_0$ can easily be converted to endofunctors on the Kleisli category, and vice versa.

(SEAGraph cannot even have the indmor of Swap, see undistr in Categoric.MonCoAlg.SEAG.FMNonProps (Sect. 5.5).)

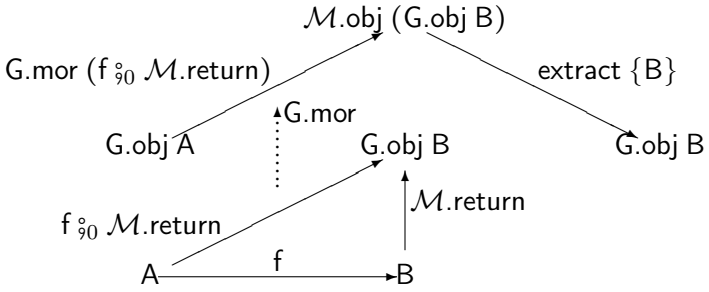
```
module _ (F : Functor 𝒞₀ 𝒞₀) where
  private module F = Functor F

  module _ (Swap : NatTrans (𝓜.M ⨟⨟ F) (F ⨟⨟ 𝓜.M)) where
    open NatTrans Swap renaming (indmor to swap; naturality to swap-naturality)
        -- swap : {A : Obj₀} → Mor₀ (F.obj (𝓜.obj A)) (𝓜.obj (F.obj A))

    liftEndoFunctor : (join-swap : {A : Obj₀} → F.mor 𝓜.join ⨟₀ swap {A} ≈₀ swap ⨟₀ 𝓜.mor swap ⨟₀ 𝓜.join)
      → (return-swap : {A : Obj₀} → F.mor (𝓜.return {A}) ⨟₀ swap {A} ≈₀ 𝓜.return {F.obj A})
      → KleisliEndoFunctor
    liftEndoFunctor join-swap return-swap = record
      {obj = F.obj
      ; mor = λ {A} {B} (f : 𝓜.KMor A B) → F.mor f ⨟₀ swap
      ; mor-cong = λ {A} {B} {f} {g} f≈g → 𝒞₀.⨟-cong₁ (F.mor-cong f≈g)
      ; mor-⨟ = λ {A} {B} {C} {f} {g} → ≈₀-begin
          F.mor (f ⨟∘ g) ⨟₀ swap
        ≈₀⟨ 𝒞₀.⨟-cong₁ F.mor-⨟⨟ ⟨≈₀≈⟩ 𝒞₀.⨟-assoc₃₊₁ ⟩
          F.mor f ⨟₀ F.mor (𝓜.mor g) ⨟₀ F.mor 𝓜.join ⨟₀ swap
        ≈₀⟨ 𝒞₀.⨟-cong₂₂ join-swap ⟩
          F.mor f ⨟₀ F.mor (𝓜.mor g) ⨟₀ swap ⨟₀ 𝓜.mor swap ⨟₀ 𝓜.join
        ≈₀⟨ 𝒞₀.⨟-cong₂ (𝒞₀.⨟-assocL ⟨≈₀≈⟩ 𝒞₀.⨟-cong₁ swap-naturality) ⟩
          F.mor f ⨟₀ (swap ⨟₀ 𝓜.mor (F.mor g)) ⨟₀ 𝓜.mor swap ⨟₀ 𝓜.join
        ≈₀˘⟨ 𝒞₀.⨟-cong₂ (𝒞₀.⨟-cong₁ 𝓜.mor-⨟ ⟨≈₀≈⟩ 𝒞₀.⨟-assoc) ⟨≈₀≈⟩ 𝒞₀.⨟-₂₂assoc₁₂₁ ⟩
          (F.mor f ⨟₀ swap) ⨟∘ (F.mor g ⨟₀ swap)
        □₀
```

```
  ; mor-Id = λ {A} → ≈₀-begin
      F.mor 𝓜.return ⨟₀ swap
    ≈₀⟨ return-swap ⟩
      𝓜.return
    □₀
  }
```

For constructing an endofunctor on $\mathcal{C}_0$ from an endofunctor G on the Kleisli category of the monad $\mathcal{M}$ on $\mathcal{C}_0$, we need to adapt both the argument and the result of G.mor to be able to construct the morphism component:



Since G is an endofunctor on the Kleisli category, and we need extract morphisms that are not morphisms of the Kleisli category, extract can be a natural transformation neither on $\mathcal{C}_0$ nor on $\mathcal{C}_1$.

```
module _ (G : KleisliEndoFunctor) where
  private module G = KleisliEndoFunctor G
  module _ (extract : {A : Obj₀} → Mor₀ (𝓜.obj (G.obj A)) (G.obj A))
           (extract-commute : {A B : Obj₀} {f : 𝓜.KMor A B}
                              → 𝓜.mor (G.mor f ⨟₀ extract {B})
                              ≈₀ extract {A} ⨟₀ G.mor f)
           (return-extract : {A : Obj₀} → G.mor 𝓜.return ⨟₀ extract {A} ≈₀ Id₀ {G.obj A})
           (join-extract : {A : Obj₀} → 𝓜.mor extract ⨟₀ extract {A} ≈₀ 𝓜.join ⨟₀ extract)
    where
    lowerKleisliEndoFunctor : Functor 𝒞₀ 𝒞₀
    lowerKleisliEndoFunctor = record
      {obj = G.obj
      ; mor = λ {A} {B} f → G.mor (f ⨟₀ 𝓜.return) ⨟₀ extract
      ; mor-cong = λ {A} {B} {f} {g} f≈g → 𝒞₀.⨟-cong₁ (G.mor-cong (𝒞₀.⨟-cong₁ f≈g))
      ; mor-⨟ = λ {A} {B} {C} {f} {g} → ≈₀-begin
          G.mor ((f ⨟₀ g) ⨟₀ 𝓜.return) ⨟₀ extract
        ≈₀˘⟨ 𝒞₀.⨟-cong₁ (G.mor-cong (𝓜.⨟-⨟∘-assoc ⟨≈₀≈⟩ 𝒞₀.⨟-cong₂ 𝓜.return-⨟∘ ⟨≈₀≈⟩ 𝒞₀.⨟-assocL)) ⟩
          G.mor ((f ⨟₀ 𝓜.return) ⨟₁ (g ⨟₀ 𝓜.return)) ⨟₀ extract
        ≈₀⟨ 𝒞₀.⨟-cong₁ G.mor-⨟ ⟩
          (G.mor (f ⨟₀ 𝓜.return) ⨟₁ G.mor (g ⨟₀ 𝓜.return)) ⨟₀ extract
        ≈₀⟨ 𝒞₀.⨟-assoc ⟨≈₀≈⟩ 𝒞₀.⨟-cong₂ 𝒞₀.⨟-assoc ⟩
          G.mor (f ⨟₀ 𝓜.return) ⨟₀ 𝓜.mor (G.mor (g ⨟₀ 𝓜.return)) ⨟₀ 𝓜.join ⨟₀ extract
        ≈₀⟨ 𝒞₀.⨟-cong₂ (𝒞₀.⨟-cong₂ join-extract ⟨≈₀˘≈⟩ 𝒞₀.⨟-assocL ⟨≈₀≈˘⟩ 𝒞₀.⨟-cong₁ 𝓜.mor-⨟) ⟩
          G.mor (f ⨟₀ 𝓜.return) ⨟₀ 𝓜.mor (G.mor (g ⨟₀ 𝓜.return) ⨟₀ extract) ⨟₀ extract
        ≈₀⟨ 𝒞₀.⨟-cong₂₁ extract-commute ⟩
          G.mor (f ⨟₀ 𝓜.return) ⨟₀ (extract ⨟₀ G.mor (g ⨟₀ 𝓜.return)) ⨟₀ extract
        ≈₀⟨ 𝒞₀.⨟-₁₂₁assoc₂₂ ⟩
          (G.mor (f ⨟₀ 𝓜.return) ⨟₀ extract) ⨟₀ G.mor (g ⨟₀ 𝓜.return) ⨟₀ extract
        □₀
      ; mor-Id = λ {A} → ≈₀-begin
          G.mor (Id₀ ⨟₀ 𝓜.return) ⨟₀ extract
        ≈₀⟨ 𝒞₀.⨟-cong₁ (G.mor-cong leftId₀) ⟩
          G.mor 𝓜.return ⨟₀ extract
        ≈₀⟨ return-extract ⟩
          Id₀
```

$\square_0$
}

Chapter 4

Monadic Co-Algebras

4.1 Categoric.MonCoAlg.Obj

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


module Categoric.MonCoAlg.Obj
  {i j k : Level} {Obj : Set i}
  (𝒞 : Category {i} j k Obj)
  (ℳ : Monad 𝒞)
  (ℱ : Functor 𝒞 𝒞)
  where
open Category 𝒞
private
  module ℳ = Monad′ ℳ
  module ℱ = Functor ℱ


record MonCoAlg : Set (i ⊎ j) where
  constructor MCA
  field Carrier : Obj
        op : Mor Carrier (ℱ.obj (ℳ.obj Carrier))
```

4.2 Categoric.MonCoAlg.Cat2

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import Categoric.IdOp
open import Categoric.Category
open import Categoric.Functor
open import Categoric.Monad


module Categoric.MonCoAlg.Cat2
  {i j k : Level} {Obj : Set i} (𝒞 : Category {i} j k Obj)
  (open Category 𝒞)
  (ℳ : Monad 𝒞)
```

```
  (let module 𝓜 = Monad′ 𝓜)
  (ℱ : Functor 𝒞 𝒞)
  (let module ℱ = Functor ℱ)
  (𝓜ℱdistrJoin : NatTrans (𝓜.M ⁘ ℱ ⁘ 𝓜.M) (𝓜.M ⁘ ℱ))
  (open NatTrans 𝓜ℱdistrJoin using () renaming
    (indmor to distrJoin; naturality to distrJoin-naturality)
  )
  (return-distrJoin : {A : Obj} → 𝓜.return ⨟ distrJoin {A} ≈ Id)
  (join-distrJoin : {A : Obj} → 𝓜.join ⨟ distrJoin {A} ≈ 𝓜.mor distrJoin ⨟ distrJoin)
  (join↗distrJoin : {A : Obj} → 𝓜.mor (ℱ.mor 𝓜.join) ⨟ distrJoin {A} ≈ distrJoin ⨟ ℱ.mor 𝓜.join)
  where
open import Categoric.MonCoAlg.Obj 𝒞 𝓜 ℱ
private
  ℱ𝓜 = ℱ ⁘ 𝓜.M
  module ℱ𝓜 = Functor ℱ𝓜
open 𝓜 using (↼; _⨟∘_)


↼-⨟-distrJoin : {A B : Obj} {f : Mor A (𝓜.obj (ℱ.obj (𝓜.obj B)))}
              → ↼ f ⨟ distrJoin {B} ≈ 𝓜.mor (f ⨟ distrJoin) ⨟ distrJoin
↼-⨟-distrJoin {A} {B} {f} = ≈-begin
    (𝓜.mor f ⨟ 𝓜.join) ⨟ distrJoin
  ≈⟨ ⨟-cong₁₂&₂ join-distrJoin ⟨≈≈˘⟩ ⨟-cong₁ 𝓜.mor-⨟ ⟩
    𝓜.mor (f ⨟ distrJoin) ⨟ distrJoin
  □


record MCAMor (A B : MonCoAlg) : Set (j ⊍ k) where
  constructor MkMCAMor
  private
    open module A = MonCoAlg A using () renaming (Carrier to A₀)
    open module B = MonCoAlg B using () renaming (Carrier to B₀)
  field mor : 𝓜.KMor A₀ B₀
  mor′ : Mor (𝓜.obj A₀) (𝓜.obj B₀)
  mor′ = ↼ mor
  field commutes : mor ⨟ 𝓜.mor B.op ⨟ distrJoin ≈ A.op ⨟ ℱ.mor mor′
  commutes′ : mor′ ⨟ 𝓜.mor B.op ⨟ distrJoin ≈ (𝓜.mor A.op ⨟ distrJoin) ⨟ ℱ.mor mor′
  commutes′ = ≈-begin
      (𝓜.mor mor ⨟ 𝓜.join) ⨟ 𝓜.mor B.op ⨟ distrJoin
    ≈⟨ ⨟-assoc ⟨≈≈˘⟩ ⨟-cong₂ (⨟-cong₁&₂₁ 𝓜.join-naturality) ⟩
      𝓜.mor mor ⨟ 𝓜.mor (𝓜.mor B.op) ⨟ 𝓜.join ⨟ distrJoin
    ≈⟨ ⨟-cong₂ (⨟-assocL ⟨≈≈⟩ ↼-⨟-distrJoin) ⟩
      𝓜.mor mor ⨟ 𝓜.mor (𝓜.mor B.op ⨟ distrJoin) ⨟ distrJoin
    ≈⟨ ⨟-assocL ⟨≈≈⟩ ⨟-cong₁ (𝓜.mor-⨟ ⟨≈˘≈⟩ 𝓜.mor-cong commutes) ⟩
      𝓜.mor (A.op ⨟ ℱ.mor (𝓜.mor mor ⨟ 𝓜.join)) ⨟ distrJoin
    ≈⟨ ⨟-cong₁ 𝓜.mor-⨟ ⟨≈≈⟩ ⨟-assoc ⟩
      𝓜.mor A.op ⨟ 𝓜.mor (ℱ.mor (𝓜.mor mor ⨟ 𝓜.join)) ⨟ distrJoin
    ≈⟨ ⨟-cong₂ (⨟-cong₁ ℱ𝓜.mor-⨟ ⟨≈≈⟩ ⨟-assoc ⟨≈≈⟩ ⨟-cong₂ join↗distrJoin) ⟩
      𝓜.mor A.op ⨟ 𝓜.mor (ℱ.mor (𝓜.mor mor)) ⨟ distrJoin ⨟ ℱ.mor 𝓜.join
    ≈⟨ ⨟-cong₂ (⨟-cong₁&₂₁ distrJoin-naturality) ⟨≈≈⟩ ⨟-assocL ⟩
      (𝓜.mor A.op ⨟ distrJoin) ⨟ ℱ.mor (𝓜.mor mor) ⨟ ℱ.mor 𝓜.join
    ≈⟨ ⨟-cong₂ (≈-sym ℱ.mor-⨟) ⟩
      (𝓜.mor A.op ⨟ distrJoin) ⨟ ℱ.mor (𝓜.mor mor ⨟ 𝓜.join)
    □
open MCAMor


module MCAMor-Comp {A B C : MonCoAlg} (F : MCAMor A B) (G : MCAMor B C) where
  module A = MonCoAlg A
  module B = MonCoAlg B
```

```
  module C = MonCoAlg C
  module F = MCAMor F
  module G = MCAMor G
  H = F.mor ⨟∘ G.mor


  comp : MCAMor A C
  comp = record
    {mor = F.mor 𝓜.⨟∘ G.mor
    ;commutes = ≈-begin
        (F.mor ⨟ G.mor′) ⨟ 𝓜.mor C.op ⨟ distrJoin
      ≈⟨ ⨟-cong₁₂&₂ G.commutes′ ⟩
        (F.mor ⨟ 𝓜.mor B.op ⨟ distrJoin) ⨟ ℱ.mor G.mor′
      ≈⟨ ⨟-cong₁ F.commutes ⟨≈≈⟩ ⨟-assoc ⟩
        A.op ⨟ ℱ.mor F.mor′ ⨟ ℱ.mor G.mor′
      ≈˘⟨ ⨟-cong₂ ℱ.mor-⨟ ⟩
        A.op ⨟ ℱ.mor (↼ F.mor ⨟ ↼ G.mor)
      ≈⟨ ⨟-cong₂ (ℱ.mor-cong 𝓜.↼-⨟-↼) ⟩
        A.op ⨟ ℱ.mor (↼ (F.mor 𝓜.⨟∘ G.mor))
      □
    }
open MCAMor-Comp public using (comp)


MCA-Id : {A : MonCoAlg} → MCAMor A A
MCA-Id {A} = record
  {mor = 𝓜.return
  ;commutes = ≈-begin
      𝓜.return ⨟ 𝓜.mor (op A) ⨟ distrJoin
    ≈˘⟨ ⨟-cong₁&₂₁ 𝓜.return-naturality ⟩
      op A ⨟ 𝓜.return ⨟ distrJoin
    ≈⟨ ⨟-cong₂ (return-distrJoin ⟨≈≈⟩ (ℱ.mor-Id ⟨≈˘≈˘⟩ ℱ.mor-cong 𝓜.↼return)) ⟩
      op A ⨟ ℱ.mor (↼ 𝓜.return)
    □
  }
  where open MonCoAlg


MCACategory : Category (j ⊍ k) k MonCoAlg
MCACategory = retract²Category 𝓜.KleisliCat
  (λ {A} → MCA-Id {A})   -- missing A.op without eta-expansion
  comp
  MonCoAlg.Carrier
  mor
  ≈-refl
  ≈-refl
```

Chapter 5

Symbolically Edge-Attributed Graphs as Monadic Co-Algebras

Consider edge-attributed graphs, with symbolic attributes taken from the term algebra over some term signature Σ and with variables from the variable carrier set for sort V:

$$\begin{array}{lll} \mathsf{sigAG}_\Sigma := \langle & \textbf{sorts:} & \mathsf{N}, \mathsf{E}, \mathsf{V} \\ & \textbf{ops:} & \mathsf{src} : \mathsf{E} \to \mathsf{N} \\ & & \mathsf{trg} : \mathsf{E} \to \mathsf{N} \\ & & \mathsf{attr} : \mathsf{E} \to \mathcal{T}_\Sigma\ \mathsf{V} \quad \rangle \end{array}$$

Definition 5.0.1 We define the category AG_Σ to have sigAG_Σ-coalgebras as objects, and a morphism $F : G_1 \to G_2$ consists of three mappings typed as shown to the left, satisfying the conditions shown to the right:

$$\begin{array}{lcl} F_\mathsf{N} & : & \mathsf{N}_1 \to \mathsf{N}_2 \\ F_\mathsf{E} & : & \mathsf{E}_1 \to \mathsf{E}_2 \\ F_\mathsf{V} & : & \mathsf{V}_1 \to \mathcal{T}_\Sigma\ \mathsf{V}_2 \end{array} \qquad \begin{array}{lcl} F_\mathsf{E} \fatsemi \mathsf{src}_2 & = & \mathsf{src}_1 \fatsemi F_\mathsf{N} \\ F_\mathsf{E} \fatsemi \mathsf{trg}_2 & = & \mathsf{trg}_1 \fatsemi F_\mathsf{N} \\ F_\mathsf{E} \fatsemi \mathsf{attr}_2 & = & \mathsf{attr}_1 \fatsemi \mathcal{T}_\Sigma\ F_\mathsf{V} \fatsemi \mu_{\mathcal{T}_\Sigma} \end{array}$$

where $\mu_{\mathcal{T}_\Sigma} : \forall X\ .\ \mathcal{T}_\Sigma(\mathcal{T}_\Sigma\ X) \to \mathcal{T}_\Sigma\ X$ is the canonical "term flattening" function that turns two-level nested terms into one-level terms. □

In Data.SEAGraph (Sect. 5.1), we define the category of such symbolically edge-attributed graphs directly, parameterised over a category of "sets". We then show how this is an instance of the monadic coalgebra definition of Chapter 4, with the equivalence of categories formally established in Categoric.MonCoAlg.SEAG.Cat2.Equiv (Sect. 5.15). (One part of this, Categoric.MonCoAlg.SEAG.Cat2.FromTo (Sect. 5.11), has been split into the three modules included after it because it requires 13.5 GB of heap to typecheck on my installation.)

We also provide a direct proof that this for this instance category, pushouts can be obtained from pushouts in the Kleisli category of the underlying monad, in Categoric.MonCoAlg.SEAG.Cat2.Pushout3 (Sect. 5.16).

5.1 Data.SEAGraph

```
open import RATH.Level

open import Categoric.Category
open import Categoric.Monad
open import Categoric.LESGraph


module Data.SEAGraph {ℓi ℓj ℓk : Level} {Obj : Set ℓi} (𝒞 : Category ℓj ℓk Obj) (𝒯 : Monad 𝒞)
  where
```

```
open Category 𝒞
private
  module 𝒯 = Monad′ 𝒯
open 𝒯 using (_⨟∘_)


record SEAGraph : Set (ℓi ⊔ ℓj) where
  field Node : Obj
        Edge : Obj
        Var  : Obj
        src  : Mor Edge Node
        trg  : Mor Edge Node
        attr : Mor Edge (𝒯.obj Var)


record SEAGraphMor (G₁ G₂ : SEAGraph) : Set (ℓj ⊔ ℓk) where
  private
    module G₁ = SEAGraph G₁
    module G₂ = SEAGraph G₂
  field f-N : Mor G₁.Node G₂.Node
        f-E : Mor G₁.Edge G₂.Edge
        f-V : Mor G₁.Var (𝒯.obj G₂.Var)
  field prop-src  : f-E ⨟ G₂.src  ≈ G₁.src ⨟ f-N
        prop-trg  : f-E ⨟ G₂.trg  ≈ G₁.trg ⨟ f-N
        prop-attr : f-E ⨟ G₂.attr ≈ G₁.attr ⨟∘ f-V


record _≋_ {G₁ G₂ : SEAGraph} (Φ Ψ : SEAGraphMor G₁ G₂) : Set ℓk where
  private
    module Φ = SEAGraphMor Φ
    module Ψ = SEAGraphMor Ψ
  field f-N≋ : Φ.f-N ≈ Ψ.f-N
        f-E≋ : Φ.f-E ≈ Ψ.f-E
        f-V≋ : Φ.f-V ≈ Ψ.f-V


SEAGraphHom : LocalSetoid SEAGraph (ℓj ⊔ ℓk) ℓk
SEAGraphHom G₁ G₂ = record
  {Carrier = SEAGraphMor G₁ G₂
  ; _≈_ = _≋_
  ; isEquivalence = record
    {refl = record {f-N≋ = ≈-refl; f-E≋ = ≈-refl; f-V≋ = ≈-refl}
    ; sym = λ Φ≋Ψ → let open _≋_ Φ≋Ψ in record
      {f-N≋ = ≈-sym f-N≋; f-E≋ = ≈-sym f-E≋; f-V≋ = ≈-sym f-V≋}
    ; trans = λ Φ≋X X≋Ψ → let open _≋_ in record
      {f-N≋ = ≈-trans (f-N≋ Φ≋X) (f-N≋ X≋Ψ)
      ; f-E≋ = ≈-trans (f-E≋ Φ≋X) (f-E≋ X≋Ψ)
      ; f-V≋ = ≈-trans (f-V≋ Φ≋X) (f-V≋ X≋Ψ)
      }
    }
  }
  where
    module G₁ = SEAGraph G₁
    module G₂ = SEAGraph G₂


comp : {G₁ G₂ G₃ : SEAGraph}
     → SEAGraphMor G₁ G₂ → SEAGraphMor G₂ G₃ → SEAGraphMor G₁ G₃
comp {G₁} {G₂} {G₃} F G = record
  {f-N = F.f-N ⨟ G.f-N
```

```
  ;f-E = F.f-E ⨟ G.f-E
  ;f-V = F.f-V ⨟∘ G.f-V
  ;prop-src = ≈-begin
      (F.f-E ⨟ G.f-E) ⨟ G₃.src
    ≈⟨ ⨟-assoc ⟨≈≈⟩ ⨟-cong₂ G.prop-src ⟩
      F.f-E ⨟ G₂.src ⨟ G.f-N
    ≈⟨ ⨟-cong₁&₂₁ F.prop-src ⟩
      G₁.src ⨟ F.f-N ⨟ G.f-N
    □
  ;prop-trg = ≈-begin
      (F.f-E ⨟ G.f-E) ⨟ G₃.trg
    ≈⟨ ⨟-assoc ⟨≈≈⟩ ⨟-cong₂ G.prop-trg ⟩
      F.f-E ⨟ G₂.trg ⨟ G.f-N
    ≈⟨ ⨟-cong₁&₂₁ F.prop-trg ⟩
      G₁.trg ⨟ F.f-N ⨟ G.f-N
    □
  ;prop-attr = ≈-begin
      (F.f-E ⨟ G.f-E) ⨟ G₃.attr
    ≈⟨ ⨟-assoc ⟨≈≈⟩ ⨟-cong₂ G.prop-attr ⟩
      F.f-E ⨟ G₂.attr ⨟∘ G.f-V
    ≈⟨ 𝒯.⨟-⨟∘-assocL ⟨≈≈⟩ 𝒯.⨟∘-cong₁ F.prop-attr ⟨≈≈⟩ 𝒯.⨟∘-assoc ⟩
      G₁.attr ⨟∘ F.f-V ⨟∘ G.f-V
    □
  }
  where
    module G₁ = SEAGraph G₁
    module G₂ = SEAGraph G₂
    module G₃ = SEAGraph G₃
    module F = SEAGraphMor F
    module G = SEAGraphMor G

SEAGraphCat : Category (ℓj ⊎ ℓk) ℓk SEAGraph
SEAGraphCat = record
  {semigroupoid = record
    {Hom = SEAGraphHom
    ;compOp = record
      {_⨟_ = comp
      ;⨟-cong = λ Φ₁≋Φ₂ Ψ₁≋Ψ₂ → let open _≋_ in record
        {f-N≋ = ⨟-cong (f-N≋ Φ₁≋Φ₂) (f-N≋ Ψ₁≋Ψ₂)
        ;f-E≋ = ⨟-cong (f-E≋ Φ₁≋Φ₂) (f-E≋ Ψ₁≋Ψ₂)
        ;f-V≋ = 𝒯.⨟∘-cong (f-V≋ Φ₁≋Φ₂) (f-V≋ Ψ₁≋Ψ₂)
        }
      ;⨟-assoc = record {f-N≋ = ⨟-assoc; f-E≋ = ⨟-assoc; f-V≋ = 𝒯.⨟∘-assoc}
      }
    }
  ;idOp = record
    {Id = record
      {f-N = Id
      ;f-E = Id
      ;f-V = 𝒯.return
      ;prop-src = leftId ⟨≈≈˘⟩ rightId
      ;prop-trg = leftId ⟨≈≈˘⟩ rightId
      ;prop-attr = leftId ⟨≈≈˘⟩ 𝒯.⨟∘-return
      }
    ;leftId = record {f-N≋ = leftId; f-E≋ = leftId; f-V≋ = 𝒯.return-⨟∘}
    ;rightId = record {f-N≋ = rightId; f-E≋ = rightId; f-V≋ = 𝒯.⨟∘-return}
    }
  }
```

5.2 Categoric.MonCoAlg.SEAG.FM

Definition of functor and monad to understand symbolically edge-attributed graphs as an instance of Categoric.MonCoAlg.Obj (Sect. 4.1).

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import RATH.PropositionalEquality using (_≡_; ≡-refl)

open import Categoric.Category
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Functor.Product
open import Categoric.Product.Category
open import Categoric.Monad
open import Categoric.Monad.Product

open CatFinLimits


module Categoric.MonCoAlg.SEAG.FM
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (𝒯 : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  where

open Category₂ 𝒞₂
open HasTerminalObject₂ 𝒞₂ hasTerminal₂
open HasProducts₂ 𝒞₂ hasProducts₂

Obj₁ : Set i
Obj₁ = Obj₂ × Obj₂
𝒞₁ : Category j k Obj₁
𝒞₁ = ProductCategory 𝒞₂ 𝒞₂
Obj₀ : Set i
Obj₀ = Obj₁ × Obj₂

𝒞₀ : Category j k Obj₀
𝒞₀ = ProductCategory 𝒞₁ 𝒞₂
module 𝒞₀ = Category 𝒞₀
module 𝒞₁ = Category 𝒞₁
module 𝒞₂ = Category 𝒞₂

⊗₂ : Functor 𝒞₁ 𝒞₂
⊗₂ = biFunctor (ProductBifunctor 𝒞₂ hasProducts₂)

module 𝒯 = Monad′ 𝒯
open Category₀ 𝒞₀
```

We have, conceptually: $\mathcal{F}\ ((\mathsf{N},\mathsf{E}),\mathcal{T}\mathsf{V}) = ((\mathbb{1},(\mathsf{N}\times\mathsf{N}\times\mathcal{T}\mathsf{V})),\mathbb{1})$

```
ℱ : Functor 𝒞₀ 𝒞₀
ℱ =
  (ConstFunctor {Src = 𝒞₀} {Trg = 𝒞₂} 𝟙₂
    ▼
    ((ProdFunctor (Proj₁ 𝒞₂ 𝒞₂ ⁂ (Identity 𝒞₂ ▼ Identity 𝒞₂) ⁂ ⊗₂)
                  (Identity 𝒞₂)
      ) ⁂ ⊗₂))
  ▼ ConstFunctor {Src = 𝒞₀} {Trg = 𝒞₂} 𝟙₂
module ℱ = Functor ℱ


private
  ℱ-welldefined : {N E 𝒯V : Obj₂} → ℱ.obj ((N,E),𝒯V) ≡ ((𝟙₂,((N ⊠₂ N) ⊠₂ 𝒯V)),𝟙₂)
  ℱ-welldefined = ≡-refl
```

We have, conceptually: $\mathcal{M}\ ((\mathsf{N},\mathsf{E}),\mathsf{V}) = ((\mathsf{N},\mathsf{E}),\mathcal{T}\ \mathsf{V})$:

```
𝓜 : Monad 𝒞₀
𝓜 = ProductMonad Id𝓜 𝒯
module 𝓜 = Monad′ 𝓜


private
  𝓜-welldefined : {N E V : Obj₂} → 𝓜.obj ((N, E), V) ≡ ((N, E), 𝒯.obj V)
  𝓜-welldefined = ≡-refl
```

5.3 Categoric.MonCoAlg.SEAG.ObjCompat

Conversion between monadic coalgebras according to Categoric.MonCoAlg.Obj (Sect. 4.1) as instantiated for symbolically edge-attributed graphs in Categoric.MonCoAlg.SEAG.FM (Sect. 5.2) and the direct formalisation of Data.SEAGraph (Sect. 5.1).

```
open import RATH.Level
open import RATH.Data.Product using (_,_)
open import Categoric.Category
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad
open CatFinLimits


module Categoric.MonCoAlg.SEAG.ObjCompat
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (𝒯 : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  where
open import Categoric.MonCoAlg.SEAG.FM 𝒞₂ 𝒯 hasTerminal₂ hasProducts₂
module SEAG-MonCoAlg where
  open import Categoric.MonCoAlg.Obj 𝒞₀ 𝓜 ℱ public
open SEAG-MonCoAlg public
open import Data.SEAGraph 𝒞₂ 𝒯
open Category₀ 𝒞₀
open Category₂ 𝒞₂
open HasTerminalObject₂ 𝒞₂ hasTerminal₂
open HasProducts₂ 𝒞₂ hasProducts₂


objTo : MonCoAlg → SEAGraph
objTo (MCA ((N, E), V) ((_, opE), _)) = record
  {Node = N
  ; Edge = E
  ; Var  = V
  ; src  = opE ⨟₂ π₂ ⨟₂ π₂
  ; trg  = opE ⨟₂ π₂ ⨟₂ ρ₂
  ; attr = opE ⨟₂ ρ₂
  }
  where


objFrom : SEAGraph → MonCoAlg
objFrom G = record
```

```
  {Carrier = (Node, Edge), Var
  ;op      = (𝕋₂ {Node}, (src ▽₂ trg) ▽₂ attr), 𝕋₂ {Var}
  }
  where
    open SEAGraph G


objFromTo : {G : SEAGraph} → Category.Iso SEAGraphCat (objTo (objFrom G)) G
objFromTo {G} = record
  {isoMor = record
      {f-N = Id₂
      ;f-E = Id₂
      ;f-V = 𝒯.return
      ;prop-src = leftId₂ ⟨≈₂≈˘⟩ (rightId₂ ⟨≈₂≈⟩ 𝒫₂.▽▽⨟ππ)
      ;prop-trg = leftId₂ ⟨≈₂≈˘⟩ (rightId₂ ⟨≈₂≈⟩ 𝒫₂.▽▽⨟πρ)
      ;prop-attr = leftId₂ ⟨≈₂≈˘⟩ (𝒞₂.⨟-cong₂ 𝒯.rightUnit ⟨≈₂≈⟩ rightId₂ ⟨≈₂≈⟩ ▽₂⨟ρ₂)
      }
  ;isIso = record
    {_⁻¹ = record
      {f-N = Id₂
      ;f-E = Id₂
      ;f-V = 𝒯.return
      ;prop-src = (leftId₂ ⟨≈₂≈⟩ 𝒫₂.▽▽⨟ππ) ⟨≈₂≈˘⟩ rightId₂
      ;prop-trg = (leftId₂ ⟨≈₂≈⟩ 𝒫₂.▽▽⨟πρ) ⟨≈₂≈˘⟩ rightId₂
      ;prop-attr = (leftId₂ ⟨≈₂≈⟩ ▽₂⨟ρ₂) ⟨≈₂≈˘⟩ (𝒞₂.⨟-cong₂ 𝒯.rightUnit ⟨≈₂≈⟩ rightId₂)
      }
    ;rightInverse = record
      {f-N≋ = leftId₂
      ;f-E≋ = leftId₂
      ;f-V≋ = 𝒞₂.⨟-cong₂ 𝒯.rightUnit ⟨≈₂≈⟩ rightId₂
      }
    ;leftInverse = record
      {f-N≋ = leftId₂
      ;f-E≋ = leftId₂
      ;f-V≋ = 𝒞₂.⨟-cong₂ 𝒯.rightUnit ⟨≈₂≈⟩ rightId₂
      }
    }
  }

objToFrom : {A : MonCoAlg}
          → Category.Iso 𝓜.KleisliCat (MonCoAlg.Carrier (objFrom (objTo A))) (MonCoAlg.Carrier A)
objToFrom {(MCA ((N, E), V) ((_, opE), _))} = record
  {isoMor = (Id₂ {N}, Id₂ {E}), 𝒯.return {V}
  ;isIso = record
    {_⁻¹ = (Id₂ {N}, Id₂ {E}), 𝒯.return {V}
    ;rightInverse = ((leftId₂ ⟨≈₂≈⟩ leftId₂ ⟨≈₂≈⟩ leftId₂)
                    , (leftId₂ ⟨≈₂≈⟩ leftId₂ ⟨≈₂≈⟩ leftId₂)
                    ), (𝒞₂.⨟-cong₂ 𝒯.rightUnit ⟨≈₂≈⟩ rightId₂)
    ;leftInverse =  ((leftId₂ ⟨≈₂≈⟩ leftId₂ ⟨≈₂≈⟩ leftId₂)
                    , (leftId₂ ⟨≈₂≈⟩ leftId₂ ⟨≈₂≈⟩ leftId₂)
                    ), (𝒞₂.⨟-cong₂ 𝒯.rightUnit ⟨≈₂≈⟩ rightId₂)
    }
  }
```

5.4 Categoric.MonCoAlg.SEAG.FMProps

We collect potentially useful properties of $\mathcal{F}$ and $\mathcal{M}$ as defined in Categoric.MonCoAlg.SEAG.FM (Sect. 5.2).

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
open import Categoric.Semigroupoid
open import Categoric.IdOp
open import Categoric.Category
open import Categoric.Category.FinColimits
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Functor.Product
open import Categoric.Product.Category
open import Categoric.Monad
open import Categoric.Monad.Product
open CatFinLimits


module Categoric.MonCoAlg.SEAG.FMProps
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (𝒯 : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  where
open import Categoric.MonCoAlg.SEAG.FM 𝒞₂ 𝒯 hasTerminal₂ hasProducts₂
open Category₀ 𝒞₀
open Category₂ 𝒞₂
open HasTerminalObject₂ 𝒞₂ hasTerminal₂
open HasProducts₂ 𝒞₂ hasProducts₂


distrJoin₀ : {A : Obj₀} → 𝒞₀.Mor (ℳ.obj (ℱ.obj (ℳ.obj A))) (ℱ.obj (ℳ.obj A))
distrJoin₀ {(N₁, E₂), V₁} = (Ⓣ₂ {Ⓣ 2}, Id₂), (Ⓣ₂ {𝒯.obj Ⓣ 2})
distrJoin : NatTrans (ℳ.M ⨟⨟ ℱ ⨟⨟ ℳ.M) (ℳ.M ⨟⨟ ℱ)
distrJoin = record {indmor = λ {A} → distrJoin₀ {A}
  ; naturality = λ {{(N₀, E₀), V₀} {(N₁, E₁), V₁} {(fN, fE), fV}
                 → (Ⓣ 2≈, (𝒞₂.rightId ⟨≈₂≈˘⟩ 𝒞₂.leftId)), Ⓣ 2≈}}
open NatTrans distrJoin public using () renaming (naturality to distrJoin₀-naturality)
return-distrJoin₀ : {A : Obj₀}
                  → ℳ.return {ℱ.obj (ℳ.obj A)} ⨟₀ distrJoin₀ {A}
                  ≈₀ Id₀ {ℱ.obj (ℳ.obj A)}
return-distrJoin₀ {(N, E), V} = (Ⓣ 2≈, rightId₂), Ⓣ 2≈
join-distrJoin₀   : {A : Obj₀}
                  → ℳ.join ⨟₀ distrJoin₀ {A}
                  ≈₀ ℳ.mor (distrJoin₀ {A}) ⨟₀ distrJoin₀ {A}
join-distrJoin₀ {(N, E), V} = (Ⓣ 2≈, rightId₂), Ⓣ 2≈
join↗distrJoin₀   : {A : Obj₀}
                  → ℳ.mor (ℱ.mor (ℳ.join {A})) ⨟₀ distrJoin₀ {A}
                  ≈₀ distrJoin₀ {ℳ.obj A} ⨟₀ ℱ.mor (ℳ.join {A})
join↗distrJoin₀ {(N, E), V} = (Ⓣ 2≈, (𝒞₂.rightId ⟨≈₂≈˘⟩ 𝒞₂.leftId)), Ⓣ 2≈


distr : NatTrans (ℱ ⨟⨟ ℳ.M) (ℳ.M ⨟⨟ ℱ)
distr = record
  {indmor = λ {{(N, E), V} → (Ⓣ₂ {Ⓣ 2}   -- : Mor₂ Ⓣ 2 Ⓣ 2
                             , Id₂ {N ⊠₂ N} ⊗₂ 𝒯.return {V}
                             ), Ⓣ₂ {𝒯.obj Ⓣ 2}   -- : Mor₂ (𝒯.obj Ⓣ 2) Ⓣ 2
                             }
  ; naturality = λ {{(N₀, E₀), V₀} {(N₁, E₁), V₁} {(fN, fE), fV}
                 → (Ⓣ 2≈   -- independent of indMor
                   , (≈₂-begin
```

```
                    ((fN ⊗₂ fN) ⊗₂ fV) ⨟₂ (Id₂ {N₁ ⊠₂ N₁} ⊗₂ 𝒯.return {V₁})
                  ≈₂⟨ 𝒫₂.⨟-⊗-⨟ ⟨≈₂˘≈⟩ 𝒫₂.⊗-cong rightId₂ 𝒯.return-naturality ⟩
                    (fN ⊗₂ fN) ⊗₂ (𝒯.return {V₀} ⨟₂ 𝒯.mor fV)
                  ≈₂˘⟨ 𝒫₂.⨟-⊗-⨟ ⟨≈₂˘≈⟩ 𝒫₂.⊗-cong₁ leftId₂ ⟩
                    (Id₂ {N₀ ⊠₂ N₀} ⊗₂ 𝒯.return {V₀}) ⨟₂ ((fN ⊗₂ fN) ⊗₂ 𝒯.mor fV)
                  □₂)   -- involves only indmorE
                ), 𝕋₂≈   -- independent of indMor
              }
  }
open NatTrans distr public using () renaming
  (indmor to distr₀; naturality to distr-naturality)

undistr″ : NatTrans (ℳ.M ⨾⨾ ℳ.M ⨾⨾ ℱ) (ℳ.M ⨾⨾ ℱ ⨾⨾ ℳ.M)
undistr″ = record
  {indmor = λ {{(N, E), V} → (𝕥₂ {𝕋₂}   -- : Mor₂ 𝕋₂ 𝕋₂
                              , Id₂ {N ⊠₂ N} ⊗₂ 𝒯.join {V}
                              ), 𝒯.return {𝕋₂}   -- : Mor₂ 𝕋₂ (𝒯.obj 𝕋₂)
                              }
  ; naturality = λ {{(N₀, E₀), V₀} {(N₁, E₁), V₁} {(fN, fE), fV}
                → (𝕋₂≈   -- independent of indMor
                  , (≈₂-begin
                        ((fN ⊗₂ fN) ⊗₂ 𝒯.mor (𝒯.mor fV)) ⨟₂ (Id₂ {N₁ ⊠₂ N₁} ⊗₂ 𝒯.join {V₁})
                      ≈₂⟨ 𝒫₂.⨟-⊗-⨟ ⟨≈₂˘≈⟩ 𝒫₂.⊗-cong rightId₂ 𝒯.join-naturality ⟩
                        (fN ⊗₂ fN) ⊗₂ (𝒯.join {V₀} ⨟₂ 𝒯.mor fV)
                      ≈₂˘⟨ 𝒫₂.⨟-⊗-⨟ ⟨≈₂˘≈⟩ 𝒫₂.⊗-cong₁ leftId₂ ⟩
                        (Id₂ {N₀ ⊠₂ N₀} ⊗₂ 𝒯.join {V₀}) ⨟₂ ((fN ⊗₂ fN) ⊗₂ 𝒯.mor fV)
                      □₂)   -- involves only indmorE
                  ), (𝒞₂.leftId
                        ⟨≈₂≈˘⟩ (𝒞₂.⨟-cong₂ 𝒯.mor-Id ⟨≈₂≈⟩ 𝒞₂.rightId))   -- independent of indMor
                }
  }
open NatTrans undistr″ public using () renaming
  (indmor to undistr″₀; naturality to undistr″-naturality)
```

5.5 Categoric.MonCoAlg.SEAG.FMNonProps

For $\mathcal{F}$ and $\mathcal{M}$ as defined in Categoric.MonCoAlg.SEAG.FM (Sect. 5.2), we here collect some properties that one might consider potentially useful, but that do *not* hold.

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
open import Categoric.Semigroupoid
open import Categoric.IdOp
open import Categoric.Category
open import Categoric.Category.FinColimits
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Functor.Product
open import Categoric.Product.Category
open import Categoric.Monad
open import Categoric.Monad.Product
open CatFinLimits

module Categoric.MonCoAlg.SEAG.FMNonProps
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (𝒯 : Monad 𝒞₂)
```

```
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  where
open import Categoric.MonCoAlg.SEAG.FM        𝒞₂ 𝒯 hasTerminal₂ hasProducts₂
open import Categoric.MonCoAlg.SEAG.FMProps 𝒞₂ 𝒯 hasTerminal₂ hasProducts₂
open Category₀ 𝒞₀
open Category₂ 𝒞₂
open HasTerminalObject₂ 𝒞₂ hasTerminal₂
open HasProducts₂ 𝒞₂ hasProducts₂
```

[WK: *The hole in idmorE has type Mor₂ (𝒯.obj V) V:*

```
undistr : NatTrans (𝓜.M ⁘ 𝓕) (𝓕 ⁘ 𝓜.M)
undistr = record
  {indmor = λ {{(N, E), V} → (①₂ {𝕋₂}   -- : Mor₂ 𝕋₂ 𝕋₂
                              , (Id₂ {N} ⊗₂ Id₂ {N}) ⊗₂ {!!}   -- [ ??? ]
                              ), 𝒯.return   -- : Mor₂ 𝕋₂ (𝒯.obj 𝕋₂)
                              }
  ; naturality = λ
    {{(N₀, E₀), V₀} {(N₁, E₁), V₁} {(fN, fE), fV}
    → (𝕋₂≈   -- independent of indmor
      , {!!}   -- [ ??? ] involves only indmorE
      ), (𝒞₂.leftId ⟨≈₂≈˘⟩ (𝒞₂.⁘-cong₂ 𝒯.mor-Id ⟨≈₂≈⟩ 𝒞₂.rightId))   -- independent of indmor
    }
  }
```

]

[WK: *For rV we have the two choices 𝒯.return and 𝒯.mor 𝒯.return.*

The hole in idmorE has type Mor₂ (𝒯.obj V) V:

```
undistr′ : Mor₂ (𝒯.obj 𝕋₂) (𝒯.obj (𝒯.obj 𝕋₂)) → NatTrans (𝓜.M ⁘ 𝓕 ⁘ 𝓜.M) (𝓕 ⁘ 𝓜.M ⁘ 𝓜.M)
undistr′ rV = record
  {indmor = λ {{(N, E), V} → (①₂ {𝕋₂}   -- : Mor₂ 𝕋₂ 𝕋₂
                              , (Id₂ {N} ⊗₂ Id₂ {N}) ⊗₂ {!!}   -- [ ??? ]
                              ), rV   -- : Mor₂ (𝒯.obj 𝕋₂) (𝒯.obj (𝒯.obj 𝕋₂))
                              }
  ; naturality = λ
    {{(N₀, E₀), V₀} {(N₁, E₁), V₁} {(fN, fE), fV}
    → (𝕋₂≈   -- independent of indmor
      , {! !}   -- [ ??? ] involves only indmorE
      ), (𝒞₂.⁘-cong₁ 𝒯.mor-Id ⟨≈₂≈⟩ 𝒞₂.leftId
        ⟨≈₂≈˘⟩ (𝒞₂.⁘-cong₂ (𝒯.mor-cong 𝒯.mor-Id ⟨≈₂≈⟩ 𝒯.mor-Id)
        ⟨≈₂≈⟩ 𝒞₂.rightId))   -- independent of indmor
    }
  }
```

]

For Categoric.MonCoAlg.Pushout: **[WK:** *For rV we have the two choices 𝒯.return and 𝒯.mor 𝒯.return.* **]**

```
RRR : Mor₂ (𝒯.obj 𝕋₂) (𝒯.obj (𝒯.obj 𝕋₂)) → NatTrans (𝓜.M ⁘ 𝓕 ⁘ 𝓜.M) (𝓜.M ⁘ 𝓕 ⁘ 𝓜.M ⁘ 𝓜.M)
RRR rV = record
  {indmor = λ {{(N, E), V} → (①₂ {𝕋₂}   -- : Mor₂ 𝕋₂ 𝕋₂
                              , Id₂ {(N ⊠₂ N) ⊠₂ 𝒯.obj V}
                              ), rV   -- two choices: : Mor₂ (𝒯.obj 𝕋₂) (𝒯.obj (𝒯.obj 𝕋₂))
                              }
```

```
  ; naturality = λ
      {{(N₀, E₀), V₀} {(N₁, E₁), V₁} {(fN, fE), fV}
      → (𝕋₂≈   -- independent of indmor
        , (𝒞₂.rightId ⟨≈₂≈˘⟩ 𝒞₂.leftId)   -- involves only indmorE
        ), (𝒞₂.⨟-cong₁ 𝒯.mor-Id ⟨≈₂≈⟩ 𝒞₂.leftId
            ⟨≈₂≈˘⟩ (𝒞₂.⨟-cong₂ (𝒯.mor-cong 𝒯.mor-Id ⟨≈₂≈⟩ 𝒯.mor-Id)
            ⟨≈₂≈⟩ 𝒞₂.rightId))   -- independent of indmor
      }
  }
module _ (rV : Mor₂ (𝒯.obj 𝕋₂) (𝒯.obj (𝒯.obj 𝕋₂))) where
  open NatTrans (RRR rV) public using () renaming
    (indmor to RRR₀; naturality to RRR-naturality)
```

For Categoric.MonCoAlg.Pushout [**WK:** *Can only be* $\mathcal{M}$*.return*]:

```
SSS : NatTrans (ℳ.M ⨟⨟ ℱ) (ℳ.M ⨟⨟ ℱ ⨟⨟ ℳ.M)
SSS = record
  {indmor = λ {{(N, E), V} → (𝕥₂ {𝕋₂}   -- : Mor₂ 𝕋₂ 𝕋₂
                              , Id₂ {(N ⊠₂ N) ⊠₂ 𝒯.obj V}
                              ), 𝒯.return   -- : Mor₂ 𝕋₂ (𝒯.obj 𝕋₂)
                              }
  ; naturality = λ
      {{(N₀, E₀), V₀} {(N₁, E₁), V₁} {(fN, fE), fV}
      → (𝕋₂≈   -- independent of indmor
        , ((𝒞₂.rightId ⟨≈₂≈˘⟩ 𝒞₂.leftId))   -- involves only indmorE
        ), (𝒞₂.leftId
            ⟨≈₂≈˘⟩ (𝒞₂.⨟-cong₂ 𝒯.mor-Id ⟨≈₂≈⟩ 𝒞₂.rightId))   -- independent of indmor
      }
  }
open NatTrans SSS public using () renaming
  (indmor to SSS₀; naturality to SSS-naturality)
```

For Categoric.MonCoAlg.Pushout:

[**WK:** *The hole has type* $rV \fatsemi_2 \mathcal{T}.join \approx_2 \textcircled{t}_2\ \{\mathcal{T}.obj\ \textcircled{T}_2\} \fatsemi_2 \mathcal{T}.return$*, which won't work no matter whether* rV *is* $\mathcal{T}.return$ *or* $\mathcal{T}.mor\ \mathcal{T}.return$*.*]

```
RRR-SSS : (rV : Mor₂ (𝒯.obj 𝕋₂) (𝒯.obj (𝒯.obj 𝕋₂)))
  → {A : Obj₀} → RRR₀ rV {A} ⨟₀ ℳ.join {ℱ.obj (ℳ.obj A)} ≈₀ distrJoin₀ {A} ⨟₀ SSS₀ {A}
RRR-SSS rV {(N, E), V} = (𝕋₂≈, leftId₂), {!!}   --  ???
```

```
distrJoin-undistrJoin : {A : Obj₀} → distrJoin₀ {A} ⨟₀ ℳ.return ≈₀ Id₀
distrJoin-undistrJoin {(N, E), V} = (𝕋₂≈, {!!}), {!!}   --  ???
```

[**WK:** *The last hole has type* $\textcircled{t}_2 \fatsemi_2 \mathcal{T}.return \approx_2 Id_2\ \{\mathcal{T}.obj\ \textcircled{T}_2\}$*:*

```
distrJoin-↼ : {A₀ A₁ : Obj₀} {f : ℳ.KMor A₀ A₁}
            → distrJoin₀ {A₀} ⨟₀ ℱ.mor (ℳ.↼ f) ⨟₀ ℳ.return
            ≈₀ ℳ.mor (ℱ.mor (ℳ.↼ f))
distrJoin-↼ {(N₀, E₀), V₀} {(N₁, E₁), V₁} {(fN, fE), fV} = (𝕋₂≈, {!!}), {!!}   --  ???
```

]

[**WK:** fV *has type* $Mor_2\ V_0\ \textcircled{T}_2$*, so we have* $fV \approx_2 \textcircled{t}_2$*.*

The last hole has type $\mathcal{T}.mor\ fV \approx_2 \mathcal{T}.mor\ fV \fatsemi_2 \textcircled{t}_2 \fatsemi_2 \mathcal{T}.return$*:*

```
↼undistr''u : {A₀ A₁ A₂ : Obj₀} {f : Mor₀ A₀ (F.obj (M.obj A₁))} {g : M.KMor A₁ A₂}
  → M.↼ (f ⨟₀ F.mor (M.mor g) ⨟₀ undistr''₀ {A₂})
  ≈₀ M.mor f ⨟₀ distrJoin₀ {A₁} ⨟₀ F.mor (M.mor g) ⨟₀ undistr''₀ {A₂}
↼undistr''u {(N₀, E₀), V₀} {(N₁, E₁), V₁} {(N₂, E₂), V₂} {(fN, fE), fV} {(gN, gE), gV}
  = (𝒯₂≈, {!!}), {!!}   -- ???
```

]

[WK: *fV has type Mor₂ V₀ 𝒯₂; the last hole has type T.mor fV ≈₂ T.mor fV ⨟₂ ⓣ₂ ⨟₂ T.return:*

```
↼undistr'' : {A₀ A₁ A₂ : Obj₀} {f : Mor₀ A₀ (F.obj (M.obj A₁))} {g : M.KMor A₁ A₂}
  → M.↼ (f ⨟₀ F.mor (M.mor g) ⨟₀ undistr''₀ {A₂})
  ≈₀ M.mor f ⨟₀ distrJoin₀ {A₁} ⨟₀ F.mor (M.↼ g) ⨟₀ M.return
↼undistr'' {(N₀, E₀), V₀} {(N₁, E₁), V₁} {(N₂, E₂), V₂} {(fN, fE), fV} {(gN, gE), gV}
  = (𝒯₂≈, {!!}), {!!}   -- ???
```

]

[WK: *The hole in indmorE has type Mor₂ (T.obj (T.obj V)) V:*

```
undistr''' : NatTrans (M.M ⨟⨟ M.M ⨟⨟ F) (F ⨟⨟ M.M ⨟⨟ M.M)
undistr''' = record
  {indmor = λ {{(N, E), V} → (ⓣ₂ {𝒯₂}   -- : Mor₂ 𝒯₂ 𝒯₂
                              , Id₂ {N ⊠₂ N} ⊗₂ {!!}   -- ???
                              ), T.return {𝒯₂} ⨟₂ T.return   -- : Mor₂ 𝒯₂ (T.obj (T.obj 𝒯₂))
                              }
  ; naturality = λ
      {{(N₀, E₀), V₀} {(N₁, E₁), V₁} {(fN, fE), fV}
      → (𝒯₂≈   -- independent of indmor
        , {! !}   -- involves only indmorE –
        ), (C₂.leftId
          ⟨≈₂≈˘⟩ (C₂.⨟-cong₂ (T.mor-cong T.mor-Id
          ⟨≈₂≈⟩ T.mor-Id) ⟨≈₂≈⟩ C₂.rightId))   -- independent of indmor
      }
  }
```

]

[WK: *The hole has type T.mor (fV ⨟₂ ⓣ₂) ≈₂ T.mor fV ⨟₂ T.join:*

```
distrJoin₀-wrapped : {A₀ A₁ : Obj₀} {f : M.KMor A₀ (F.obj (M.obj A₁))}
  → M.mor (f ⨟₀ distrJoin₀ {A₁}) ≈₀ M.↼ f
distrJoin₀-wrapped {(N₀, E₀), V₀} {(N₁, E₁), V₁} {(fN, fE), fV}
  = (𝒯₂≈, C₂.⨟-cong₂ (≈₂-sym C₂.rightId)), {!!}   -- ???
```

]

5.6 Categoric.MonCoAlg.SEAG.Cat2

We now produce the MonCoAlg category from $\mathcal{F}$ and $\mathcal{M}$ as defined in Categoric.MonCoAlg.SEAG.FM (Sect. 5.2).

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

```
open CatFinLimits


module Categoric.MonCoAlg.SEAG.Cat2
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (𝒯 : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  where
open import Categoric.MonCoAlg.SEAG.FM           𝒞₂ 𝒯 hasTerminal₂ hasProducts₂ using (𝒞₀; ℳ; ℱ)
open import Categoric.MonCoAlg.SEAG.ObjCompat 𝒞₂ 𝒯 hasTerminal₂ hasProducts₂
  using (module SEAG-MonCoAlg)
open import Categoric.MonCoAlg.SEAG.FMProps    𝒞₂ 𝒯 hasTerminal₂ hasProducts₂
  using (distrJoin; return-distrJoin₀; join-distrJoin₀; join↗distrJoin₀)
open Category₂ 𝒞₂
private
  module 𝒞₂ = Category 𝒞₂


module SEAG-MonCoAlg-Cat2 where
  open import Categoric.MonCoAlg.Cat2 𝒞₀ ℳ ℱ distrJoin
    (λ {A} → return-distrJoin₀ {A})
    (λ {A} → join-distrJoin₀ {A})
    (λ {A} → join↗distrJoin₀ {A})
    public using (MCACategory; MCAMor; MkMCAMor; module MCAMor; _↼̟-distrJoin)
  open SEAG-MonCoAlg public
open SEAG-MonCoAlg-Cat2 public
```

5.7 Categoric.MonCoAlg.SEAG.Cat2.MorCompat

We establish the lowest layer of the correspondence between SEAGraph morphisms and SEAG-MonCoAlg-Cat2 morphisms.

```
open import RATH.Level
open import RATH.Data.Product using (_,_; proj₁; proj₂)
open import Categoric.Category
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad
open CatFinLimits


module Categoric.MonCoAlg.SEAG.Cat2.MorCompat
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (𝒯 : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  where
open import Data.SEAGraph 𝒞₂ 𝒯
open import Categoric.MonCoAlg.SEAG.FM           𝒞₂ 𝒯 hasTerminal₂ hasProducts₂
  using (module ℳ)
open import Categoric.MonCoAlg.SEAG.ObjCompat 𝒞₂ 𝒯 hasTerminal₂ hasProducts₂
  using (objFrom; objTo; objToFrom)
open import Categoric.MonCoAlg.SEAG.Cat2         𝒞₂ 𝒯 hasTerminal₂ hasProducts₂
  using (MonCoAlg; module MonCoAlg; MCAMor; MkMCAMor; MCACategory)
open Category₂ 𝒞₂
open HasTerminalObject₂ 𝒞₂ hasTerminal₂
```

```
open HasProducts₂ 𝒞₂ hasProducts₂
private
  module 𝒞₂ = Category 𝒞₂
  module 𝒯 = Monad′ 𝒯
  module 𝒫₂′ = HasProducts 𝒞₂ hasProducts₂  -- for Category.FinColimits material


morFrom : {G₁ G₂ : SEAGraph} → SEAGraphMor G₁ G₂ → MCAMor (objFrom G₁) (objFrom G₂)
morFrom {G₁} {G₂} F = record
  {mor = (f-N, f-E), f-V
  ;commutes = (Ⓓ ₂≈
    ,(≈₂-begin
        f-E ⨟₂ ((G₂.src ▽₂ G₂.trg) ▽₂ G₂.attr) ⨟₂ Id₂
      ≈₂⟨ 𝒞₂.⨟-cong₂ rightId₂ ⟨≈₂≈⟩ ⨟-▽₂ ⟨≈₂≈⟩ ▽₂-cong₁ ⨟-▽₂ ⟩
        ((f-E ⨟₂ G₂.src) ▽₂ (f-E ⨟₂ G₂.trg)) ▽₂ (f-E ⨟₂ G₂.attr)
      ≈₂⟨ ▽₂-cong (▽₂-cong prop-src prop-trg) prop-attr ⟩
        (G₁.src ⨟₂ f-N ▽₂ G₁.trg ⨟₂ f-N) ▽₂ G₁.attr ⨟₂ 𝒯.↼ f-V
      ≈₂˘⟨ 𝒫₂.▽-⨟-⊗ ⟨≈₂≈⟩ ▽₂-cong₁ 𝒫₂.▽-⨟-⊗ ⟩
        ((G₁.src ▽₂ G₁.trg) ▽₂ G₁.attr) ⨟₂ ((f-N ⊗₂ f-N) ⊗₂ 𝒯.↼ f-V)
      ≈₂˘⟨ 𝒞₂.⨟-cong₂ (𝒫₂.⊗-cong₁ (𝒫₂.⊗-cong 𝒞₂.rightId² 𝒞₂.rightId²)) ⟩
        (((G₁.src ▽₂ G₁.trg) ▽₂ G₁.attr) ⨟₂
          ((f-N ⨟₂ Id₂ ⨟₂ Id₂ ⊗₂ f-N ⨟₂ Id₂ ⨟₂ Id₂) ⊗₂ 𝒯.↼ f-V)
      □₂))
    ),Ⓓ ₂≈
  }
  where
    module G₁ = SEAGraph G₁
    module G₂ = SEAGraph G₂
    open SEAGraphMor F


morTo : {A₁ A₂ : MonCoAlg} → MCAMor A₁ A₂ → SEAGraphMor (objTo A₁) (objTo A₂)
morTo {A₁} {A₂} (MkMCAMor ((F-N, F-E), F-V) ((_, commutesE), _)) = record
  {f-N = F-N
  ;f-E = F-E
  ;f-V = F-V
  ;prop-src = ≈₂-begin
      F-E ⨟₂ opE₂ ⨟₂ π₂ ⨟₂ π₂
    ≈₂⟨ 𝒞₂.⨟-cong₂₁ rightId₂ ⟨≈₂˘≈⟩ 𝒞₂.⨟-assocL ⟩
      (F-E ⨟₂ opE₂ ⨟₂ Id₂) ⨟₂ π₂ ⨟₂ π₂
    ≈₂⟨ 𝒞₂.⨟-cong₁ commutesE ⟨≈₂≈⟩ 𝒞₂.⨟-assoc ⟨≈₂≈⟩ 𝒞₂.⨟-cong₂ 𝒫₂.⊗⊗⨟ππ ⟩
      opE₁ ⨟₂ (π₂ ⨟₂ π₂) ⨟₂ F-N ⨟₂ Id₂ ⨟₂ Id₂
    ≈₂⟨ 𝒞₂.⨟-assocL ⟨≈₂≈⟩ 𝒞₂.⨟-cong₂ (𝒞₂.⨟-cong₂ rightId₂ ⟨≈₂≈⟩ rightId₂) ⟩
      (opE₁ ⨟₂ π₂ ⨟₂ π₂) ⨟₂ F-N
    □₂
  ;prop-trg = ≈₂-begin
      F-E ⨟₂ opE₂ ⨟₂ π₂ ⨟₂ ρ₂
    ≈₂⟨ 𝒞₂.⨟-cong₂₁ rightId₂ ⟨≈₂˘≈⟩ 𝒞₂.⨟-assocL ⟩
      (F-E ⨟₂ opE₂ ⨟₂ Id₂) ⨟₂ π₂ ⨟₂ ρ₂
    ≈₂⟨ 𝒞₂.⨟-cong₁ commutesE ⟨≈₂≈⟩ 𝒞₂.⨟-assoc ⟨≈₂≈⟩ 𝒞₂.⨟-cong₂ 𝒫₂.⊗⊗⨟πρ ⟩
      opE₁ ⨟₂ (π₂ ⨟₂ ρ₂) ⨟₂ F-N ⨟₂ Id₂ ⨟₂ Id₂
    ≈₂⟨ 𝒞₂.⨟-assocL ⟨≈₂≈⟩ 𝒞₂.⨟-cong₂ (𝒞₂.⨟-cong₂ rightId₂ ⟨≈₂≈⟩ rightId₂) ⟩
      (opE₁ ⨟₂ π₂ ⨟₂ ρ₂) ⨟₂ F-N
    □₂
  ;prop-attr = ≈₂-begin
      F-E ⨟₂ opE₂ ⨟₂ ρ₂
    ≈₂⟨ 𝒞₂.⨟-cong₂₁ rightId₂ ⟨≈₂˘≈⟩ 𝒞₂.⨟-assocL ⟩
      (F-E ⨟₂ opE₂ ⨟₂ Id₂) ⨟₂ ρ₂
    ≈₂⟨ 𝒞₂.⨟-cong₁ commutesE ⟨≈₂≈⟩ 𝒞₂.⨟-assoc ⟨≈₂≈⟩ 𝒞₂.⨟-cong₂ 𝒫₂.⊗⨟ρ ⟨≈₂≈⟩ 𝒞₂.⨟-assocL ⟩
      (opE₁ ⨟₂ ρ₂) ⨟₂ 𝒯.↼ F-V
```

```
    □₂
  }
  where
    opE₁ = proj₂ (proj₁ (MonCoAlg.op A₁))
    opE₂ = proj₂ (proj₁ (MonCoAlg.op A₂))


objToFrom′ : {A : MonCoAlg}
           → Category.Iso MCACategory (objFrom (objTo A)) A
objToFrom′ {A} = record
  {isoMor = MkMCAMor (𝒞₃.isoMor (objToFrom {A}))
      ((ⓣ₂≈, (leftId₂ ⟨≈₂≈˘⟩ 𝒞₂.⨟-cong
        (∇₂-cong₁ (∇₂-cong 𝒞₂.⨟-assocL 𝒞₂.⨟-assocL ⟨≈₂≈˘⟩ to-∇₂) ⟨≈₂≈˘⟩ to-∇₂)
        (𝒫₂.⊗-cong (𝒫₂.⊗-cong 𝒞₂.rightId² 𝒞₂.rightId² ⟨≈₂≈⟩ 𝒫₂′.⊗-Id) 𝒯.rightUnit ⟨≈₂≈⟩ 𝒫₂′.⊗-Id))
    ), ⓣ₂≈)
  ; isIso = record
    {_⁻¹ = MkMCAMor (𝒞₃._⁻¹ (objToFrom {A}))
      ((ⓣ₂≈, (leftId₂ ⟨≈₂≈⟩ 𝒞₂.⨟-cong₁ (∇₂-cong₁ (∇₂-cong 𝒞₂.⨟-assocL 𝒞₂.⨟-assocL ⟨≈₂≈˘⟩ to-∇₂)
                                          ⟨≈₂≈˘⟩ to-∇₂)
                              ⟨≈₂≈˘⟩ 𝒞₂.⨟-cong₂ (𝒫₂.⊗-cong (𝒫₂.⊗-cong 𝒞₂.rightId² 𝒞₂.rightId² ⟨≈₂≈⟩ 𝒫₂′.⊗-Id)
                                                              𝒯.rightUnit ⟨≈₂≈⟩ 𝒫₂′.⊗-Id))
      ), ⓣ₂≈)
    ; rightInverse = 𝒞₃.rightInverse (objToFrom {A})
    ; leftInverse = 𝒞₃.leftInverse (objToFrom {A})
    }
  }
  where
    module A = MonCoAlg A
    open Category₃ 𝓜.KleisliCat
    module 𝒞₃ = Category 𝓜.KleisliCat
```

5.8 Categoric.MonCoAlg.SEAG.Cat2.To

We produce the functor from the SEAG-MonCoAlg-Cat2 category to the SEAGraph category.

```
open import RATH.Level
open import RATH.Data.Product using (_,_)
open import Categoric.Category
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad
open CatFinLimits


module Categoric.MonCoAlg.SEAG.Cat2.To
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (𝒯 : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  where
open import Data.SEAGraph 𝒞₂ 𝒯 using (SEAGraphCat)
open import Categoric.MonCoAlg.SEAG.ObjCompat       𝒞₂ 𝒯 hasTerminal₂ hasProducts₂
  using (objTo)
open import Categoric.MonCoAlg.SEAG.Cat2            𝒞₂ 𝒯 hasTerminal₂ hasProducts₂
  using (MCACategory)
open import Categoric.MonCoAlg.SEAG.Cat2.MorCompat 𝒞₂ 𝒯 hasTerminal₂ hasProducts₂
  using (morTo)
```

```
open Category₂ 𝒞₂
private
  module 𝒞₂ = Category 𝒞₂


To : Functor MCACategory SEAGraphCat
To = record
  {obj = objTo
  ; mor = morTo
  ; mor-cong = λ {((≈N, ≈E), ≈V) → record {f-N≋ = ≈N; f-E≋ = ≈E; f-V≋ = ≈V}}
  ; mor-⨟ = record {f-N≋ = 𝒞₂.⨟-cong₂ 𝒞₂.rightId²; f-E≋ = 𝒞₂.⨟-cong₂ 𝒞₂.rightId²; f-V≋ = ≈₂-refl}
  ; mor-Id = record {f-N≋ = ≈₂-refl; f-E≋ = ≈₂-refl; f-V≋ = ≈₂-refl}
  }
```

5.9 Categoric.MonCoAlg.SEAG.Cat2.From

We produce the functor from the SEAGraph category to the SEAG-MonCoAlg-Cat2 category.

```
open import RATH.Level
open import RATH.Data.Product using (_,_)
open import Categoric.Category
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad
open CatFinLimits


module Categoric.MonCoAlg.SEAG.Cat2.From
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (𝒯 : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  where
open import Data.SEAGraph 𝒞₂ 𝒯
open import Categoric.MonCoAlg.SEAG.ObjCompat      𝒞₂ 𝒯 hasTerminal₂ hasProducts₂
  using (objFrom)
open import Categoric.MonCoAlg.SEAG.Cat2           𝒞₂ 𝒯 hasTerminal₂ hasProducts₂
  using (MCACategory)
open import Categoric.MonCoAlg.SEAG.Cat2.MorCompat 𝒞₂ 𝒯 hasTerminal₂ hasProducts₂
  using (morFrom)
open Category₂ 𝒞₂
open HasTerminalObject₂ 𝒞₂ hasTerminal₂
open HasProducts₂ 𝒞₂ hasProducts₂
private
  module 𝒞₂ = Category 𝒞₂
  module 𝒯 = Monad′ 𝒯


From : Functor SEAGraphCat MCACategory
From = record
  {obj = objFrom
  ; mor = morFrom
  ; mor-cong = λ F₁≈F₂ → let open _≋_ F₁≈F₂ in (f-N≋, f-E≋), f-V≋
  ; mor-⨟    = (≈₂-sym (𝒞₂.⨟-cong₂ 𝒞₂.rightId²), ≈₂-sym (𝒞₂.⨟-cong₂ 𝒞₂.rightId²)), ≈₂-refl
  ; mor-Id   = (≈₂-refl, ≈₂-refl), ≈₂-refl
  }
```

5.10 Categoric.MonCoAlg.SEAG.Cat2.ToFrom

We show one side of the equivalence between the SEAGraph category and the corresponding MonCoAlg category.

```
open import RATH.Level
open import RATH.Data.Product using (_,_)
open import Categoric.Category
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad
open CatFinLimits


module Categoric.MonCoAlg.SEAG.Cat2.ToFrom
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (𝒯 : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  where
open import Categoric.MonCoAlg.SEAG.Cat2.From        𝒞₂ 𝒯 hasTerminal₂ hasProducts₂ using (From)
open import Categoric.MonCoAlg.SEAG.Cat2.To          𝒞₂ 𝒯 hasTerminal₂ hasProducts₂ using (To)
open import Categoric.MonCoAlg.SEAG.Cat2.MorCompat 𝒞₂ 𝒯 hasTerminal₂ hasProducts₂
  using (objToFrom′)
open import Categoric.MonCoAlg.SEAG.Cat2             𝒞₂ 𝒯 hasTerminal₂ hasProducts₂
  using (MCACategory)
open Category₂ 𝒞₂
private
  module 𝒞₂ = Category 𝒞₂
  module 𝒯 = Monad′ 𝒯


ToFrom : NatIso (To ⨟ From) (Identity MCACategory)
ToFrom = IsoNat objToFrom′
  (((𝒞₂.rightId³ ⟨≈₂≈˘⟩ (leftId₂ ⟨≈₂≈⟩ 𝒞₂.rightId²))
  , (𝒞₂.rightId³ ⟨≈₂≈˘⟩ (leftId₂ ⟨≈₂≈⟩ 𝒞₂.rightId²)))
  , (𝒞₂.⨟-cong₂ 𝒯.rightUnit ⟨≈₂≈˘⟩ (𝒞₂.⨟-cong₁&₂₁ 𝒯.return-naturality ⟨≈₂˘≈⟩ 𝒞₂.⨟-cong₂ 𝒯.leftUnit))
  )
```

5.11 Categoric.MonCoAlg.SEAG.Cat2.FromTo

We show one side of the equivalence between the SEAGraph category and the corresponding MonCoAlg category.

```
open import RATH.Level
open import Categoric.Category
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad
open CatFinLimits


module Categoric.MonCoAlg.SEAG.Cat2.FromTo
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (𝒯 : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  where
open import Categoric.MonCoAlg.SEAG.ObjCompat 𝒞₂ 𝒯 hasTerminal₂ hasProducts₂ using (objFromTo)
```

```
open import Categoric.MonCoAlg.SEAG.Cat2.From  𝒞₂ 𝒯 hasTerminal₂ hasProducts₂ using (From)
open import Categoric.MonCoAlg.SEAG.Cat2.To    𝒞₂ 𝒯 hasTerminal₂ hasProducts₂ using (To)
open import Data.SEAGraph 𝒞₂ 𝒯
open Category₂ 𝒞₂
private
  module 𝒞₂ = Category 𝒞₂
  module 𝒯 = Monad′ 𝒯


FromTo : NatIso (From ⨟⨟ To) (Identity SEAGraphCat)
FromTo = IsoNat objFromTo
  (λ {G₁} {G₂} {F} → let module F = SEAGraphMor F in record
    {f-N≋ = rightId₂ ⟨≈₂≈˘⟩ leftId₂
    ;f-E≋ = rightId₂ ⟨≈₂≈˘⟩ leftId₂
    ;f-V≋ = ≈₂-begin
        F.f-V ⨟₂ 𝒯.mor 𝒯.return ⨟₂ 𝒯.join
      ≈₂⟨ 𝒞₂.⨟-cong₂ (𝒯.rightUnit ⟨≈₂≈˘⟩ 𝒯.leftUnit) ⟩
        F.f-V ⨟₂ 𝒯.return ⨟₂ 𝒯.join
      ≈₂⟨ 𝒞₂.⨟-cong₁&₂₁ 𝒯.return-naturality ⟩
        𝒯.return ⨟₂ 𝒯.mor F.f-V ⨟₂ 𝒯.join
      □₂
    })
```

5.12 Categoric.MonCoAlg.SEAG.Cat2.FromToFunctor

Since monolithic Categoric.MonCoAlg.SEAG.Cat2.FromTo requires 13.5 GB heap, we provide also a split version — here we just define the functor composition From ⨟⨟ To.

```
open import RATH.Level
open import Categoric.Category
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad
open CatFinLimits


module Categoric.MonCoAlg.SEAG.Cat2.FromToFunctor
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (𝒯 : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  where
open import Categoric.MonCoAlg.SEAG.Cat2.From 𝒞₂ 𝒯 hasTerminal₂ hasProducts₂ using (From)
open import Categoric.MonCoAlg.SEAG.Cat2.To   𝒞₂ 𝒯 hasTerminal₂ hasProducts₂ using (To)
open import Data.SEAGraph 𝒞₂ 𝒯 using (SEAGraphCat)


FromToFunctor : Functor SEAGraphCat SEAGraphCat
FromToFunctor = From ⨟⨟ To
```

5.13 Categoric.MonCoAlg.SEAG.Cat2.FromToNaturality

Since monolithic Categoric.MonCoAlg.SEAG.Cat2.FromTo requires 13.5 GB heap, we provide also a split version — here we use the "pre-composed" functor From ⨟⨟ To from Categoric.MonCoAlg.SEAG.Cat2.FromToFunctor (Sect. 5.12) and show the naturality condition for objFromTo.

```
open import RATH.Level

open import Categoric.Category
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad

open CatFinLimits


module Categoric.MonCoAlg.SEAG.Cat2.FromToNaturality
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (𝒯 : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  where

open import Categoric.MonCoAlg.SEAG.ObjCompat 𝒞₂ 𝒯 hasTerminal₂ hasProducts₂ using (objFromTo)
open import Categoric.MonCoAlg.SEAG.Cat2.FromToFunctor
  𝒞₂ 𝒯 hasTerminal₂ hasProducts₂ using (FromToFunctor)

open import Data.SEAGraph 𝒞₂ 𝒯
  using (SEAGraph; SEAGraphMor; module SEAGraphMor; SEAGraphCat)

private
  module 𝒞₂ = Category 𝒞₂
  module 𝒯 = Monad′ 𝒯
  module 𝒞₄ = Category SEAGraphCat
  module FromToF = Functor FromToFunctor
open Category₂ 𝒞₂
open Category₄ SEAGraphCat


objFromTo-naturality : {G₁ G₂ : SEAGraph} {F : SEAGraphMor G₁ G₂}
                     → FromToF.mor F ⨟₄ 𝒞₄.isoMor (objFromTo {G₂})
                     ≈₄ 𝒞₄.isoMor (objFromTo {G₁}) ⨟₄ F
objFromTo-naturality {G₁} {G₂} {F} = let module F = SEAGraphMor F in record
  {f-N≋ = rightId₂ ⟨≈₂≈˘⟩ leftId₂
  ;f-E≋ = rightId₂ ⟨≈₂≈˘⟩ leftId₂
  ;f-V≋ = ≈₂-begin
      F.f-V ⨟₂ 𝒯.mor 𝒯.return ⨟₂ 𝒯.join
    ≈₂⟨ 𝒞₂.⨟-cong₂ (𝒯.rightUnit ⟨≈₂≈˘⟩ 𝒯.leftUnit) ⟩
      F.f-V ⨟₂ 𝒯.return ⨟₂ 𝒯.join
    ≈₂⟨ 𝒞₂.⨟-cong₁&₂₁ 𝒯.return-naturality ⟩
      𝒯.return ⨟₂ 𝒯.mor F.f-V ⨟₂ 𝒯.join
    □₂
  }
```

5.14 Categoric.MonCoAlg.SEAG.Cat2.FromTo1

Since monolithic Categoric.MonCoAlg.SEAG.Cat2.FromTo requires 13.5 GB heap, we provide also a split version — here we use the "pre-composed" functor From ⨟⨟ To from Categoric.MonCoAlg.SEAG.Cat2.FromToFunctor (Sect. 5.12) and the naturality shown in Categoric.MonCoAlg.SEAG.Cat2.FromToNaturality (Sect. 5.13).

```
open import RATH.Level

open import Categoric.Category
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad

open CatFinLimits
```

```
module Categoric.MonCoAlg.SEAG.Cat2.FromTo1
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (𝒯 : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  where
open import Categoric.MonCoAlg.SEAG.ObjCompat
  𝒞₂ 𝒯 hasTerminal₂ hasProducts₂ using (objFromTo)
open import Categoric.MonCoAlg.SEAG.Cat2.FromToFunctor
  𝒞₂ 𝒯 hasTerminal₂ hasProducts₂ using (FromToFunctor)
open import Categoric.MonCoAlg.SEAG.Cat2.FromToNaturality
  𝒞₂ 𝒯 hasTerminal₂ hasProducts₂ using (objFromTo-naturality)
open import Data.SEAGraph 𝒞₂ 𝒯 using (SEAGraphCat)


FromTo : NatIso FromToFunctor (Identity SEAGraphCat)
FromTo = IsoNat objFromTo objFromTo-naturality
```

5.15 Categoric.MonCoAlg.SEAG.Cat2.Equiv

We collect all of the equivalence between the SEAGraph category and the SEAG-MonCoAlg-Cat2 category into a single proof.

```
open import RATH.Level
open import RATH.Data.Product using (_,_)
open import Categoric.Category
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Functor.Equivalence
open import Categoric.Monad
open CatFinLimits


module Categoric.MonCoAlg.SEAG.Cat2.Equiv
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (𝒯 : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  where
open import Data.SEAGraph 𝒞₂ 𝒯 using (SEAGraphCat)
open import Categoric.MonCoAlg.SEAG.Cat2.From      𝒞₂ 𝒯 hasTerminal₂ hasProducts₂ using (From)
open import Categoric.MonCoAlg.SEAG.Cat2.To        𝒞₂ 𝒯 hasTerminal₂ hasProducts₂ using (To)
open import Categoric.MonCoAlg.SEAG.Cat2.FromTo1 𝒞₂ 𝒯 hasTerminal₂ hasProducts₂ using (FromTo)
open import Categoric.MonCoAlg.SEAG.Cat2.ToFrom  𝒞₂ 𝒯 hasTerminal₂ hasProducts₂ using (ToFrom)
open import Categoric.MonCoAlg.SEAG.Cat2           𝒞₂ 𝒯 hasTerminal₂ hasProducts₂
  using (MCACategory)


MonCoAlg-SEAGraph-Equivalence : CatEquivalence MCACategory SEAGraphCat
MonCoAlg-SEAGraph-Equivalence = record
  { To     = To
  ; From   = From
  ; ToFrom = ToFrom
  ; FromTo = FromTo
  }
```

5.16 Categoric.MonCoAlg.SEAG.Cat2.Pushout3

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import Categoric.Category
open import Categoric.Category.FinColimits
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad
open CatFinLimits


module Categoric.MonCoAlg.SEAG.Cat2.Pushout3
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (𝒯 : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  where
open import Categoric.MonCoAlg.SEAG.FM  𝒞₂ 𝒯 hasTerminal₂ hasProducts₂
open import Categoric.MonCoAlg.SEAG.Cat2 𝒞₂ 𝒯 hasTerminal₂ hasProducts₂
open Category 𝒞₀
Obj = Obj₀
open Category₂ 𝒞₂
open HasTerminalObject₂ 𝒞₂ hasTerminal₂
open HasProducts₂ 𝒞₂ hasProducts₂
open CatFinColimits 𝒞₀
open ℳ using (↼; _⨾∘_)
ℱℳ = ℱ ⨾⨾ ℳ.M
module ℱℳ = Functor ℱℳ
```

distrJoin is used to construct the operation D.op of the pushout object, and distrJoin-wrapped is used to show the commutes property of the two pushout morphisms.

```
module MCAMor-Pushout
  {A B C : MonCoAlg} {D₀ : Obj} (F : MCAMor A B) (G : MCAMor A C)
  (let module A = MonCoAlg A)
  (let module B = MonCoAlg B)
  (let module C = MonCoAlg C)
  (let module F = MCAMor F)
  (let module G = MCAMor G)
  (H₀ : ℳ.KMor B.Carrier D₀) (K₀ : ℳ.KMor C.Carrier D₀)
  (isPushout : CatFinColimits.IsPushout ℳ.KleisliCat F.mor G.mor H₀ K₀) where


module isPO = CatFinColimits.IsPushout ℳ.KleisliCat isPushout
A-opE = proj₂ (proj₁ A.op)
F-N = proj₁ (proj₁ F.mor)
F-E = proj₂ (proj₁ F.mor)
F-V = proj₂ F.mor
G-N = proj₁ (proj₁ G.mor)
G-E = proj₂ (proj₁ G.mor)
G-V = proj₂ G.mor
D-N = proj₁ (proj₁ D₀)
D-E = proj₂ (proj₁ D₀)
D-V = proj₂ D₀
B-opE = proj₂ (proj₁ B.op)
B-V = proj₂ B.Carrier
C-opE = proj₂ (proj₁ C.op)
```

```
C-V = proj₂ C.Carrier
H-N = proj₁ (proj₁ H₀)
H-E = proj₂ (proj₁ H₀)
H-V = proj₂ H₀
K-N = proj₁ (proj₁ K₀)
K-E = proj₂ (proj₁ K₀)
K-V = proj₂ K₀
D′V : Obj₂
D′V = D-V
D′ : Obj
D′ = (𝕋₂, ((D-N ⊠₂ D-N) ⊠₂ 𝒯.obj D-V)), D′V
H′V : Mor₂ B-V (𝒯.obj D′V)
H′V = H-V
H′ : ℳ.KMor B.Carrier D′
H′ = (𝕋₂, B-opE ⨟₂ ((H-N ⊗₂ H-N) ⊗₂ 𝒯.↼ H-V))
   , H′V  -- originally: B.op ⨟ ℱ.mor (↼ H₀) ⨟ ℳ.return
K′V : Mor₂ C-V (𝒯.obj D′V)
K′V = K-V
K′ : ℳ.KMor C.Carrier D′
K′ = (𝕋₂, C-opE ⨟₂ ((K-N ⊗₂ K-N) ⊗₂ 𝒯.↼ K-V))
   , K′V  -- originally: C.op ⨟ ℱ.mor (↼ K₀) ⨟ ℳ.return

F-commutes-E : F-E ⨟₂ B-opE ≈₂ A-opE ⨟₂ ((F-N ⊗₂ F-N) ⊗₂ 𝒯.↼ F-V)
F-commutes-E = 𝒞₂.⨟-cong₂ rightId₂ ⟨≈₂˘≈⟩ proj₂ (proj₁ F.commutes) ⟨≈₂≈⟩
               𝒞₂.⨟-cong₂ (𝒫₂.⊗-cong₁ (𝒫₂.⊗-cong 𝒞₂.rightId² 𝒞₂.rightId²))
G-commutes-E : G-E ⨟₂ C-opE ≈₂ A-opE ⨟₂ ((G-N ⊗₂ G-N) ⊗₂ 𝒯.↼ G-V)
G-commutes-E = 𝒞₂.⨟-cong₂ rightId₂ ⟨≈₂˘≈⟩ proj₂ (proj₁ G.commutes) ⟨≈₂≈⟩
               𝒞₂.⨟-cong₂ (𝒫₂.⊗-cong₁ (𝒫₂.⊗-cong 𝒞₂.rightId² 𝒞₂.rightId²))
isPO-commutes-N : F-N ⨟₂ H-N ≈₂ G-N ⨟₂ K-N
isPO-commutes-N = 𝒞₂.⨟-cong₂ 𝒞₂.rightId² ⟨≈₂˘≈⟩ proj₁ (proj₁ isPO.commutes)
                  ⟨≈₂≈⟩ 𝒞₂.⨟-cong₂ 𝒞₂.rightId²
isPO-commutes-E : F-E ⨟₂ H-E ≈₂ G-E ⨟₂ K-E
isPO-commutes-E = 𝒞₂.⨟-cong₂ 𝒞₂.rightId² ⟨≈₂˘≈⟩ proj₂ (proj₁ isPO.commutes)
                  ⟨≈₂≈⟩ 𝒞₂.⨟-cong₂ 𝒞₂.rightId²
isPO-commutes-V : F-V ⨟₂ 𝒯.↼ H-V ≈₂ G-V ⨟₂ 𝒯.↼ K-V
isPO-commutes-V = proj₂ isPO.commutes
opPOuniv : CatFinColimits.CoCone2Univ ℳ.KleisliCat H₀ K₀ H′ K′
opPOuniv = isPO.universal {D′} {H′} {K′}
  ((𝕋₂≈
    , (≈₂-begin
         F-E ⨟₂ (B-opE ⨟₂ ((H-N ⊗₂ H-N) ⊗₂ 𝒯.↼ H-V)) ⨟₂ Id₂ ⨟₂ Id₂
       ≈₂⟨ 𝒞₂.⨟-cong₂ 𝒞₂.rightId² ⟨≈₂≈⟩ 𝒞₂.⨟-cong₁&₂₁ F-commutes-E ⟩
         A-opE ⨟₂ ((F-N ⊗₂ F-N) ⊗₂ 𝒯.↼ F-V) ⨟₂ ((H-N ⊗₂ H-N) ⊗₂ 𝒯.↼ H-V)
       ≈₂⟨ 𝒞₂.⨟-cong₂ (𝒫₂.⨟-⊗-⨟ ⟨≈₂˘≈˘⟩ 𝒫₂.⊗-cong₁ 𝒫₂.⨟-⊗-⨟) ⟩
         A-opE ⨟₂ (((F-N ⨟₂ H-N) ⊗₂ (F-N ⨟₂ H-N)) ⊗₂ (𝒯.↼ F-V ⨟₂ 𝒯.↼ H-V))
       ≈₂⟨ 𝒞₂.⨟-cong₂ (𝒫₂.⊗-cong (𝒫₂.⊗-cong isPO-commutes-N isPO-commutes-N)
         (𝒯.↼-⨟-↼ ⟨≈₂≈⟩ 𝒯.↼-cong isPO-commutes-V ⟨≈₂≈˘⟩ 𝒯.↼-⨟-↼)) ⟩
         A-opE ⨟₂ (((G-N ⨟₂ K-N) ⊗₂ (G-N ⨟₂ K-N)) ⊗₂ (𝒯.↼ G-V ⨟₂ 𝒯.↼ K-V))
       ≈₂˘⟨ 𝒞₂.⨟-cong₂ (𝒫₂.⨟-⊗-⨟ ⟨≈₂˘≈˘⟩ 𝒫₂.⊗-cong₁ 𝒫₂.⨟-⊗-⨟) ⟩
         A-opE ⨟₂ ((G-N ⊗₂ G-N) ⊗₂ 𝒯.↼ G-V) ⨟₂ ((K-N ⊗₂ K-N) ⊗₂ 𝒯.↼ K-V)
       ≈₂˘⟨ 𝒞₂.⨟-cong₂ 𝒞₂.rightId² ⟨≈₂≈⟩ 𝒞₂.⨟-cong₁&₂₁ G-commutes-E ⟩
         G-E ⨟₂ (C-opE ⨟₂ ((K-N ⊗₂ K-N) ⊗₂ 𝒯.↼ K-V)) ⨟₂ Id₂ ⨟₂ Id₂
       □₂)
    ), isPO-commutes-V  -- : F-V ⨟₂ 𝒯.↼ H′V ≈₂ G-V ⨟₂ 𝒯.↼ K′V in Mor₂ A-V (𝒯.obj D′V)
  )
module opPOuniv = CatFinColimits.CoCone2Univ ℳ.KleisliCat opPOuniv

D : MonCoAlg
D = record
```

```
  {Carrier = D₀
  ;op      =  opPOuniv.univMor ⨟ ((𝔗₂ {𝔗 ₂}, Id₂), 𝔗₂ {𝒯.obj D-V})
  }
module D = MonCoAlg D
D-opE = proj₂ (proj₁ D.op)
H : MCAMor B D
H = record
  {mor = H₀
  ;commutes =
    ( 𝔗 ₂≈
    , (≈₂-begin
            H-E ⨟₂ D-opE ⨟₂ Id₂
          ≈₂⟨ 𝒞₂.⨟-cong₂ 𝒞₂.⨟-assoc ⟨≈₂≈⟩ proj₂ (proj₁ opPOuniv.univMor-factors-left) ⟩
            B-opE ⨟₂ ((H-N ⊗₂ H-N) ⊗₂ 𝒯.↼ H-V)
          ≈₂˘⟨ 𝒞₂.⨟-cong₂ (𝒫₂.⊗-cong₁ (𝒫₂.⊗-cong 𝒞₂.rightId² 𝒞₂.rightId²)) ⟩
            B-opE ⨟₂ ((H-N ⨟₂ Id₂ ⨟₂ Id₂ ⊗₂ H-N ⨟₂ Id₂ ⨟₂ Id₂) ⊗₂ 𝒯.↼ H-V)
          □₂)
    ), 𝔗 ₂≈   -- : H-N ⨟₂ 𝒯.mor D-opV ⨟₂ 𝔗₂ ≈₂ B-opV ⨟₂ Id₂
  }
K : MCAMor C D
K = record
  {mor = K₀
  ;commutes =
    ( 𝔗 ₂≈
    , (≈₂-begin
            K-E ⨟₂ D-opE ⨟₂ Id₂
          ≈₂⟨ 𝒞₂.⨟-cong₂ 𝒞₂.⨟-assoc ⟨≈₂≈⟩ proj₂ (proj₁ opPOuniv.univMor-factors-right) ⟩
            C-opE ⨟₂ ((K-N ⊗₂ K-N) ⊗₂ 𝒯.↼ K-V)
          ≈₂˘⟨ 𝒞₂.⨟-cong₂ (𝒫₂.⊗-cong₁ (𝒫₂.⊗-cong 𝒞₂.rightId² 𝒞₂.rightId²)) ⟩
            C-opE ⨟₂ ((K-N ⨟₂ Id₂ ⨟₂ Id₂ ⊗₂ K-N ⨟₂ Id₂ ⨟₂ Id₂) ⊗₂ 𝒯.↼ K-V)
          □₂)
    ), 𝔗 ₂≈
  }
module H = MCAMor H
module K = MCAMor K
PO : CatFinColimits.Pushout MCACategory F G
PO = record
  {obj = D
  ;left = H
  ;right = K
  ;prf = record
    {commutes = isPO.commutes
    ;universal = λ {Z} {H′} {K′} F⨟∘H′≈G⨟∘K′ → let
        module Z = MonCoAlg Z
        module H′ = MCAMor H′
        module K′ = MCAMor K′
        uPOuniv : CatFinColimits.CoCone2Univ 𝓜.KleisliCat H₀ K₀ H′.mor K′.mor
        uPOuniv = isPO.universal {Z.Carrier} {H′.mor} {K′.mor} F⨟∘H′≈G⨟∘K′
        module uPOuniv = CatFinColimits.CoCone2Univ 𝓜.KleisliCat uPOuniv
          -- uPOuniv.univMor-factors-left has the following three component types:
            -- H-N ⨟₂ univMor-N ⨟₂ Id₂ ⨟₂ Id₂ ≈₂ proj₁ (proj₁ H′.mor)
            -- H-E ⨟₂ univMor-E ⨟₂ Id₂ ⨟₂ Id₂ ≈₂ proj₂ (proj₁ H′.mor)
            -- H-V ⨟₂ 𝒯.↼ univMor-V ≈₂ proj₂ H′.mor
        Z′ : Obj
        Z′ = 𝓜.obj (ℱ.obj (𝓜.obj Z.Carrier))
          -- [ WK: ] We cannot add a distrJoin at the end since we need to be in Kleisli! ]
        H′Z : Mor B.Carrier Z′
        H′Z = H′.mor ⨟ 𝓜.mor Z.op
```

```
K′Z : Mor C.Carrier Z′
K′Z = K′.mor ⨟ 𝓜.mor Z.op
uOPcommutes : F.mor ⨟∘ H′Z ≈ G.mor ⨟∘ K′Z
uOPcommutes = ≈-begin
    F.mor ⨟∘ (H′.mor ⨟ 𝓜.mor Z.op)
  ≈⟨ 𝓜.⨟∘-⨟-assocL ⟩
    (F.mor ⨟∘ H′.mor) ⨟ 𝓜.mor Z.op
  ≈⟨ ⨟-cong₁ F⨟∘H′≈G⨟∘K′ ⟩
    (G.mor ⨟∘ K′.mor) ⨟ 𝓜.mor Z.op
  ≈⟨ 𝓜.⨟∘-⨟-assoc ⟩
    G.mor ⨟∘ (K′.mor ⨟ 𝓜.mor Z.op)
  □
uOPuniv : CatFinColimits.CoCone2Univ 𝓜.KleisliCat H₀ K₀ H′Z K′Z
uOPuniv = isPO.universal uOPcommutes
module uOPuniv = CatFinColimits.CoCone2Univ 𝓜.KleisliCat uOPuniv
U : MCAMor D Z
U = record
  {mor = uPOuniv.univMor
  ;commutes = let
      univMor-N : _
      univMor-N = proj₁ (proj₁ uPOuniv.univMor)
      univMor-E : _
      univMor-E = proj₂ (proj₁ uPOuniv.univMor)
      univMor-V : _
      univMor-V = proj₂ uPOuniv.univMor
      Z-opE : _
      Z-opE = proj₂ (proj₁ Z.op)
      Z-opV = proj₂ Z.op
    in (𝕋 ₂≈,
    (≈₂-begin
      univMor-E ⨟₂ Z-opE ⨟₂ Id₂
    ≈₂⟨ 𝒞₂.⨟-cong₂ rightId₂ ⟩
      univMor-E ⨟₂ Z-opE
    ≈₂⟨ proj₂ (proj₁ (uOPuniv.univMor-unique′
      {uPOuniv.univMor ⨟ 𝓜.mor Z.op}
      {proj₁ (D.op ⨟ ℱ.mor (↼ uPOuniv.univMor) ⨟ 𝓜.return), (univMor-V ⨟₂ 𝒯.mor Z-opV)}
      ((𝕋 ₂≈
        ,(≈₂-begin
            H-E ⨟₂ (univMor-E ⨟₂ Z-opE) ⨟₂ Id₂ ⨟₂ Id₂
          ≈₂⟨ 𝒞₂.⨟-cong₂ 𝒞₂.rightId² ⟩
            H-E ⨟₂ univMor-E ⨟₂ Z-opE
          ≈₂⟨ 𝒞₂.⨟-assocL ⟨≈₂≈⟩ 𝒞₂.⨟-cong₁ (𝒞₂.⨟-cong₂ 𝒞₂.rightId² ⟨≈₂˘≈⟩
                                              proj₂ (proj₁ uPOuniv.univMor-factors-left)) ⟩
            proj₂ (proj₁ H′.mor) ⨟₂ Z-opE
            □₂))
        ,(≈₂-begin
            H-V ⨟₂ 𝒯.↼ (univMor-V ⨟₂ 𝒯.mor Z-opV)
          ≈₂⟨ 𝒞₂.⨟-cong₂ 𝒯.↼-⨟M-mor ⟩
            H-V ⨟₂ 𝒯.↼ univMor-V ⨟₂ 𝒯.mor Z-opV
          ≈₂⟨ 𝒞₂.⨟-assocL ⟨≈₂≈⟩ 𝒞₂.⨟-cong₁ (proj₂ uPOuniv.univMor-factors-left) ⟩
            proj₂ H′.mor ⨟₂ 𝒯.mor Z-opV
          □₂))
      ((𝕋 ₂≈
        ,(≈₂-begin
            K-E ⨟₂ (univMor-E ⨟₂ Z-opE) ⨟₂ Id₂ ⨟₂ Id₂
          ≈₂⟨ 𝒞₂.⨟-cong₂ 𝒞₂.rightId² ⟩
            K-E ⨟₂ univMor-E ⨟₂ Z-opE
          ≈₂⟨ 𝒞₂.⨟-assocL ⟨≈₂≈⟩ 𝒞₂.⨟-cong₁ (𝒞₂.⨟-cong₂ 𝒞₂.rightId² ⟨≈₂˘≈⟩
                                              proj₂ (proj₁ uPOuniv.univMor-factors-right)) ⟩
```

```
        proj₂ (proj₁ K′.mor) ⨟₂ Z-opE
        □₂))
  , (≈₂-begin
        K-V ⨟₂ 𝒯.↼ (univMor-V ⨟₂ 𝒯.mor Z-opV)
      ≈₂⟨ 𝒞₂.⨟-cong₂ 𝒯.↼-⨟M-mor ⟩
        K-V ⨟₂ 𝒯.↼ univMor-V ⨟₂ 𝒯.mor Z-opV
      ≈₂⟨ 𝒞₂.⨟-assocL ⟨≈₂≈⟩ 𝒞₂.⨟-cong₁ (proj₂ uPOuniv.univMor-factors-right) ⟩
        proj₂ K′.mor ⨟₂ 𝒯.mor Z-opV
      □₂))
((𝕋 ₂≈, (≈₂-begin
        H-E ⨟₂ (D-opE ⨟₂ ((univMor-N ⨟₂ Id₂ ⨟₂ Id₂ ⊗₂ univMor-N ⨟₂ Id₂ ⨟₂ Id₂)
          ⊗₂ 𝒯.↼ univMor-V) ⨟₂ Id₂) ⨟₂ Id₂ ⨟₂ Id₂
      ≈₂⟨  𝒞₂.⨟-cong₂ (𝒞₂.rightId² ⟨≈₂≈⟩ 𝒞₂.⨟-cong₂ rightId₂) ⟨≈₂≈⟩
        𝒞₂.⨟-cong₁&₂₁ (𝒞₂.⨟-cong₂ rightId₂ ⟨≈₂˘≈⟩ proj₂ (proj₁ H.commutes)) ⟩
        B-opE ⨟₂ (((H-N ⨟₂ Id₂ ⨟₂ Id₂) ⊗₂ (H-N ⨟₂ Id₂ ⨟₂ Id₂)) ⊗₂ 𝒯.↼ H-V)
              ⨟₂ ((univMor-N ⨟₂ Id₂ ⨟₂ Id₂ ⊗₂ univMor-N ⨟₂ Id₂ ⨟₂ Id₂) ⊗₂ 𝒯.↼ univMor-V)
      ≈₂⟨ 𝒞₂.⨟-cong₂ (𝒫₂.⨟-⊗-⨟ ⟨≈₂˘≈⟩ 𝒫₂.⊗-cong₁
              (𝒫₂.⨟-⊗-⨟ ⟨≈₂˘≈⟩ 𝒫₂.⊗-cong (𝒞₂.⨟-cong₁ 𝒞₂.rightId²) (𝒞₂.⨟-cong₁ 𝒞₂.rightId²))) ⟩
        B-opE ⨟₂ (   ((H-N ⨟₂ univMor-N ⨟₂ Id₂ ⨟₂ Id₂) ⊗₂ (H-N ⨟₂ univMor-N ⨟₂ Id₂ ⨟₂ Id₂))
              ⊗₂ (𝒯.↼ H-V ⨟₂ 𝒯.↼ univMor-V))
      ≈₂⟨ 𝒞₂.⨟-cong₂ (𝒫₂.⊗-cong (𝒫₂.⊗-cong (proj₁ (proj₁ uPOuniv.univMor-factors-left))
                                             (proj₁ (proj₁ uPOuniv.univMor-factors-left)))
              (𝒯.↼-⨟-↼ ⟨≈₂≈⟩ 𝒯.↼-cong (proj₂ uPOuniv.univMor-factors-left))) ⟩
        B-opE ⨟₂ (   (proj₁ (proj₁ H′.mor) ⊗₂ proj₁ (proj₁ H′.mor))
              ⊗₂ 𝒯.↼ (proj₂ H′.mor))
    ≈₂⟨ (𝒞₂.⨟-cong₂ (𝒫₂.⊗-cong₁ (𝒫₂.⊗-cong 𝒞₂.rightId² 𝒞₂.rightId²))
      ⟨≈₂˘≈˘⟩ proj₂ (proj₁ H′.commutes)) ⟨≈₂≈⟩ 𝒞₂.⨟-cong₂ rightId₂ ⟩
      proj₂ (proj₁ H′.mor) ⨟₂ Z-opE
    ≈₂⟨ 𝒞₂.⨟-cong₁ (proj₂ (proj₁ uPOuniv.univMor-factors-left)
                ⟨≈₂˘≈⟩ 𝒞₂.⨟-cong₂ 𝒞₂.rightId²) ⟨≈₂≈⟩ 𝒞₂.⨟-assoc ⟩
      H-E ⨟₂ univMor-E ⨟₂ Z-opE
    ≈₂⟨ 𝒞₂.⨟-assocL ⟨≈₂≈⟩ 𝒞₂.⨟-cong₁
       (𝒞₂.⨟-cong₂ 𝒞₂.rightId² ⟨≈₂˘≈⟩ proj₂ (proj₁ uPOuniv.univMor-factors-left)) ⟩
      proj₂ (proj₁ H′.mor) ⨟₂ Z-opE
    □₂)
),   -- With unpatched second implicit argument to ...unique′
       -- this starts at:
       -- H-V ⨟₂ 𝒯.↼ ((proj₂ opPOuniv.univMor ⨟₂ 𝕥₂) ⨟₂ Id₂ ⨟₂ 𝒯.return)
    (≈₂-begin
      H-V ⨟₂ 𝒯.↼ (univMor-V ⨟₂ 𝒯.mor Z-opV)
    ≈₂⟨ 𝒞₂.⨟-cong₂ 𝒯.↼-⨟M-mor ⟩
      H-V ⨟₂ 𝒯.↼ univMor-V ⨟₂ 𝒯.mor Z-opV
    ≈₂⟨ 𝒞₂.⨟-assocL ⟨≈₂≈⟩ 𝒞₂.⨟-cong₁ (proj₂ uPOuniv.univMor-factors-left) ⟩
      proj₂ H′.mor ⨟₂ 𝒯.mor Z-opV
    □₂))
((𝕋 ₂≈, (≈₂-begin
      K-E ⨟₂ (D-opE ⨟₂ ((univMor-N ⨟₂ Id₂ ⨟₂ Id₂ ⊗₂ univMor-N ⨟₂ Id₂ ⨟₂ Id₂)
                ⊗₂ 𝒯.↼ univMor-V) ⨟₂ Id₂) ⨟₂ Id₂ ⨟₂ Id₂
    ≈₂⟨  𝒞₂.⨟-cong₂ (𝒞₂.rightId² ⟨≈₂≈⟩ 𝒞₂.⨟-cong₂ rightId₂)
      ⟨≈₂≈⟩ 𝒞₂.⨟-cong₁&₂₁ (𝒞₂.⨟-cong₂ rightId₂ ⟨≈₂˘≈⟩ proj₂ (proj₁ K.commutes)) ⟩
      C-opE ⨟₂ (((K-N ⨟₂ Id₂ ⨟₂ Id₂) ⊗₂ (K-N ⨟₂ Id₂ ⨟₂ Id₂)) ⊗₂ 𝒯.↼ K-V)
            ⨟₂ ((univMor-N ⨟₂ Id₂ ⨟₂ Id₂ ⊗₂ univMor-N ⨟₂ Id₂ ⨟₂ Id₂) ⊗₂ 𝒯.↼ univMor-V)
    ≈₂⟨ 𝒞₂.⨟-cong₂ (𝒫₂.⨟-⊗-⨟ ⟨≈₂˘≈⟩ 𝒫₂.⊗-cong₁
            (𝒫₂.⨟-⊗-⨟ ⟨≈₂˘≈⟩ 𝒫₂.⊗-cong (𝒞₂.⨟-cong₁ 𝒞₂.rightId²) (𝒞₂.⨟-cong₁ 𝒞₂.rightId²))) ⟩
      C-opE ⨟₂ (   ((K-N ⨟₂ univMor-N ⨟₂ Id₂ ⨟₂ Id₂) ⊗₂ (K-N ⨟₂ univMor-N ⨟₂ Id₂ ⨟₂ Id₂))
            ⊗₂ (𝒯.↼ K-V ⨟₂ 𝒯.↼ univMor-V))
    ≈₂⟨ 𝒞₂.⨟-cong₂ (𝒫₂.⊗-cong (𝒫₂.⊗-cong (proj₁ (proj₁ uPOuniv.univMor-factors-right))
                                           (proj₁ (proj₁ uPOuniv.univMor-factors-right)))
```

```
                    (𝒯.↼-⨟-↼ ⟨≈₂≈⟩ 𝒯.↼-cong (proj₂ uPOuniv.univMor-factors-right))) ⟩
              C-opE ⨟₂ (   (proj₁ (proj₁ K′.mor) ⊗₂ proj₁ (proj₁ K′.mor))
                    ⊗₂ 𝒯.↼ (proj₂ K′.mor))
            ≈₂⟨ (𝒞₂.⨟-cong₂ (𝒫₂.⊗-cong₁ (𝒫₂.⊗-cong 𝒞₂.rightId² 𝒞₂.rightId²))
                ⟨≈₂˘≈˘⟩ proj₂ (proj₁ K′.commutes)) ⟨≈₂≈⟩ 𝒞₂.⨟-cong₂ rightId₂ ⟩
              proj₂ (proj₁ K′.mor) ⨟₂ Z-opE
            ≈₂⟨ 𝒞₂.⨟-cong₁ (proj₂ (proj₁ uPOuniv.univMor-factors-right)
                        ⟨≈₂˘≈⟩ 𝒞₂.⨟-cong₂ 𝒞₂.rightId²) ⟨≈₂≈⟩ 𝒞₂.⨟-assoc ⟩
              K-E ⨟₂ univMor-E ⨟₂ Z-opE
            ≈₂⟨ 𝒞₂.⨟-assocL ⟨≈₂≈⟩ 𝒞₂.⨟-cong₁
                (𝒞₂.⨟-cong₂ 𝒞₂.rightId² ⟨≈₂˘≈⟩ proj₂ (proj₁ uPOuniv.univMor-factors-right)) ⟩
              proj₂ (proj₁ K′.mor) ⨟₂ Z-opE
            □₂)
          ), (≈₂-begin
              K-V ⨟₂ 𝒯.↼ (univMor-V ⨟₂ 𝒯.mor Z-opV)
            ≈₂⟨ 𝒞₂.⨟-cong₂ 𝒯.↼-⨟M-mor ⟩
              K-V ⨟₂ 𝒯.↼ univMor-V ⨟₂ 𝒯.mor Z-opV
            ≈₂⟨ 𝒞₂.⨟-assocL ⟨≈₂≈⟩ 𝒞₂.⨟-cong₁ (proj₂ uPOuniv.univMor-factors-right) ⟩
              proj₂ K′.mor ⨟₂ 𝒯.mor Z-opV
            □₂))
          )) ⟩
          D-opE ⨟₂ ((univMor-N ⨟₂ Id₂ ⨟₂ Id₂ ⊗₂ univMor-N ⨟₂ Id₂ ⨟₂ Id₂) ⊗₂ 𝒯.↼ univMor-V) ⨟₂ Id₂
        ≈₂⟨ 𝒞₂.⨟-cong₂ rightId₂ ⟩
          D-opE ⨟₂ ((univMor-N ⨟₂ Id₂ ⨟₂ Id₂ ⊗₂ univMor-N ⨟₂ Id₂ ⨟₂ Id₂) ⊗₂ 𝒯.↼ univMor-V)
        □₂))
        , 𝕀 ₂≈
      }
  in record
    {univMor = U
    ;univMor-factors-left = uPOuniv.univMor-factors-left
    ;univMor-factors-right = uPOuniv.univMor-factors-right
    ;univMor-unique = uPOuniv.univMor-unique
    }
  }
}
```

Chapter 6

Simple Term Graphs as Monadic Co-Algebras

Consider term graphs consisting of inner nodes and variable nodes, where inner nodes are labelled and have lists of successors:

$$\begin{array}{lll} \mathsf{sigTG} := \langle & \textbf{sorts:} & \mathsf{V}, \mathsf{N} \\ & \textbf{ops:} & \mathsf{lab} : \mathsf{N} \to L \\ & & \mathsf{suc} : \mathsf{N} \to \mathsf{List}\ (\mathsf{N} + \mathsf{V}) \quad \rangle \end{array}$$

For "substituting" term graph homomorphisms, we want to allow variables to be mapped also to inner nodes, and therefore adapt the type of F_V accordingly. The resulting adaptation in the commutativity condition for suc affects only the argument of the List functor:

Definition 6.0.1 We define the category TG to have sigTG-coalgebras as objects, and a morphism $F : G_1 \to G_2$ consists of two mappings

$$\begin{array}{lcl} F_\mathsf{N} & : & \mathsf{N}_1 \to \mathsf{N}_2 \\ F_\mathsf{V} & : & \mathsf{V}_1 \to \mathsf{N}_2 + \mathsf{V}_2 \end{array}$$

satisfying the following conditions:

$$\begin{array}{lcl} F_\mathsf{N} \mathbin{;} \mathsf{lab}_2 & = & \mathsf{lab}_1 \\ F_\mathsf{N} \mathbin{;} \mathsf{suc}_2 & = & \mathsf{suc}_1 \mathbin{;} \mathsf{List}((F_\mathsf{N} + F_\mathsf{V}) \mathbin{;} \mu_{(\mathsf{N}_2 +)}) \end{array}$$

where $\mu_{(\mathsf{N}+)} : \forall X\ .\ (\mathsf{N} + (\mathsf{N} + X)) \to (\mathsf{N} + X)$ is the canonical flattening function for nested alternatives with N. □

In Data.TermGraph (Sect. 6.1), we define the category of such simple term graphs directly, parameterised over a category of "sets". We then show how this is an instance of the monadic coalgebra definition of Chapter 4, with the equivalence of categories formally established in Categoric.MonCoAlg.TG.Cat2.Equiv (Sect. 6.14). (One part of this, Categoric.MonCoAlg.TG.Cat2.FromTo (Sect. 6.10), has been split into the three modules included after it because it requires over 15 GB of heap to typecheck on my installation.)

It is easy to see that, a pushout in the Kleisli category for the monad Categoric.MonCoAlg.TG.FM.$\mathcal{M}$ derived from the coproduct monad does *not* in extend to a pushout in the term graph category.

6.1 Data.TermGraph

```
open import RATH.Level
open import RATH.Data.Product as Product using (Σ:•; _,_; _×_; proj₁; proj₂)
open import Categoric.Category
open import Categoric.Monad
open import Categoric.LESGraph
open import Categoric.Semigroupoid
```

```
open import Categoric.Category.FinColimits
open import Categoric.Monad.FunctorMonad
open import Categoric.Monad.FunctorMonad.Coproduct
open CatFinColimits
```

[WK: *The list monad is a parameter for the time being.* **]**

```
module Data.TermGraph {ℓi ℓj ℓk : Level} {Obj : Set ℓi}
                      (𝒞 : Category ℓj ℓk Obj)
                      (hasCoproducts : HasCoproducts 𝒞)
                      (NLab : Obj) (ℒ : Monad 𝒞)
  where
  open Category 𝒞
  open HasCoproducts 𝒞 hasCoproducts
  ⊕ = CoproductFM 𝒞 hasCoproducts
  private
    module ℒ = Monad' ℒ
    module ⊕ = FunctorMonad ⊕
  open ℒ using (_⨟∘_)


  record TermGraph : Set (ℓi ⊔ ℓj) where
    field Node : Obj
          Var  : Obj
          nLab : Mor Node NLab
          suc  : Mor Node (ℒ.obj (Node ⊞ Var))


  record TermGraphMor (G₁ G₂ : TermGraph) : Set (ℓj ⊔ ℓk) where
    private
      module G₁ = TermGraph G₁
      module G₂ = TermGraph G₂
    field f-N : Mor G₁.Node G₂.Node
          f-V : Mor G₁.Var (G₂.Node ⊞ G₂.Var)
    field prop-nLab : f-N ⨟ G₂.nLab ≈ G₁.nLab
          prop-suc  : f-N ⨟ G₂.suc  ≈ G₁.suc ⨟ ℒ.mor ((f-N ⊕ f-V) ⨟ ⊕.join)


  record _≋_ {G₁ G₂ : TermGraph} (Φ Ψ : TermGraphMor G₁ G₂) : Set ℓk where
    private
      module Φ = TermGraphMor Φ
      module Ψ = TermGraphMor Ψ
    field f-N≋ : Φ.f-N ≈ Ψ.f-N
          f-V≋ : Φ.f-V ≈ Ψ.f-V


  TermGraphHom : LocalSetoid TermGraph (ℓj ⊔ ℓk) ℓk
  TermGraphHom G₁ G₂ = record
    {Carrier = TermGraphMor G₁ G₂
    ; _≈_ = _≋_
    ; isEquivalence = record
      {refl = record {f-N≋ = ≈-refl; f-V≋ = ≈-refl}
      ; sym = λ Φ≋Ψ → let open _≋_ Φ≋Ψ in record
        {f-N≋ = ≈-sym f-N≋; f-V≋ = ≈-sym f-V≋}
      ; trans = λ Φ≋X X≋Ψ → let open _≋_ in record
        {f-N≋ = ≈-trans (f-N≋ Φ≋X) (f-N≋ X≋Ψ)
        ; f-V≋ = ≈-trans (f-V≋ Φ≋X) (f-V≋ X≋Ψ)
        }
      }
```

```
  }
  where
    module G₁ = TermGraph G₁
    module G₂ = TermGraph G₂
open LocalSetoidCalc₁ TermGraphHom

comp : {G₁ G₂ G₃ : TermGraph} → TermGraphMor G₁ G₂ → TermGraphMor G₂ G₃ → TermGraphMor G₁ G₃
comp {G₁} {G₂} {G₃} F G = record
  {f-N = F.f-N ⨟ G.f-N
  ; f-V = F.f-V ⨟ (G.f-N ⊕ G.f-V) ⨟ ⊕.join
  ; prop-nLab = ≈-begin
      (F.f-N ⨟ G.f-N) ⨟ G₃.nLab
    ≈⟨ ⨟-assoc ⟨≈≈⟩ ⨟-cong₂ G.prop-nLab ⟩
      F.f-N ⨟ G₂.nLab
    ≈⟨ F.prop-nLab ⟩
      G₁.nLab
    □
  ; prop-suc = ≈-begin
      (F.f-N ⨟ G.f-N) ⨟ G₃.suc
    ≈⟨ ⨟-cong₁₂&₂ G.prop-suc ⟩
      (F.f-N ⨟ G₂.suc) ⨟ ℒ.mor ((G.f-N ⊕ G.f-V) ⨟ ⊕.join)
    ≈⟨ ⨟-cong₁ F.prop-suc ⟨≈≈⟩ ⨟-assoc ⟨≈≈˘⟩ ⨟-cong₂ ℒ.mor-⨟ ⟩
      G₁.suc ⨟ ℒ.mor (((F.f-N ⊕ F.f-V) ⨟ ⊕.join) ⨟ (G.f-N ⊕ G.f-V) ⨟ ⊕.join)
    ≈⟨ ⨟-cong₂ (ℒ.mor-cong (⨟-₂₂assoc₁₂₁ ⟨≈≈⟩ ⨟-cong₂ (⨟-cong₁ ⊕.FM-join ⟨≈≈⟩ ⨟-assoc))) ⟩
      G₁.suc ⨟ ℒ.mor ((F.f-N ⊕ F.f-V) ⨟ (G.f-N ⊕ (G.f-N ⊕ G.f-V)) ⨟ ⊕.join ⨟ ⊕.join)
    ≈⟨ ⨟-cong₂ (ℒ.mor-cong (⨟-cong₂ (⨟-cong₂ ⊕.assoc ⟨≈≈⟩ ⨟-assocL ⟨≈≈˘⟩ ⨟-cong₁ ⊕.mor-₁⨟))) ⟩
      G₁.suc ⨟ ℒ.mor ((F.f-N ⊕ F.f-V) ⨟ (G.f-N ⊕ ((G.f-N ⊕ G.f-V) ⨟ ⊕.join)) ⨟ ⊕.join)
    ≈⟨ ⨟-cong₂ (ℒ.mor-cong (⨟-assocL ⟨≈≈˘⟩ ⨟-cong₁ ⨟-⊕-⨟)) ⟩
      G₁.suc ⨟ ℒ.mor (((F.f-N ⨟ G.f-N) ⊕ (F.f-V ⨟ (G.f-N ⊕ G.f-V) ⨟ ⊕.join)) ⨟ ⊕.join)
    □
  }
  where
    module G₁ = TermGraph G₁
    module G₂ = TermGraph G₂
    module G₃ = TermGraph G₃
    module F = TermGraphMor F
    module G = TermGraphMor G

comp-assoc : {G₁ G₂ G₃ G₄ : TermGraph} {F : TermGraphMor G₁ G₂} {G : TermGraphMor G₂ G₃} {H : TermGraphMor G₃ G₄}
             → comp (comp F G) H ≈₁ comp F (comp G H)
comp-assoc {G₁} {G₂} {G₃} {G₄} {F} {G} {H} = record
  {f-N≋ = ⨟-assoc
  ; f-V≋ = ≈-begin
      (F.f-V ⨟ (G.f-N ⊕ G.f-V) ⨟ ⊕.join) ⨟ (H.f-N ⊕ H.f-V) ⨟ ⊕.join
    ≈⟨ ⨟-assoc₃₊₁ ⟨≈≈⟩ ⨟-cong₂₂ (⨟-cong₁&₂₁ ⊕.FM-join) ⟩
      F.f-V ⨟ (G.f-N ⊕ G.f-V) ⨟ (H.f-N ⊕ (H.f-N ⊕ H.f-V)) ⨟ ⊕.join ⨟ ⊕.join
    ≈⟨ ⨟-cong₂ (⨟-assocL ⟨≈≈⟩ ⨟-cong (≈-sym ⨟-⊕-⨟) ⊕.assoc) ⟩
      F.f-V ⨟ ((G.f-N ⨟ H.f-N) ⊕ (G.f-V ⨟ (H.f-N ⊕ H.f-V))) ⨟ (Id ⊕ ⊕.join) ⨟ ⊕.join
    ≈⟨ ⨟-cong₂ (⨟-assocL ⟨≈≈⟩ ⨟-cong₁ (⊕.mor-₁⨟ ⟨≈˘≈⟩ ⊕-cong₂ ⨟-assoc)) ⟩
      F.f-V ⨟ ((G.f-N ⨟ H.f-N) ⊕ (G.f-V ⨟ (H.f-N ⊕ H.f-V) ⨟ ⊕.join)) ⨟ ⊕.join
    □
  }
  where
    module F = TermGraphMor F
    module G = TermGraphMor G
    module H = TermGraphMor H

TermGraphCat : Category (ℓj ⊍ ℓk) ℓk TermGraph
TermGraphCat = record
```

```
  {semigroupoid = record
    {Hom = TermGraphHom
    ;compOp = record
      {_⨟_ = comp
      ;⨟-cong = λ Φ₁≋Φ₂ Ψ₁≋Ψ₂ → let open _≋_ in record
        {f-N≋ = ⨟-cong (f-N≋ Φ₁≋Φ₂) (f-N≋ Ψ₁≋Ψ₂)
        ;f-V≋ = ⨟-cong (f-V≋ Φ₁≋Φ₂) (⨟-cong₁ (⊕-cong (f-N≋ Ψ₁≋Ψ₂) (f-V≋ Ψ₁≋Ψ₂)))
        }
      ;⨟-assoc = comp-assoc
      }
    }
  ;idOp = record
    {Id = record
      {f-N = Id
      ;f-V = κ
      ;prop-nLab = leftId
      ;prop-suc = leftId ⟨≈≈˘⟩ (⨟-cong₂ (ℒ.mor-cong ⊕.rightUnit ⟨≈≈⟩ ℒ.mor-Id) ⟨≈≈⟩ rightId)
      }
    ;leftId = record {f-N≋ = leftId; f-V≋ = ⨟-cong₁&₂₁ ⊕.FM-return ⟨≈≈⟩ ⨟-cong₂ ⊕.leftUnit ⟨≈≈⟩ rightId}
    ;rightId = record {f-N≋ = rightId; f-V≋ = ⨟-cong₂ ⊕.rightUnit ⟨≈≈⟩ rightId}
    }
  }
```

6.2 Categoric.MonCoAlg.TG.FM

Definition of functor and monad to understand simple term graphs as an instance of Categoric.MonCoAlg.Obj (Sect. 4.1).

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import RATH.PropositionalEquality using (_≡_; ≡-refl)

open import Categoric.Category
open import Categoric.Category.FinLimits
open import Categoric.Category.FinColimits
open import Categoric.Functor
open import Categoric.Functor.Product
open import Categoric.Functor.Coproduct using (CoproductBifunctor)
open import Categoric.Product.Category
open import Categoric.Monad
open import Categoric.Monad.FunctorMonad
open import Categoric.Monad.FunctorMonad.Coproduct using (CoproductFM)
open import Categoric.Monad.FunctorMonad.Monad using (FM-Monad)
open import Categoric.Monad.Product

open CatFinLimits
open CatFinColimits
```

[WK: *We make the list monad $\mathcal{L}$ a parameter until we have a proper definition.* **]**

```
module Categoric.MonCoAlg.TG.FM
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (ℒ : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  (hasCoproducts₂ : HasCoproducts 𝒞₂)
  (L : Obj₂)   -- node labels
  where
```

```
open Category₂ 𝒞₂
open HasTerminalObject₂ 𝒞₂ hasTerminal₂
open HasProducts₂ 𝒞₂ hasProducts₂
open HasCoproducts₂ 𝒞₂ hasCoproducts₂

Obj₀ : Set i
Obj₀ = Obj₂ × Obj₂

𝒞₀ : Category j k Obj₀
𝒞₀ = ProductCategory 𝒞₂ 𝒞₂
module 𝒞₀ = Category 𝒞₀
module 𝒞₂ = Category 𝒞₂

⊕₂ = CoproductFM 𝒞₂ hasCoproducts₂
module ⊕₂ = FunctorMonad ⊕₂

⊗₂ : Functor 𝒞₀ 𝒞₂
⊗₂ = biFunctor (ProductBifunctor 𝒞₂ hasProducts₂)

module ℒ = Monad′ ℒ
open Category₀ 𝒞₀
```

We have, conceptually: $\mathcal{F}\ (\mathsf{N},\mathsf{S}) = ((\mathsf{L},\mathcal{L}\ \mathsf{S}),\mathbb{1})$

```
ℱ : Functor 𝒞₀ 𝒞₀
ℱ = (ProdFunctor (ConstFunctor {Src = 𝒞₂} {Trg = 𝒞₂} L) ℒ.M ⁂ ⊗₂)
      ▼ ConstFunctor {Src = 𝒞₀} {Trg = 𝒞₂} 𝕋₂
module ℱ = Functor ℱ


private
  ℱ-welldefined : {N S : Obj₂} → ℱ.obj (N,S) ≡ ((L ⊠₂ ℒ.obj S),𝕋₂)
  ℱ-welldefined = ≡-refl
```

We have, conceptually: $\mathcal{M}\ (\mathsf{N},\mathsf{V}) = \mathsf{N},(\mathsf{N} \boxplus \mathsf{V})$:

```
ℳ : Monad 𝒞₀
ℳ = FM-Monad 𝒞₂ 𝒞₂ (CoproductFM 𝒞₂ hasCoproducts₂)
module ℳ = Monad′ ℳ


private
  ℳ-welldefined : {N V : Obj₂} → ℳ.obj (N,V) ≡ (N,(N ⊞₂ V))
  ℳ-welldefined = ≡-refl
```

6.3 Categoric.MonCoAlg.TG.ObjCompat

Conversion between monadic coalgebras according to Categoric.MonCoAlg.Obj (Sect. 4.1) as instantiated for symbolically edge-attributed graphs in Categoric.MonCoAlg.TG.FM (Sect. 6.2) and the direct formalisation of Data.TermGraph (Sect. 6.1).

```
open import RATH.Level
open import RATH.Data.Product using (_,_)

open import Categoric.Category
open import Categoric.Category.FinColimits
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad

open CatFinLimits
open CatFinColimits
```

```
module Categoric.MonCoAlg.TG.ObjCompat
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (ℒ : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  (hasCoproducts₂ : HasCoproducts 𝒞₂)
  (L : Obj₂)   -- node labels
  where
open import Data.TermGraph 𝒞₂ hasCoproducts₂ L ℒ
open import Categoric.MonCoAlg.TG.FM 𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
module TG-MonCoAlg where
  open import Categoric.MonCoAlg.Obj 𝒞₀ ℳ ℱ public
open TG-MonCoAlg public
open Category₂ 𝒞₂
open HasTerminalObject₂ 𝒞₂ hasTerminal₂
open HasProducts₂ 𝒞₂ hasProducts₂
open HasCoproducts₂ 𝒞₂ hasCoproducts₂


objTo : MonCoAlg → TermGraph
objTo (MCA (N, V) (opN, _)) = record
  {Node = N
  ; Var  = V
  ; nLab = opN ⨟₂ π₂
  ; suc  = opN ⨟₂ ρ₂
  }
  where


objFrom : TermGraph → MonCoAlg
objFrom G = record
  {Carrier = Node, Var
  ; op     = (nLab ▽₂ suc), Ⓣ₂ {Var}
  }
  where
    open TermGraph G


objFromTo : {G : TermGraph} → Category.Iso TermGraphCat (objTo (objFrom G)) G
objFromTo {G} = record
  {isoMor = record
    {f-N = Id₂
    ; f-V = κ₂
    ; prop-nLab = leftId₂ ⟨≈₂≈˘⟩ ▽₂⨟π₂
    ; prop-suc = leftId₂ ⟨≈₂≈˘⟩
                 (𝒞₂.⨟-cong ▽₂⨟ρ₂ (ℒ.mor-cong ⊕₂.rightUnit ⟨≈₂≈⟩ ℒ.mor-Id) ⟨≈₂≈⟩ rightId₂)
    }
  ; isIso = record
    {_⁻¹ = record
      {f-N = Id₂
      ; f-V = κ₂
      ; prop-nLab = leftId₂ ⟨≈₂≈⟩ ▽₂⨟π₂
      ; prop-suc = leftId₂ ⟨≈₂≈⟩ ▽₂⨟ρ₂ ⟨≈₂≈˘⟩
                   (𝒞₂.⨟-cong₂ (ℒ.mor-cong ⊕₂.rightUnit ⟨≈₂≈⟩ ℒ.mor-Id) ⟨≈₂≈⟩ rightId₂)
      }
    ; rightInverse = record
      {f-N≋ = leftId₂
      ; f-V≋ = 𝒞₂.⨟-cong₁&₂₁ κ⨟◬₂ ⟨≈₂≈⟩ 𝒞₂.⨟-cong₂ κ⨟◬₂ ⟨≈₂≈⟩ rightId₂
      }
    ; leftInverse = record
```

```
      {f-N≋ = leftId₂
      ;f-V≋ = C₂.⨟-cong₁&₂₁ κ⨟⧋₂ ⟨≈₂≈⟩ C₂.⨟-cong₂ κ⨟⧋₂ ⟨≈₂≈⟩ rightId₂
      }
    }
  }
  where
    module G = TermGraph G


objToFrom : {A : MonCoAlg}
          → Category.Iso M.KleisliCat (MonCoAlg.Carrier (objFrom (objTo A))) (MonCoAlg.Carrier A)
objToFrom {MCA (N, V) (opN, _)} = record
  {isoMor = Id₂ {N}, κ₂
  ;isIso = record
    {_⁻¹ = Id₂ {N}, κ₂
    ;rightInverse = (leftId₂ ⟨≈₂≈⟩ leftId₂)
                  , (C₂.⨟-cong₁&₂₁ κ⨟⧋₂ ⟨≈₂≈⟩ C₂.⨟-cong₂ κ⨟⧋₂ ⟨≈₂≈⟩ rightId₂)
    ;leftInverse  = (leftId₂ ⟨≈₂≈⟩ leftId₂)
                  , (C₂.⨟-cong₁&₂₁ κ⨟⧋₂ ⟨≈₂≈⟩ C₂.⨟-cong₂ κ⨟⧋₂ ⟨≈₂≈⟩ rightId₂)
    }
  }
```

6.4 Categoric.MonCoAlg.TG.FMProps

We collect potentially useful properties of $\mathcal{F}$ and $\mathcal{M}$ as defined in Categoric.MonCoAlg.TG.FM (Sect. 6.2).

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
open import Categoric.Semigroupoid
open import Categoric.IdOp
open import Categoric.Category
open import Categoric.Category.FinColimits
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Functor.Product
open import Categoric.Product.Category
open import Categoric.Monad
open import Categoric.Monad.Product
open CatFinLimits
open CatFinColimits


module Categoric.MonCoAlg.TG.FMProps
  {i j k : Level} {Obj₂ : Set i} (C₂ : Category {i} j k Obj₂) (L : Monad C₂)
  (hasTerminal₂ : HasTerminalObject C₂)
  (hasProducts₂ : HasProducts C₂)
  (hasCoproducts₂ : HasCoproducts C₂)
  (L : Obj₂)   -- node labels
  where
open import Categoric.MonCoAlg.TG.FM C₂ L hasTerminal₂ hasProducts₂ hasCoproducts₂ L
open Category₀ C₀
open Category₂ C₂
open HasTerminalObject₂ C₂ hasTerminal₂
open HasProducts₂ C₂ hasProducts₂
open HasCoproducts₂ C₂ hasCoproducts₂
```

```
distrJoin₀ : {A : Obj₀} → 𝒞₀.Mor (ℳ.obj (ℱ.obj (ℳ.obj A))) (ℱ.obj (ℳ.obj A))
distrJoin₀ {N₁, V₁} = Id₂, Ⓣ₂

distrJoin : NatTrans (ℳ.M ⁘ ℱ ⁘ ℳ.M) (ℳ.M ⁘ ℱ)
distrJoin = record {indmor = λ {A} → distrJoin₀ {A}
  ; naturality = λ {{N₀, V₀} {N₁, V₁} {fN, fV}
                  → (𝒞₂.rightId ⟨≈₂˘⟩ 𝒞₂.leftId), Ⓣ-₂≈}}
open NatTrans distrJoin public using () renaming (naturality to distrJoin₀-naturality)

return-distrJoin₀ : {A : Obj₀}
                  → ℳ.return {ℱ.obj (ℳ.obj A)} ⨟₀ distrJoin₀ {A}
                  ≈₀ Id₀ {ℱ.obj (ℳ.obj A)}
return-distrJoin₀ {N, V} = leftId₂, Ⓣ-₂≈

join-distrJoin₀ : {A : Obj₀}
                → ℳ.join ⨟₀ distrJoin₀ {A}
                ≈₀ ℳ.mor (distrJoin₀ {A}) ⨟₀ distrJoin₀ {A}
join-distrJoin₀ {N, V} = ≈₂-refl, Ⓣ-₂≈

join↗distrJoin₀ : {A : Obj₀}
                → ℳ.mor (ℱ.mor (ℳ.join {A})) ⨟₀ distrJoin₀ {A}
                ≈₀ distrJoin₀ {ℳ.obj A} ⨟₀ ℱ.mor (ℳ.join {A})
join↗distrJoin₀ {N, V} = (𝒞₂.rightId ⟨≈₂˘⟩ 𝒞₂.leftId), Ⓣ-₂≈


distr : NatTrans (ℱ ⁘ ℳ.M) (ℳ.M ⁘ ℱ)
distr = record
  {indmor = λ {{N, V} → (Id₂ {L} ⊗₂ ℒ.mor κ₂
                        , Ⓣ₂
                        )}
  ; naturality = λ {{N₀, V₀} {N₁, V₁} {fN, fV}
                → (≈₂-begin
                     (Id₂ {L} ⊗₂ ℒ.mor fV) ⨟₂ (Id₂ {L} ⊗₂ ℒ.mor κ₂)
                   ≈₂⟨ 𝒫₂.⨟-⊗-⨟ ⟨≈₂˘≈⟩ 𝒫₂.⊗-cong leftId₂ (≈₂-sym ℒ.mor-⨟) ⟩
                     Id₂ {L} ⊗₂ ℒ.mor (fV ⨟₂ κ₂)
                   ≈₂˘⟨ 𝒫₂.⨟-⊗-⨟ ⟨≈₂˘≈⟩ 𝒫₂.⊗-cong leftId₂ (ℒ.mor-⨟ ⟨≈₂˘≈⟩ ℒ.mor-cong 𝒫₂.κ⨟⊕) ⟩
                     (Id₂ {L} ⊗₂ ℒ.mor κ₂) ⨟₂ (Id₂ {L} ⊗₂ ℒ.mor (fN ⊕₂ fV))
                   □₂)
                , Ⓣ-₂≈
                }
  }
open NatTrans distr public using () renaming
  (indmor to distr₀; naturality to distr-naturality)


undistr″ : NatTrans (ℳ.M ⁘ ℳ.M ⁘ ℱ) (ℳ.M ⁘ ℱ ⁘ ℳ.M)
undistr″ = record
  {indmor = λ {{N, V} → Id₂ {L} ⊗₂ ℒ.mor ⨁₂.join, κ₂
                      }
  ; naturality = λ {{N₀, V₀} {N₁, V₁} {fN, fV}
                → (≈₂-begin
                     (Id₂ ⊗₂ ℒ.mor (fN ⊕₂ (fN ⊕₂ fV))) ⨟₂ (Id₂ ⊗₂ ℒ.mor ⨁₂.join)
                   ≈₂⟨ 𝒫₂.⨟-⊗-⨟ ⟨≈₂˘≈˘⟩ 𝒫₂.⊗-cong₂ (ℒ.mor-⨟ ⟨≈₂˘≈˘⟩ ℒ.mor-cong ⨁₂.FM-join) ⟩
                     (Id₂ ⨟₂ Id₂) ⊗₂ ℒ.mor (⨁₂.join ⨟₂ (fN ⊕₂ fV))
                   ≈₂⟨ 𝒫₂.⊗-cong₂ ℒ.mor-⨟ ⟨≈₂≈⟩ 𝒫₂.⨟-⊗-⨟ ⟩
                     (Id₂ ⊗₂ ℒ.mor ⨁₂.join) ⨟₂ (Id₂ ⊗₂ ℒ.mor (fN ⊕₂ fV))
                   □₂
                   ), ≈₂-sym κ⨟⨪₂
                }
  }
open NatTrans undistr″ public using () renaming
  (indmor to undistr″₀; naturality to undistr″-naturality)
```

6.5 Categoric.MonCoAlg.TG.Cat2

We now produce the MonCoAlg category from $\mathcal{F}$ and $\mathcal{M}$ as defined in Categoric.MonCoAlg.TG.FM (Sect. 6.2).

```
open import RATH.Level

open import Categoric.Category
open import Categoric.Category.FinColimits
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad

open CatFinLimits
open CatFinColimits


module Categoric.MonCoAlg.TG.Cat2
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (ℒ : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  (hasCoproducts₂ : HasCoproducts 𝒞₂)
  (L : Obj₂)   -- node labels
  where
open import Categoric.MonCoAlg.TG.FM          𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (𝒞₀; ℳ; ℱ)
open import Categoric.MonCoAlg.TG.ObjCompat 𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (module TG-MonCoAlg)
open import Categoric.MonCoAlg.TG.FMProps    𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (distrJoin; return-distrJoin₀; join-distrJoin₀; join↗distrJoin₀)


module TG-MonCoAlg-Cat2 where
  open import Categoric.MonCoAlg.Cat2 𝒞₀ ℳ ℱ distrJoin
    (λ {A} → return-distrJoin₀ {A})
    (λ {A} → join-distrJoin₀ {A})
    (λ {A} → join↗distrJoin₀ {A})
    public using (MCACategory; MCAMor; MkMCAMor; module MCAMor; _↼⨟_-distrJoin)
  open TG-MonCoAlg public
open TG-MonCoAlg-Cat2 public
```

6.6 Categoric.MonCoAlg.TG.Cat2.MorCompat

We establish the lowest layer of the correspondence between TermGraph morphisms and TG-MonCoAlg-Cat2 morphisms.

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

open import Categoric.Category
open import Categoric.Category.FinColimits
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad

open CatFinLimits
open CatFinColimits


module Categoric.MonCoAlg.TG.Cat2.MorCompat
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (ℒ : Monad 𝒞₂)
```

```
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  (hasCoproducts₂ : HasCoproducts 𝒞₂)
  (L : Obj₂)   -- node labels
  where
open import Data.TermGraph 𝒞₂ hasCoproducts₂ L ℒ
open import Categoric.MonCoAlg.TG.FM          𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
    -- using (module ℳ)
open import Categoric.MonCoAlg.TG.ObjCompat 𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (objFrom; objTo; objToFrom)
open import Categoric.MonCoAlg.TG.Cat2        𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (MonCoAlg; module MonCoAlg; MCAMor; MkMCAMor; MCACategory)
open Category₂ 𝒞₂
open HasTerminalObject₂ 𝒞₂ hasTerminal₂
open HasProducts₂ 𝒞₂ hasProducts₂
open HasCoproducts₂ 𝒞₂ hasCoproducts₂
private
  module 𝒫₂′ = HasProducts 𝒞₂ hasProducts₂   -- for Category.FinColimits material


morFrom : {G₁ G₂ : TermGraph} → TermGraphMor G₁ G₂ → MCAMor (objFrom G₁) (objFrom G₂)
morFrom {G₁} {G₂} F = record
  {mor = f-N, f-V
  ;commutes = (≈₂-begin
                 f-N ⨟₂ (G₂.nLab ▽₂ G₂.suc) ⨟₂ Id₂
               ≈₂⟨ 𝒞₂.⨟-cong₂ rightId₂ ⟨≈₂≈⟩ ⨟-▽₂ ⟩
                 (f-N ⨟₂ G₂.nLab) ▽₂ (f-N ⨟₂ G₂.suc)
               ≈₂⟨ ▽₂-cong prop-nLab prop-suc ⟩
                 G₁.nLab ▽₂ (G₁.suc ⨟₂ ℒ.mor ((f-N ⊕₂ f-V) ⨟₂ ⨁₂.join))
               ≈₂˘⟨ 𝒫₂.▽-⨟-⊗ ⟨≈₂≈⟩ ▽₂-cong₁ rightId₂ ⟩
                 (G₁.nLab ▽₂ G₁.suc) ⨟₂ (Id₂ ⊗₂ ℒ.mor ((f-N ⊕₂ f-V) ⨟₂ ⨁₂.join))
               □₂)
               , Ⓣ ₂≈
  }
  where
    module G₁ = TermGraph G₁
    module G₂ = TermGraph G₂
    open TermGraphMor F


morTo : {A₁ A₂ : MonCoAlg} → MCAMor A₁ A₂ → TermGraphMor (objTo A₁) (objTo A₂)
morTo {A₁} {A₂} (MkMCAMor (f-N, f-V) (commutesN, _)) = record
  {f-N = f-N
  ;f-V = f-V
  ;prop-nLab = ≈₂-begin
      f-N ⨟₂ opN₂ ⨟₂ π₂
    ≈₂⟨ 𝒞₂.⨟-cong₂₁ rightId₂ ⟨≈₂˘≈⟩ 𝒞₂.⨟-assocL ⟩
      (f-N ⨟₂ opN₂ ⨟₂ Id₂) ⨟₂ π₂
    ≈₂⟨ 𝒞₂.⨟-cong₁ commutesN ⟨≈₂≈⟩ 𝒞₂.⨟-assoc ⟨≈₂≈⟩ 𝒞₂.⨟-cong₂ (𝒫₂.⊗⨟π ⟨≈₂≈⟩ rightId₂) ⟩
      opN₁ ⨟₂ π₂
    □₂
  ;prop-suc = ≈₂-begin
      f-N ⨟₂ opN₂ ⨟₂ ρ₂
    ≈₂⟨ 𝒞₂.⨟-cong₂₁ rightId₂ ⟨≈₂˘≈⟩ 𝒞₂.⨟-assocL ⟩
      (f-N ⨟₂ opN₂ ⨟₂ Id₂) ⨟₂ ρ₂
    ≈₂⟨ 𝒞₂.⨟-cong₁ commutesN ⟨≈₂≈⟩ 𝒞₂.⨟-assoc ⟨≈₂≈⟩ 𝒞₂.⨟-cong₂ 𝒫₂.⊗⨟ρ ⟨≈₂≈⟩ 𝒞₂.⨟-assocL ⟩
      (opN₁ ⨟₂ ρ₂) ⨟₂ ℒ.mor ((f-N ⊕₂ f-V) ⨟₂ ⨁₂.join)
    □₂
  }
```

```
  where
    opN₁ = proj₁ (MonCoAlg.op A₁)
    opN₂ = proj₁ (MonCoAlg.op A₂)


objToFrom′ : {A : MonCoAlg}
             → Category.Iso MCACategory (objFrom (objTo A)) A
objToFrom′ {A} = record
  {isoMor = MkMCAMor (𝒞₃.isoMor (objToFrom {A}))
      (  (leftId₂ ⟨≈₂≈˘⟩ 𝒞₂.⨟-cong (≈₂-sym to-∇₂)
           (𝒫₂.⊗-cong₂ (ℒ.mor-cong ⊕₂.rightUnit ⟨≈₂≈⟩ ℒ.mor-Id) ⟨≈₂≈⟩ 𝒫₂′.⊗-Id))
      , 𝟙 ₂≈)
  ; isIso = record
    {_⁻¹ = MkMCAMor (𝒞₃._⁻¹ (objToFrom {A}))
      (((leftId₂ ⟨≈₂≈˘⟩ 𝒞₂.⨟-cong₁ to-∇₂)
        ⟨≈₂≈˘⟩ 𝒞₂.⨟-cong₂ (𝒫₂.⊗-cong₂ (ℒ.mor-cong ⊕₂.rightUnit ⟨≈₂≈⟩ ℒ.mor-Id) ⟨≈₂≈⟩ 𝒫₂′.⊗-Id)
      ), 𝟙 ₂≈)
    ; rightInverse = 𝒞₃.rightInverse (objToFrom {A})
    ; leftInverse = 𝒞₃.leftInverse (objToFrom {A})
    }
  }
  where
    module A = MonCoAlg A
    open Category₃ 𝓜.KleisliCat
    module 𝒞₃ = Category 𝓜.KleisliCat
```

6.7 Categoric.MonCoAlg.TG.Cat2.To

We produce the functor from the TG-MonCoAlg-Cat2 category to the TermGraph category.

```
open import RATH.Level
open import RATH.Data.Product using (_,_)
open import Categoric.Category
open import Categoric.Category.FinColimits
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad
open CatFinLimits
open CatFinColimits


module Categoric.MonCoAlg.TG.Cat2.To
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (ℒ : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  (hasCoproducts₂ : HasCoproducts 𝒞₂)
  (L : Obj₂)  -- node labels
  where
open import Data.TermGraph 𝒞₂ hasCoproducts₂ L ℒ
open import Categoric.MonCoAlg.TG.ObjCompat      𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (objTo)
open import Categoric.MonCoAlg.TG.Cat2           𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (MCACategory)
open import Categoric.MonCoAlg.TG.Cat2.MorCompat 𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (morTo)
open Category₂ 𝒞₂
```

```
private
  module C₂ = Category C₂


To : Functor MCACategory TermGraphCat
To = record
  {obj = objTo
  ; mor = morTo
  ; mor-cong = λ {(≈N, ≈V) → record {f-N≋ = ≈N; f-V≋ = ≈V}}
  ; mor-⨟ = record {f-N≋ = C₂.⨟-cong₂ rightId₂; f-V≋ = ≈₂-refl}
  ; mor-Id = record {f-N≋ = ≈₂-refl; f-V≋ = ≈₂-refl}
  }
```

6.8 Categoric.MonCoAlg.TG.Cat2.From

We produce the functor from the TermGraph category to the TG-MonCoAlg-Cat2 category.

```
open import RATH.Level
open import RATH.Data.Product using (_,_)
open import Categoric.Category
open import Categoric.Category.FinColimits
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad
open CatFinLimits
open CatFinColimits


module Categoric.MonCoAlg.TG.Cat2.From
  {i j k : Level} {Obj₂ : Set i} (C₂ : Category {i} j k Obj₂) (L : Monad C₂)
  (hasTerminal₂ : HasTerminalObject C₂)
  (hasProducts₂ : HasProducts C₂)
  (hasCoproducts₂ : HasCoproducts C₂)
  (L : Obj₂)   -- node labels
  where
open import Data.TermGraph C₂ hasCoproducts₂ L L
open import Categoric.MonCoAlg.TG.ObjCompat       C₂ L hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (objFrom)
open import Categoric.MonCoAlg.TG.Cat2            C₂ L hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (MCACategory)
open import Categoric.MonCoAlg.TG.Cat2.MorCompat C₂ L hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (morFrom)
open Category₂ C₂
private
  module C₂ = Category C₂


From : Functor TermGraphCat MCACategory
From = record
  {obj = objFrom
  ; mor = morFrom
  ; mor-cong = λ F₁≈F₂ → let open _≋_ F₁≈F₂ in f-N≋, f-V≋
  ; mor-⨟    = ≈₂-sym (C₂.⨟-cong₂ rightId₂), ≈₂-refl
  ; mor-Id   = ≈₂-refl, ≈₂-refl
  }
```

6.9 Categoric.MonCoAlg.TG.Cat2.ToFrom

We show one side of the equivalence between the TermGraph category and the corresponding MonCoAlg category.

```
open import RATH.Level
open import RATH.Data.Product using (_,_)

open import Categoric.Category
open import Categoric.Category.FinColimits
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad

open CatFinLimits
open CatFinColimits


module Categoric.MonCoAlg.TG.Cat2.ToFrom
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (ℒ : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  (hasCoproducts₂ : HasCoproducts 𝒞₂)
  (L : Obj₂)   -- node labels
  where
open import Categoric.MonCoAlg.TG.FM                𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (module ⊕₂)
open import Categoric.MonCoAlg.TG.Cat2.From          𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (From)
open import Categoric.MonCoAlg.TG.Cat2.To            𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (To)
open import Categoric.MonCoAlg.TG.Cat2.MorCompat 𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (objToFrom′)
open import Categoric.MonCoAlg.TG.Cat2               𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (MCACategory)

open Category₂ 𝒞₂
open HasTerminalObject₂ 𝒞₂ hasTerminal₂
open HasProducts₂ 𝒞₂ hasProducts₂
open HasCoproducts₂ 𝒞₂ hasCoproducts₂
private
  module 𝒞₂ = Category 𝒞₂
  module ℒ = Monad′ ℒ


ToFrom : NatIso (To ⨟⨟ From) (Identity MCACategory)
ToFrom = IsoNat objToFrom′
  ((𝒞₂.⨟-cong₂ rightId₂ ⟨≈₂≈˘⟩ leftId₂)
  , (𝒞₂.⨟-cong₂ ⊕₂.rightUnit ⟨≈₂≈˘⟩ (𝒞₂.⨟-cong₁&₂₁ κ⨟△₂ ⟨≈₂≈⟩ 𝒞₂.⨟-cong₂ κ⨟△₂))
  )
```

6.10 Categoric.MonCoAlg.TG.Cat2.FromTo

We show one side of the equivalence between the TermGraph category and the corresponding MonCoAlg category.

```
open import RATH.Level

open import Categoric.Category
open import Categoric.Category.FinColimits using (module CatFinColimits)
open import Categoric.Category.FinLimits using (module CatFinLimits)
open import Categoric.Functor using (NatIso; IsoNat; _⨟⨟_; Identity)
```

```
open import Categoric.Monad using (Monad)
open CatFinLimits
open CatFinColimits


module Categoric.MonCoAlg.TG.Cat2.FromTo
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (ℒ : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  (hasCoproducts₂ : HasCoproducts 𝒞₂)
  (L : Obj₂)  -- node labels
  where
open import Data.TermGraph 𝒞₂ hasCoproducts₂ L ℒ using (TermGraphCat)
open import Categoric.MonCoAlg.TG.FM          𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (module ⊕₂)
open import Categoric.MonCoAlg.TG.ObjCompat 𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (objFromTo)
open import Categoric.MonCoAlg.TG.Cat2.From  𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (From)
open import Categoric.MonCoAlg.TG.Cat2.To    𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (To)
open Category₂ 𝒞₂ using (leftId₂; rightId₂; _⟨≈₂≈ˇ⟩_; _⟨≈₂≈⟩_)
open HasCoproducts₂ 𝒞₂ hasCoproducts₂ using (κ⨟△₂)
private
  module 𝒞₂ = Category 𝒞₂


FromTo : NatIso (From ⨟⨟ To) (Identity TermGraphCat)
FromTo = IsoNat objFromTo (record
  {f-N≋ = rightId₂ ⟨≈₂≈ˇ⟩ leftId₂
  ;f-V≋ = 𝒞₂.⨟-cong₂ ⊕₂.rightUnit ⟨≈₂≈ˇ⟩ (𝒞₂.⨟-cong₁&₂₁ κ⨟△₂ ⟨≈₂≈⟩ 𝒞₂.⨟-cong₂ κ⨟△₂)
  })
```

6.11 Categoric.MonCoAlg.TG.Cat2.FromToFunctor

```
open import RATH.Level
open import Categoric.Category
open import Categoric.Category.FinColimits using (module CatFinColimits)
open import Categoric.Category.FinLimits using (module CatFinLimits)
open import Categoric.Functor using (Functor; _⨟⨟_)
open import Categoric.Monad using (Monad)
open CatFinLimits
open CatFinColimits


module Categoric.MonCoAlg.TG.Cat2.FromToFunctor
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (ℒ : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  (hasCoproducts₂ : HasCoproducts 𝒞₂)
  (L : Obj₂)  -- node labels
  where
open import Data.TermGraph 𝒞₂ hasCoproducts₂ L ℒ using (TermGraphCat)
open import Categoric.MonCoAlg.TG.Cat2.From 𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
```

```
  using (From)
open import Categoric.MonCoAlg.TG.Cat2.To    𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (To)


FromToFunctor : Functor TermGraphCat TermGraphCat
FromToFunctor = From ⨟⨟ To
```

6.12 Categoric.MonCoAlg.TG.Cat2.FromToNaturality

We show naturality for objFromTo.

```
open import RATH.Level
open import Categoric.Category
open import Categoric.Category.FinColimits using (module CatFinColimits)
open import Categoric.Category.FinLimits using (module CatFinLimits)
open import Categoric.Functor using (module Functor)
open import Categoric.Monad using (Monad)
open CatFinLimits
open CatFinColimits


module Categoric.MonCoAlg.TG.Cat2.FromToNaturality
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (ℒ : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  (hasCoproducts₂ : HasCoproducts 𝒞₂)
  (L : Obj₂)   -- node labels
  where
open import Data.TermGraph 𝒞₂ hasCoproducts₂ L ℒ
  using (TermGraphCat; TermGraph; TermGraphMor)
open import Categoric.MonCoAlg.TG.FM          𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (module ⊕₂)
open import Categoric.MonCoAlg.TG.ObjCompat 𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (objFromTo)
open import Categoric.MonCoAlg.TG.Cat2.FromToFunctor
                                                𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L
  using (FromToFunctor)
open Category₂ 𝒞₂ using (leftId₂; rightId₂; _⟨≈₂≈˘⟩_; _⟨≈₂≈⟩_)
open Category₄ TermGraphCat
open HasCoproducts₂ 𝒞₂ hasCoproducts₂ using (κ⨟△₂)
private
  module 𝒞₂ = Category 𝒞₂
  module 𝒞₄ = Category TermGraphCat
  module FromToF = Functor FromToFunctor


objFromTo-naturality : {A B : TermGraph} {f : TermGraphMor A B}
                     → FromToF.mor f ⨟₄ 𝒞₄.isoMor (objFromTo {B})
                     ≈₄ 𝒞₄.isoMor (objFromTo {A}) ⨟₄ f
objFromTo-naturality = record
  {f-N≋ = rightId₂ ⟨≈₂≈˘⟩ leftId₂
  ;f-V≋ = 𝒞₂.⨟-cong₂ ⊕₂.rightUnit ⟨≈₂≈˘⟩ (𝒞₂.⨟-cong₁&₂₁ κ⨟△₂ ⟨≈₂≈⟩ 𝒞₂.⨟-cong₂ κ⨟△₂)
  }
```

6.13 Categoric.MonCoAlg.TG.Cat2.FromTo1

We show one side of the equivalence between the TermGraph category and the corresponding MonCoAlg category.

```
open import RATH.Level

open import Categoric.Category
open import Categoric.Category.FinColimits using (module CatFinColimits)
open import Categoric.Category.FinLimits using (module CatFinLimits)
open import Categoric.Functor using (NatIso; IsoNat; Identity)
open import Categoric.Monad using (Monad)

open CatFinLimits
open CatFinColimits


module Categoric.MonCoAlg.TG.Cat2.FromTo1
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (ℒ : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  (hasCoproducts₂ : HasCoproducts 𝒞₂)
  (L : Obj₂)   -- node labels
  where

open import Data.TermGraph 𝒞₂ hasCoproducts₂ L ℒ using (TermGraphCat)

open import Categoric.MonCoAlg.TG.ObjCompat
               𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L using (objFromTo)
open import Categoric.MonCoAlg.TG.Cat2.FromToFunctor
               𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L using (FromToFunctor)
open import Categoric.MonCoAlg.TG.Cat2.FromToNaturality
               𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L using (objFromTo-naturality)


FromTo : NatIso FromToFunctor (Identity TermGraphCat)
FromTo = IsoNat objFromTo objFromTo-naturality
```

6.14 Categoric.MonCoAlg.TG.Cat2.Equiv

We collect all of the equivalence between the TermGraph category and the TG-MonCoAlg-Cat2 category into a single proof.

```
open import RATH.Level
open import RATH.Data.Product using (_,_)

open import Categoric.Category
open import Categoric.Category.FinLimits
open import Categoric.Category.FinColimits
open import Categoric.Functor
open import Categoric.Functor.Equivalence
open import Categoric.Monad

open CatFinLimits
open CatFinColimits


module Categoric.MonCoAlg.TG.Cat2.Equiv
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (ℒ : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  (hasCoproducts₂ : HasCoproducts 𝒞₂)
```

```
  (L : Obj₂)  -- node labels
  where
open import Data.TermGraph 𝒞₂ hasCoproducts₂ L ℒ using (TermGraphCat)
open import Categoric.MonCoAlg.TG.Cat2.From
                 𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L using (From)
open import Categoric.MonCoAlg.TG.Cat2.To
                 𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L using (To)
open import Categoric.MonCoAlg.TG.Cat2.FromTo1
                 𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L using (FromTo)
open import Categoric.MonCoAlg.TG.Cat2.ToFrom
                 𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L using (ToFrom)
open import Categoric.MonCoAlg.TG.Cat2
                 𝒞₂ ℒ hasTerminal₂ hasProducts₂ hasCoproducts₂ L using (MCACategory)


MonCoAlg-TermGraph-Equivalence : CatEquivalence MCACategory TermGraphCat
MonCoAlg-TermGraph-Equivalence = record
  {To     = To
  ;From   = From
  ;ToFrom = ToFrom
  ;FromTo = FromTo
  }
```

Chapter 7

Monadic Co-Algebras over a FunctorMonad

7.1 Categoric.MonCoAlg.FM.Obj

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import Categoric.Category
open import Categoric.Functor
open import Categoric.Monad.FunctorMonad
```

Here we investigate a specialised variant of monadic coalgebras based on a ProductCategory $\mathcal{C}_1$ $\mathcal{C}_2$, where the monad is only on $\mathcal{C}_2$, but parameterised over $\mathcal{C}_1$ as a FunctorMonad, and the functor goes only to $\mathcal{C}_1$, which means that there are no operations starting at $\mathcal{C}_2$ sorts.

Both SEAGraph and TermGraph are instances of this, but the setting is not sufficient for obtaining pushouts from underlying Kleisli category pushouts.

```
module Categoric.MonCoAlg.FM.Obj
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (𝒞₁ : Category {i₁} j₁ k₁ Obj₁)
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (𝒞₂ : Category {i₂} j₂ k₂ Obj₂)
  (ℳ : FunctorMonad {𝒞₁ = 𝒞₁} {𝒞₂ = 𝒞₂})
  (ℱ : Bifunctor 𝒞₁ 𝒞₂ 𝒞₁)
  where

open Category₁ 𝒞₁
open Category₂ 𝒞₂
private
  module 𝒞₁ = Category 𝒞₁
  module 𝒞₂ = Category 𝒞₂
  module ℳ = FunctorMonad ℳ
  module ℱ = Bifunctor ℱ

record MonProdCoAlg : Set (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂) where
  constructor MPCA
  field Carrier : ℳ.ObjPair
  open ℳ.ObjPair Carrier
  field op : Mor₁ Carrier₁ (ℱ.obj Carrier₁ Carrier′)
  open ℳ.ObjPair Carrier public
```

7.2 Categoric.MonCoAlg.FM.ObjCompat

```
open import RATH.Level
open import RATH.PropositionalEquality using (_≡_; ≡-refl)
```

```
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
open import Categoric.Semigroupoid
open import Categoric.IdOp
open import Categoric.Category
open import Categoric.Category.FinLimits
open import Categoric.Product.Category
open import Categoric.Functor
open import Categoric.Monad
open import Categoric.Monad.FunctorMonad
open CatFinLimits


module Categoric.MonCoAlg.FM.ObjCompat
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (𝒞₁ : Category {i₁} j₁ k₁ Obj₁)
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (𝒞₂ : Category {i₂} j₂ k₂ Obj₂)
  (ℳ : FunctorMonad {𝒞₁ = 𝒞₁} {𝒞₂ = 𝒞₂})
  (ℱ : Bifunctor 𝒞₁ 𝒞₂ 𝒞₁)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  where
open Category₁ 𝒞₁
open Category₂ 𝒞₂
open HasTerminalObject₂ 𝒞₂ hasTerminal₂
private
  module 𝒞₁ = Category 𝒞₁
  module 𝒞₂ = Category 𝒞₂
  module ℳ = FunctorMonad ℳ
  module ℱ = Bifunctor ℱ
open import Categoric.MonCoAlg.FM.Obj 𝒞₁ 𝒞₂ ℳ ℱ


Obj₀ : Set (i₁ ⊎ i₂)
Obj₀ = Obj₁ × Obj₂
𝒞₀ : Category (j₁ ⊎ j₂) (k₁ ⊎ k₂) Obj₀
𝒞₀ = ProductCategory 𝒞₁ 𝒞₂
fromObjPair : ℳ.ObjPair → Obj₀
fromObjPair p = ℳ.ObjPair.Carrier₁ p , ℳ.ObjPair.Carrier₂ p
M₀ : Functor 𝒞₀ 𝒞₀
M₀ = Proj₁ 𝒞₁ 𝒞₂ ▼ biFunctor ℳ.bifunctor
ℳ₀ : Monad 𝒞₀
ℳ₀ = record
  {M = M₀
  ;trafos = record
    {M-return = record
      {indmor = Id₁ , ℳ.return
      ;naturality = (𝒞₁.rightId ⟨≈₁≈˘⟩ 𝒞₁.leftId) , ≈₂-sym ℳ.FM-return
      }
    ;M-join = record
      {indmor = Id₁ , ℳ.join
      ;naturality = (𝒞₁.rightId ⟨≈₁≈˘⟩ 𝒞₁.leftId) , ≈₂-sym ℳ.FM-join
      }
    ;leftUnit  = 𝒞₁.leftId , ℳ.leftUnit
    ;rightUnit = 𝒞₁.leftId , ℳ.rightUnit
    ;assoc     = ≈₁-refl , ℳ.assoc
    }
  }
ℱ₀ : Functor 𝒞₀ 𝒞₀
ℱ₀ = biFunctor ℱ ▼ ConstFunctor {Src = 𝒞₀} {Trg = 𝒞₂} ⓘ₂
```

```
private
  module 𝒞₀ = Category 𝒞₀
  module ℳ₀ = Monad′ ℳ₀
  module ℱ₀ = Functor ℱ₀
open Category₀ 𝒞₀
open import Categoric.MonCoAlg.Obj 𝒞₀ ℳ₀ ℱ₀ public
open ℳ₀ using () renaming (↼ to ↼₀; _⨟∘_ to _⨟∘₀_)


objTo : MonCoAlg → MonProdCoAlg
objTo (MCA A (opA, _)) = record
  {Carrier = A
  ;op = opA
  }


objFrom : MonProdCoAlg → MonCoAlg
objFrom (MPCA A opA) = record
  {Carrier = A
  ;op = opA, Ⓣ₂
  }


objFromTo : {G : MonProdCoAlg} → (objTo (objFrom G)) ≡ G
objFromTo = ≡-refl
```

For the converse composition, we obtain only morphism equality of the second component of MonCoAlg.op, due to $\approx Ⓣ_2$.

7.3 Categoric.MonCoAlg.FM.Cat

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
open import Categoric.Semigroupoid
open import Categoric.IdOp
open import Categoric.Category
open import Categoric.Functor
open import Categoric.Monad
open import Categoric.Monad.FunctorMonad


module Categoric.MonCoAlg.FM.Cat
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (𝒞₁ : Category {i₁} j₁ k₁ Obj₁)
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (𝒞₂ : Category {i₂} j₂ k₂ Obj₂)
  (ℳ : FunctorMonad {𝒞₁ = 𝒞₁} {𝒞₂ = 𝒞₂})
  (ℱ : Bifunctor 𝒞₁ 𝒞₂ 𝒞₁)
  where
open Category₁ 𝒞₁
open Category₂ 𝒞₂
private
  module 𝒞₁ = Category 𝒞₁
  module 𝒞₂ = Category 𝒞₂
  module ℳ = FunctorMonad ℳ
  module ℱ = Bifunctor ℱ
open import Categoric.MonCoAlg.FM.Obj 𝒞₁ 𝒞₂ ℳ ℱ
```

```
record MPCAMor (A B : MonProdCoAlg) : Set (j₁ ⊎ j₂ ⊎ k₁ ⊎ k₂) where
  open module A = MonProdCoAlg A using () renaming (Carrier₁ to A₁; Carrier₂ to A₂; Carrier′ to A′)
  open module B = MonProdCoAlg B using () renaming (Carrier₁ to B₁; Carrier₂ to B₂; Carrier′ to B′)
  field mor : 𝓜.FMMor A.Carrier B.Carrier
  open 𝓜.FMMor mor
  field commutes : mor₁ ⨟₁ B.op ≈₁ A.op ⨟₁ 𝓕.mor mor₁ mor′
  open 𝓜.FMMor mor public


module MPCAMor-Comp {A B C : MonProdCoAlg} (F : MPCAMor A B) (G : MPCAMor B C) where
  private
    module A = MonProdCoAlg A
    module B = MonProdCoAlg B
    module C = MonProdCoAlg C
    module F = MPCAMor F
    module G = MPCAMor G
  open 𝓜.FMMor-Comp F.mor G.mor hiding (module F; module G)


MPCAMor-comp : MPCAMor A C
MPCAMor-comp = record
  {mor = FMMor-comp
  ;commutes = ≈₁-begin
      (F.mor₁ ⨟₁ G.mor₁) ⨟₁ C.op
    ≈₁⟨ 𝓒₁.⨟-assoc ⟨≈₁≈⟩ 𝓒₁.⨟-cong₂ G.commutes ⟩
      F.mor₁ ⨟₁ B.op ⨟₁ 𝓕.mor G.mor₁ G.mor′
    ≈₁⟨ 𝓒₁.⨟-assocL ⟨≈₁≈⟩ 𝓒₁.⨟-cong₁ F.commutes ⟨≈₁≈⟩ 𝓒₁.⨟-assoc ⟩
      A.op ⨟₁ 𝓕.mor F.mor₁ F.mor′ ⨟₁ 𝓕.mor G.mor₁ G.mor′
    ≈₁˘⟨ 𝓒₁.⨟-cong₂ 𝓕.mor-⨟ ⟩
      A.op ⨟₁ 𝓕.mor (F.mor₁ ⨟₁ G.mor₁) (F.mor′ ⨟₂ G.mor′)
    ≈₁⟨ 𝓒₁.⨟-cong₂ (𝓕.mor-cong₂ FMMor-comp′) ⟩
      A.op ⨟₁ 𝓕.mor H.mor₁ H.mor′
    □₁
  }
open MPCAMor-Comp public using (MPCAMor-comp)


open Category₄ 𝓜.FMCategory
MPCAIdMor : {A : MonProdCoAlg} → MPCAMor A A
MPCAIdMor = record
  {mor = Id₄
  ;commutes = leftId₁ ⟨≈₁≈˘⟩ (𝓒₁.⨟-cong₂ (𝓕.mor-cong₂ 𝓜.rightUnit ⟨≈₁≈⟩ 𝓕.mor-Id) ⟨≈₁≈⟩ rightId₁)
  }
MPCACategory : Category (j₁ ⊎ j₂ ⊎ k₁ ⊎ k₂) (k₁ ⊎ k₂) MonProdCoAlg
MPCACategory = retract²Category 𝓜.FMCategory
  (λ {A} → MPCAIdMor {A})
  (λ {A} {B} {C} F G → MPCAMor-comp {A} {B} {C} F G)
  MonProdCoAlg.Carrier
  MPCAMor.mor
  ≈₄-refl
  ≈₄-refl
```

7.4 Categoric.MonCoAlg.FM.MorCompat

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
```

```
open import Categoric.Semigroupoid
open import Categoric.IdOp
open import Categoric.Category
open import Categoric.Category.FinColimits
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Functor.Product
open import Categoric.Product.Category
open import Categoric.Monad
open import Categoric.Monad.FunctorMonad
open import Categoric.Monad.Product

open CatFinLimits


module Categoric.MonCoAlg.FM.MorCompat
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (𝒞₁ : Category {i₁} j₁ k₁ Obj₁)
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (𝒞₂ : Category {i₂} j₂ k₂ Obj₂)
  (ℳ : FunctorMonad {𝒞₁ = 𝒞₁} {𝒞₂ = 𝒞₂})
  (ℱ : Bifunctor 𝒞₁ 𝒞₂ 𝒞₁)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  where

open import Categoric.MonCoAlg.FM.Obj 𝒞₁ 𝒞₂ ℳ ℱ
open import Categoric.MonCoAlg.FM.Cat 𝒞₁ 𝒞₂ ℳ ℱ
open import Categoric.MonCoAlg.FM.ObjCompat 𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂
open import Categoric.MonCoAlg.FM.DistrJoin 𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂

open Category₁ 𝒞₁
open Category₂ 𝒞₂
open HasTerminalObject₂ 𝒞₂ hasTerminal₂

private
  module 𝒞₀ = Category 𝒞₀
  module 𝒞₁ = Category 𝒞₁
  module 𝒞₂ = Category 𝒞₂
  module ℳ = FunctorMonad ℳ
  module ℱ = Bifunctor ℱ
  module ℳ₀ = Monad′ ℳ₀
  module ℱ₀ = Functor ℱ₀


morFrom : {A B : MonProdCoAlg} → MPCAMor A B → MCAMor (objFrom A) (objFrom B)
morFrom {MPCA _ Aop} {MPCA _ Bop} F = record
  {mor = F.mor
  ;commutes = (≈₁-begin
                 F.mor₁ ⨟₁ Bop ⨟₁ Id₁
               ≈₁⟨ 𝒞₁.⨟-cong₂ rightId₁ ⟩
                 F.mor₁ ⨟₁ Bop
               ≈₁⟨ F.commutes ⟨≈₁≈ˇ⟩ 𝒞₁.⨟-cong₂ (ℱ.mor-cong₁ rightId₁) ⟩
                 Aop ⨟₁ ℱ.mor (F.mor₁ ⨟₁ Id₁) F.mor′
               □₁)
             ,Ⓘ₂≈
  }
  where
    module F = MPCAMor F


morTo : {A₁ A₂ : MonCoAlg} → MCAMor A₁ A₂ → MPCAMor (objTo A₁) (objTo A₂)
morTo (MkMCAMor F (commutes, _)) = record
  {mor = F
  ;commutes = 𝒞₁.⨟-cong₂ rightId₁ ⟨≈₁ˇ≈⟩ commutes ⟨≈₁≈⟩ 𝒞₁.⨟-cong₂ (ℱ.mor-cong₁ rightId₁)
  }
```

7.5 Categoric.MonCoAlg.FM.DistrJoin

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
open import Categoric.Semigroupoid
open import Categoric.IdOp
open import Categoric.Category
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad
open import Categoric.Monad.FunctorMonad

open CatFinLimits


module Categoric.MonCoAlg.FM.DistrJoin
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (𝒞₁ : Category {i₁} j₁ k₁ Obj₁)
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (𝒞₂ : Category {i₂} j₂ k₂ Obj₂)
  (ℳ : FunctorMonad {𝒞₁ = 𝒞₁} {𝒞₂ = 𝒞₂})
  (ℱ : Bifunctor 𝒞₁ 𝒞₂ 𝒞₁)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  where

open import Categoric.MonCoAlg.FM.Obj 𝒞₁ 𝒞₂ ℳ ℱ
open import Categoric.MonCoAlg.FM.Cat 𝒞₁ 𝒞₂ ℳ ℱ
open import Categoric.MonCoAlg.FM.ObjCompat 𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂

open Category₀ 𝒞₀
open Category₁ 𝒞₁
open Category₂ 𝒞₂
open HasTerminalObject₂ 𝒞₂ hasTerminal₂

private
  module 𝒞₀ = Category 𝒞₀
  module 𝒞₁ = Category 𝒞₁
  module 𝒞₂ = Category 𝒞₂
  module ℳ = FunctorMonad ℳ
  module ℱ = Bifunctor ℱ
  module ℳ₀ = Monad′ ℳ₀
  module ℱ₀ = Functor ℱ₀


distrJoin₀ : {A : Obj₀} → 𝒞₀.Mor (ℳ₀.obj (ℱ₀.obj (ℳ₀.obj A))) (ℱ₀.obj (ℳ₀.obj A))
distrJoin₀ {A₁, A₂} = Id₁, ⓣ₂


distrJoin-naturality : {A B : Obj₀} {f : Mor₀ A B}
                     → Functor.mor (ℳ₀.M ⨟ ℱ₀ ⨟ ℳ₀.M) f ⨟₀ distrJoin₀ {B}
                     ≈₀ distrJoin₀ {A} ⨟₀ Functor.mor (ℳ₀.M ⨟ ℱ₀) f
distrJoin-naturality = (rightId₁ ⟨≈₁≈˘⟩ leftId₁), Ⓣ₂≈

ℳℱdistrJoin : NatTrans (ℳ₀.M ⨟ ℱ₀ ⨟ ℳ₀.M) (ℳ₀.M ⨟ ℱ₀)
ℳℱdistrJoin = record {indmor = distrJoin₀; naturality = distrJoin-naturality}


return-distrJoin : {A : Obj₀} → ℳ₀.return ⨟₀ distrJoin₀ {A} ≈₀ Id₀

return-distrJoin {A₁, A₂} = leftId₁, Ⓣ₂≈

join-distrJoin : {A : Obj₀} → ℳ₀.join ⨟₀ distrJoin₀ {A} ≈₀ ℳ₀.mor distrJoin₀ ⨟₀ distrJoin₀
join-distrJoin {A₁, A₂} = ≈₁-refl, Ⓣ₂≈

join↗distrJoin : {A : Obj₀} → ℳ₀.mor (ℱ₀.mor ℳ₀.join) ⨟₀ distrJoin₀ {A} ≈₀ distrJoin₀ ⨟₀ ℱ₀.mor ℳ₀.join
join↗distrJoin {A₁, A₂} = (rightId₁ ⟨≈₁≈˘⟩ leftId₁), Ⓣ₂≈
```

```
open import Categoric.MonCoAlg.Cat2 𝒞₀ ℳ₀ ℱ₀ ℳℱdistrJoin
  (λ {A} → return-distrJoin {A})
  (λ {A} → join-distrJoin {A})
  (λ {A} → join↗distrJoin {A})
  public using (MCACategory; MCAMor; MkMCAMor; module MCAMor; ↼-⨟-distrJoin)
```

7.6 Categoric.MonCoAlg.FM.SEAG

Symbolically edge-attributed graphs.

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
open import Categoric.Semigroupoid
open import Categoric.IdOp
open import Categoric.Category
open import Categoric.Category.FinColimits
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Functor.Product
open import Categoric.Product.Category
open import Categoric.Monad
open import Categoric.Monad.FunctorMonad
open CatFinLimits


module Categoric.MonCoAlg.FM.SEAG
  {i j k : Level} {Obj₂ : Set i} (𝒞₂ : Category {i} j k Obj₂) (𝒯 : Monad 𝒞₂)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  (hasProducts₂ : HasProducts 𝒞₂)
  where
open Category₂ 𝒞₂
open HasTerminalObject₂ 𝒞₂ hasTerminal₂
open HasProducts₂ 𝒞₂ hasProducts₂
module MPCS-SEAG where
  Obj₁ = Obj₂ × Obj₂
  𝒞₁ : Category j k Obj₁
  𝒞₁ = ProductCategory 𝒞₂ 𝒞₂
  ⊗₂ : Functor 𝒞₁ 𝒞₂
  ⊗₂ = biFunctor (ProductBifunctor 𝒞₂ hasProducts₂)
  module 𝒞₁ = Category 𝒞₁
  module 𝒞₂ = Category 𝒞₂
  ℳ : FunctorMonad {𝒞₁ = 𝒞₁} {𝒞₂ = 𝒞₂}
  ℳ = ConstFunctorMonad 𝒯
  module ℳ = FunctorMonad ℳ
  module 𝒯 = Monad′ 𝒯
open MPCS-SEAG
```

We have, conceptually: $\mathcal{F}\ (\mathsf{N},\mathsf{E})\ \mathcal{T}\mathsf{V} = (1, (\mathsf{N} \times \mathsf{N} \times \mathcal{T}\mathsf{V}))$

```
ℱ : Bifunctor 𝒞₁ 𝒞₂ 𝒞₁
ℱ = toBifunctor
  (ConstFunctor {Src = ProductCategory 𝒞₁ 𝒞₂} {Trg = 𝒞₂} 𝒯₂
    ▼
    ((ProdFunctor (Proj₁ 𝒞₂ 𝒞₂ ⨟⨟ (Identity 𝒞₂ ▼ Identity 𝒞₂) ⨟⨟ ⊗₂)
```

```
                    (Identity C₂)
    ) ⁘ ⊗₂))
module F = Bifunctor F
```

With this, we can instantiate the Categoric.MonCoAlg.FM setup:

```
open import Categoric.MonCoAlg.FM.ObjCompat C₁ C₂ M F hasTerminal₂
open import Categoric.MonCoAlg.FM.MorCompat C₁ C₂ M F hasTerminal₂
```

Chapter 8

Monadic Co-Algebras over a Product Category

We can specialise the monadic coalgebras of Chapter 4 in a way that generalises the setup for symbolically edge-attributed graphs in Categoric.MonCoAlg.SEAG.FM (Sect. 5.2), while still allowing the pushout construction there.

The most general shape we have been able to identify for this are "monadic product coalgebras" over a product category $\mathcal{C}_1 \times \mathcal{C}_2$, defined via a monad $\mathcal{M}$ on $\mathcal{C}_2$ and a functor $\mathcal{F}$ from $\mathcal{C}_1 \times \mathcal{C}_2$ to $\mathcal{C}_1$. In terms of coalgebraic signatures this implements the restriction that sorts mentioned as monad arguments do not occur as source sorts of operators, and that the monad must not depend on sorts that do occur as source sorts of operators.

8.1 Categoric.MonCoAlg.P.Obj

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import Categoric.Category
open import Categoric.Functor using (Bifunctor; module Bifunctor)
open import Categoric.Monad using (Monad; module Monad′)
```

Here we investigate a specialised variant of monadic coalgebras based on a ProductCategory $\mathcal{C}_1$ $\mathcal{C}_2$, where the monad is only on $\mathcal{C}_2$, and the functor goes only to $\mathcal{C}_1$, which means that there are no operations starting at $\mathcal{C}_2$ sorts.

SEAGraph is an instance of this, but TermGraph is not.

```
module Categoric.MonCoAlg.P.Obj
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (𝒞₁ : Category {i₁} j₁ k₁ Obj₁)
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (𝒞₂ : Category {i₂} j₂ k₂ Obj₂)
  (ℳ : Monad 𝒞₂)
  (ℱ : Bifunctor 𝒞₁ 𝒞₂ 𝒞₁)
  where
private
  Obj₀ = Obj₁ × Obj₂
  module 𝒞₁ = Category 𝒞₁
  module 𝒞₂ = Category 𝒞₂
  module ℳ = Monad′ ℳ
  module ℱ = Bifunctor ℱ
open Category₁ 𝒞₁
open Category₂ 𝒞₂

record MonProdCoAlg : Set (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂) where
  constructor MPCA
  field Carrier : Obj₀
  Carrier₁ = proj₁ Carrier
```

```
  Carrier₂ = proj₂ Carrier
  field op : Mor₁ Carrier₁ (ℱ.obj Carrier₁ (ℳ.obj Carrier₂))
```

8.2 Categoric.MonCoAlg.P.ObjCompat

```
open import RATH.Level
open import RATH.PropositionalEquality using (_≡_; ≡-refl)
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)

open import Categoric.Category
open import Categoric.Category.FinLimits using (module CatFinLimits)
open import Categoric.Functor
  using (Functor; module Functor; Bifunctor; module Bifunctor; ConstFunctor)
open import Categoric.Monad using (Monad; module Monad′; Id ℳ)
open import Categoric.Product.Category using (ProductCategory; _▼_; biFunctor)
open import Categoric.Monad.Product using (ProductMonad)
open CatFinLimits using (HasTerminalObject; module HasTerminalObject₂)


module Categoric.MonCoAlg.P.ObjCompat
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (𝒞₁ : Category {i₁} j₁ k₁ Obj₁)
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (𝒞₂ : Category {i₂} j₂ k₂ Obj₂)
  (ℳ : Monad 𝒞₂)
  (ℱ : Bifunctor 𝒞₁ 𝒞₂ 𝒞₁)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  where

open import Categoric.MonCoAlg.P.Obj 𝒞₁ 𝒞₂ ℳ ℱ
open import Categoric.MonCoAlg.P.Monad 𝒞₁ 𝒞₂ ℳ ℱ public

private
  module 𝒞₁ = Category 𝒞₁
  module 𝒞₂ = Category 𝒞₂
  module 𝒞₃ = Category 𝒞₃
  module ℳ = Monad′ ℳ
  module ℱ = Bifunctor ℱ

open Category₁ 𝒞₁
open Category₂ 𝒞₂
open Category₃ 𝒞₃
open HasTerminalObject₂ 𝒞₂ hasTerminal₂


ℱ₀ : Functor 𝒞₀ 𝒞₀
ℱ₀ = biFunctor ℱ ▼ ConstFunctor {Src = 𝒞₀} {Trg = 𝒞₂} 𝟙₂

private
  module 𝒞₀ = Category 𝒞₀
  module ℳ₀ = Monad′ ℳ₀
  module ℱ₀ = Functor ℱ₀

open Category₀ 𝒞₀
open import Categoric.MonCoAlg.Obj 𝒞₀ ℳ₀ ℱ₀ public
open ℳ₀ using () renaming (↼ to ↼₀; _⨟_ to _⨟₀_)


objTo : MonCoAlg → MonProdCoAlg
objTo (MCA A (opA, _)) = record
  {Carrier = A
  ;op = opA
  }
```

```
objFrom : MonProdCoAlg → MonCoAlg
objFrom (MPCA A opA) = record
  {Carrier = A
  ;op = opA, ⓣ₂
  }

objFromTo : {G : MonProdCoAlg} → (objTo (objFrom G)) ≡ G
objFromTo = ≡-refl
```

For the converse composition, we obtain only morphism equality of the second component of MonCoAlg.op, due to $\approx ⓣ_2$, but not propositional equality.

```
objToFrom : {A : MonCoAlg}
          → 𝒞₃.Iso (MonCoAlg.Carrier (objFrom (objTo A))) (MonCoAlg.Carrier A)
objToFrom {(MCA A _)} = 𝒞₃.IdIso {A}
```

8.3 Categoric.MonCoAlg.P.Monad

```
open import RATH.Level
open import RATH.Data.Product using (_×_)
open import Categoric.Category
open import Categoric.Functor using (Bifunctor)
open import Categoric.Monad using (Monad; module Monad′; Id𝓜)
open import Categoric.Product.Category using (ProductCategory)
open import Categoric.Monad.Product using (ProductMonad)

module Categoric.MonCoAlg.P.Monad
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (𝒞₁ : Category {i₁} j₁ k₁ Obj₁)
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (𝒞₂ : Category {i₂} j₂ k₂ Obj₂)
  (𝓜 : Monad 𝒞₂)
  (ℱ : Bifunctor 𝒞₁ 𝒞₂ 𝒞₁)
  where
open import Categoric.MonCoAlg.P.Obj 𝒞₁ 𝒞₂ 𝓜 ℱ

Obj₀ : Set (i₁ ⊎ i₂)
Obj₀ = Obj₁ × Obj₂
𝒞₀ : Category (j₁ ⊎ j₂) (k₁ ⊎ k₂) Obj₀
𝒞₀ = ProductCategory 𝒞₁ 𝒞₂
𝓜₀ : Monad 𝒞₀
𝓜₀ = ProductMonad Id𝓜 𝓜
𝒞₃ : Category (j₁ ⊎ j₂) (k₁ ⊎ k₂) Obj₀
𝒞₃ = Monad′.KleisliCat 𝓜₀
```

8.4 Categoric.MonCoAlg.P.Cat

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import Categoric.LESGraph using (LocalSetoid; module LocalEdgeSetoid)
open import Categoric.Semigroupoid
open import Categoric.IdOp
```

```
open import Categoric.Category
open import Categoric.Functor
open import Categoric.Monad
open import Categoric.Monad.Product


module Categoric.MonCoAlg.P.Cat
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (𝒞₁ : Category {i₁} j₁ k₁ Obj₁)
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (𝒞₂ : Category {i₂} j₂ k₂ Obj₂)
  (ℳ : Monad 𝒞₂)
  (ℱ : Bifunctor 𝒞₁ 𝒞₂ 𝒞₁)
  where
open import Categoric.MonCoAlg.P.Obj 𝒞₁ 𝒞₂ ℳ ℱ
open import Categoric.MonCoAlg.P.Monad 𝒞₁ 𝒞₂ ℳ ℱ using (𝒞₃; ℳ₀)
private
    -- module 𝒞₀ = Category 𝒞₀
  module 𝒞₁ = Category 𝒞₁
  module 𝒞₂ = Category 𝒞₂
  module ℳ = Monad′ ℳ
  module ℱ = Bifunctor ℱ
  module ℳ₀ = Monad′ ℳ₀
  module 𝒞₃ = Category 𝒞₃
  -- open Category₀ 𝒞₀
open Category₁ 𝒞₁
open Category₂ 𝒞₂
open Category₃ 𝒞₃


record MPCAMor (A B : MonProdCoAlg) : Set (j₁ ∪ j₂ ∪ k₁ ∪ k₂) where
  private module A = MonProdCoAlg A
          module B = MonProdCoAlg B
  field mor : Mor₃ A.Carrier B.Carrier
  mor₁ = proj₁ mor
  mor₂ = proj₂ mor
  field commutes : mor₁ ⨾₁ B.op ≈₁ A.op ⨾₁ ℱ.mor mor₁ (ℳ.η mor₂)


module MPCAMor-Comp {A B C : MonProdCoAlg} (F : MPCAMor A B) (G : MPCAMor B C) where
  private
    module A = MonProdCoAlg A
    module B = MonProdCoAlg B
    module C = MonProdCoAlg C
    module F = MPCAMor F
    module G = MPCAMor G
    H = F.mor ⨾₃ G.mor


MPCAMor-comp : MPCAMor A C
MPCAMor-comp = record
  {mor = H
  ;commutes = ≈₁-begin
      (F.mor₁ ⨾₁ G.mor₁ ⨾₁ Id₁ ⨾₁ Id₁) ⨾₁ C.op
    ≈₁⟨ 𝒞₁.⨾-assoc ⟨≈₁≈⟩ 𝒞₁.⨾-cong₂ (𝒞₁.⨾-cong₁ 𝒞₁.rightId² ⟨≈₁≈⟩ G.commutes) ⟩
      F.mor₁ ⨾₁ B.op ⨾₁ ℱ.mor G.mor₁ (ℳ.η G.mor₂)
    ≈₁⟨ 𝒞₁.⨾-assocL ⟨≈₁≈⟩ 𝒞₁.⨾-cong₁ F.commutes ⟨≈₁≈⟩ 𝒞₁.⨾-assoc ⟩
      A.op ⨾₁ ℱ.mor F.mor₁ (ℳ.η F.mor₂) ⨾₁ ℱ.mor G.mor₁ (ℳ.η G.mor₂)
    ≈₁˘⟨ 𝒞₁.⨾-cong₂ ℱ.mor-⨾ ⟩
      A.op ⨾₁ ℱ.mor (F.mor₁ ⨾₁ G.mor₁) (ℳ.η F.mor₂ ⨾₂ ℳ.η G.mor₂)
    ≈₁⟨ 𝒞₁.⨾-cong₂ ((ℱ.mor-cong (𝒞₁.⨾-cong₂ (≈₁-sym 𝒞₁.rightId²)) ℳ.η-⨾-η)) ⟩
```

```
      A.op ⨟₁ 𝓕.mor (F.mor₁ ⨟₁ G.mor₁ ⨟₁ Id₁ ⨟₁ Id₁) (𝓜.↼ (proj₂ H))
    □₁
  }
open MPCAMor-Comp public using (MPCAMor-comp)

MPCAIdMor : {A : MonProdCoAlg} → MPCAMor A A
MPCAIdMor = record
  {mor = Id₃
  ;commutes = leftId₁ ⟨≈₁≈˘⟩ (𝓒₁.⨟-cong₂ (𝓕.mor-cong₂ 𝓜.rightUnit ⟨≈₁≈⟩ 𝓕.mor-Id) ⟨≈₁≈⟩ rightId₁)
  }
MPCACategory : Category (j₁ ⊎ j₂ ⊎ k₁ ⊎ k₂) (k₁ ⊎ k₂) MonProdCoAlg
MPCACategory = retract²Category 𝓜₀.KleisliCat
  (λ {A} → MPCAIdMor {A})
  (λ {A} {B} {C} F G → MPCAMor-comp {A} {B} {C} F G)
  MonProdCoAlg.Carrier
  MPCAMor.mor
  ≈₃-refl
  ≈₃-refl
```

8.5 Categoric.MonCoAlg.P.DistrJoin

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import Categoric.Category
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad
open CatFinLimits using (HasTerminalObject; module HasTerminalObject₂)


module Categoric.MonCoAlg.P.DistrJoin
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (𝓒₁ : Category {i₁} j₁ k₁ Obj₁)
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (𝓒₂ : Category {i₂} j₂ k₂ Obj₂)
  (𝓜 : Monad 𝓒₂)
  (𝓕 : Bifunctor 𝓒₁ 𝓒₂ 𝓒₁)
  (hasTerminal₂ : HasTerminalObject 𝓒₂)
  where
open import Categoric.MonCoAlg.P.Obj 𝓒₁ 𝓒₂ 𝓜 𝓕
open import Categoric.MonCoAlg.P.Cat 𝓒₁ 𝓒₂ 𝓜 𝓕
open import Categoric.MonCoAlg.P.ObjCompat 𝓒₁ 𝓒₂ 𝓜 𝓕 hasTerminal₂
open Category₀ 𝓒₀
open Category₁ 𝓒₁
open Category₂ 𝓒₂
open HasTerminalObject₂ 𝓒₂ hasTerminal₂
private
  module 𝓒₀ = Category 𝓒₀
  module 𝓒₁ = Category 𝓒₁
  module 𝓒₂ = Category 𝓒₂
  module 𝓜 = Monad′ 𝓜
  module 𝓕 = Bifunctor 𝓕
  module 𝓜₀ = Monad′ 𝓜₀
  module 𝓕₀ = Functor 𝓕₀

distrJoin₀ : {A : Obj₀} → 𝓒₀.Mor (𝓜₀.obj (𝓕₀.obj (𝓜₀.obj A))) (𝓕₀.obj (𝓜₀.obj A))
distrJoin₀ {A₁, A₂} = Id₁, 𝕥₂
```

```
distrJoin-naturality : {A B : Obj₀} {f : Mor₀ A B}
                     → Functor.mor (ℳ₀.M ⨟ ℱ₀ ⨟ ℳ₀.M) f ⨟₀ distrJoin₀ {B}
                     ≈₀ distrJoin₀ {A} ⨟₀ Functor.mor (ℳ₀.M ⨟ ℱ₀) f
distrJoin-naturality = (rightId₁ ⟨≈₁≈˘⟩ leftId₁), Ⓣ₂≈

ℳℱdistrJoin : NatTrans (ℳ₀.M ⨟ ℱ₀ ⨟ ℳ₀.M) (ℳ₀.M ⨟ ℱ₀)
ℳℱdistrJoin = record {indmor = distrJoin₀; naturality = distrJoin-naturality}


return-distrJoin : {A : Obj₀} → ℳ₀.return ⨟₀ distrJoin₀ {A} ≈₀ Id₀
return-distrJoin {A₁, A₂} = leftId₁, Ⓣ₂≈

join-distrJoin : {A : Obj₀} → ℳ₀.join ⨟₀ distrJoin₀ {A} ≈₀ ℳ₀.mor distrJoin₀ ⨟₀ distrJoin₀
join-distrJoin {A₁, A₂} = rightId₁, Ⓣ₂≈

join↗distrJoin : {A : Obj₀}
               → ℳ₀.mor (ℱ₀.mor ℳ₀.join) ⨟₀ distrJoin₀ {A} ≈₀ distrJoin₀ ⨟₀ ℱ₀.mor ℳ₀.join
join↗distrJoin {A₁, A₂} = (rightId₁ ⟨≈₁≈˘⟩ leftId₁), Ⓣ₂≈


open import Categoric.MonCoAlg.Cat2 𝒞₀ ℳ₀ ℱ₀ ℳℱdistrJoin
  (λ {A} → return-distrJoin {A})
  (λ {A} → join-distrJoin {A})
  (λ {A} → join↗distrJoin {A})
  public using (MCACategory; MCAMor; MkMCAMor; module MCAMor; _↼-⨟-distrJoin)
```

8.6 Categoric.MonCoAlg.P.MorCompat

We collect potentially useful properties of $\mathcal{F}$ and $\mathcal{M}$ as defined in Categoric.MonCoAlg.SEAG.FM (Sect. 5.2).

```
open import RATH.Level
open import RATH.Data.Product using (_×_; _,_; proj₁; proj₂)
open import Categoric.Category
open import Categoric.Category.FinColimits
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Functor.Product
open import Categoric.Product.Category
open import Categoric.Monad
open import Categoric.Monad.Product
open CatFinLimits


module Categoric.MonCoAlg.P.MorCompat
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (𝒞₁ : Category {i₁} j₁ k₁ Obj₁)
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (𝒞₂ : Category {i₂} j₂ k₂ Obj₂)
  (ℳ : Monad 𝒞₂)
  (ℱ : Bifunctor 𝒞₁ 𝒞₂ 𝒞₁)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  where
open import Categoric.MonCoAlg.P.Obj 𝒞₁ 𝒞₂ ℳ ℱ
open import Categoric.MonCoAlg.P.Cat 𝒞₁ 𝒞₂ ℳ ℱ
open import Categoric.MonCoAlg.P.ObjCompat 𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂
open import Categoric.MonCoAlg.P.DistrJoin 𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂
open Category₁ 𝒞₁
open Category₂ 𝒞₂
open HasTerminalObject₂ 𝒞₂ hasTerminal₂
private
```

```
  module 𝒞₀ = Category 𝒞₀
  module 𝒞₁ = Category 𝒞₁
  module 𝒞₂ = Category 𝒞₂
  module 𝒞₃ = Category 𝒞₃
  module ℳ = Monad′ ℳ
  module ℱ = Bifunctor ℱ
  module ℳ₀ = Monad′ ℳ₀
  module ℱ₀ = Functor ℱ₀


morFrom : {A B : MonProdCoAlg} → MPCAMor A B → MCAMor (objFrom A) (objFrom B)
morFrom {MPCA _ Aop} {MPCA _ Bop} F = record
  {mor = F.mor
  ;commutes = (≈₁-begin
                 F.mor₁ ⨟₁ Bop ⨟₁ Id₁
               ≈₁⟨ 𝒞₁.⨟-cong₂ rightId₁ ⟩
                 F.mor₁ ⨟₁ Bop
               ≈₁⟨ F.commutes ⟨≈₁≈˘⟩ 𝒞₁.⨟-cong₂ (ℱ.mor-cong₁ 𝒞₁.rightId²) ⟩
                 Aop ⨟₁ ℱ.mor (F.mor₁ ⨟₁ Id₁ ⨟₁ Id₁) (ℳ.↼ F.mor₂)
               □₁)
             ,𝟙₂≈
  }
  where
    module F = MPCAMor F


morTo : {A₁ A₂ : MonCoAlg} → MCAMor A₁ A₂ → MPCAMor (objTo A₁) (objTo A₂)
morTo (MkMCAMor F (commutes, _)) = record
  {mor = F
  ;commutes = 𝒞₁.⨟-cong₂ rightId₁ ⟨≈₁˘≈⟩ commutes ⟨≈₁≈⟩ 𝒞₁.⨟-cong₂ (ℱ.mor-cong₁ 𝒞₁.rightId²)
  }


objToFrom′ : {A : MonCoAlg}
           → Category.Iso MCACategory (objFrom (objTo A)) A
objToFrom′ {A} = record
  {isoMor = MkMCAMor (𝒞₃.isoMor (objToFrom {A}))
      ((leftId₁ ⟨≈₁≈˘⟩ 𝒞₁.⨟-cong₂ (ℱ.mor-cong 𝒞₁.rightId² ℳ.rightUnit ⟨≈₁≈⟩ ℱ.mor-Id)
      ),𝟙₂≈)
  ;isIso = record
    {_⁻¹ = MkMCAMor (𝒞₃._⁻¹ (objToFrom {A}))
      ((leftId₁ ⟨≈₁≈˘⟩ 𝒞₁.⨟-cong₂ (ℱ.mor-cong 𝒞₁.rightId² ℳ.rightUnit ⟨≈₁≈⟩ ℱ.mor-Id)
      ),𝟙₂≈)
    ;rightInverse = 𝒞₃.rightInverse (objToFrom {A})
    ;leftInverse = 𝒞₃.leftInverse (objToFrom {A})
    }
  }
  where
    module A = MonCoAlg A
```

8.7 Categoric.MonCoAlg.P.To

We produce the functor to the MPCACategory from the appropriately instantiated MCACategory.

```
open import RATH.Level
open import RATH.Data.Product using (_,_)
open import Categoric.Category
```

```
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad

open CatFinLimits


module Categoric.MonCoAlg.P.To
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (𝒞₁ : Category {i₁} j₁ k₁ Obj₁)
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (𝒞₂ : Category {i₂} j₂ k₂ Obj₂)
  (ℳ : Monad 𝒞₂)
  (ℱ : Bifunctor 𝒞₁ 𝒞₂ 𝒞₁)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  where

open import Categoric.MonCoAlg.P.Obj          𝒞₁ 𝒞₂ ℳ ℱ
open import Categoric.MonCoAlg.P.Cat          𝒞₁ 𝒞₂ ℳ ℱ using (MPCACategory)
open import Categoric.MonCoAlg.P.DistrJoin    𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂ using (MCACategory)
open import Categoric.MonCoAlg.P.ObjCompat    𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂ using (𝒞₀; ℳ₀; ℱ₀; objTo)
open import Categoric.MonCoAlg.P.MorCompat    𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂ using (morTo)

open Category₀ 𝒞₀
open Category₁ 𝒞₁
open Category₂ 𝒞₂
open HasTerminalObject₂ 𝒞₂ hasTerminal₂
private
  module 𝒞₀ = Category 𝒞₀
  module 𝒞₁ = Category 𝒞₁
  module 𝒞₂ = Category 𝒞₂
  module ℳ = Monad′ ℳ
  module ℱ = Bifunctor ℱ
  module ℳ₀ = Monad′ ℳ₀
  module ℱ₀ = Functor ℱ₀


To : Functor MCACategory MPCACategory
To = record
  {obj = objTo
  ; mor = morTo
  ; mor-cong = λ F≈G → F≈G
  ; mor-⨟ = ≈₀-refl
  ; mor-Id = ≈₀-refl
  }
```

8.8 Categoric.MonCoAlg.P.From

We produce the functor from the MPCACategory to the appropriately instantiated MCACategory.

```
open import RATH.Level
open import RATH.Data.Product using (_,_)

open import Categoric.Category
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad

open CatFinLimits


module Categoric.MonCoAlg.P.From
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (𝒞₁ : Category {i₁} j₁ k₁ Obj₁)
```

```
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (𝒞₂ : Category {i₂} j₂ k₂ Obj₂)
  (ℳ : Monad 𝒞₂)
  (ℱ : Bifunctor 𝒞₁ 𝒞₂ 𝒞₁)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  where
open import Categoric.MonCoAlg.P.Obj          𝒞₁ 𝒞₂ ℳ ℱ
open import Categoric.MonCoAlg.P.Cat          𝒞₁ 𝒞₂ ℳ ℱ using (MPCACategory)
open import Categoric.MonCoAlg.P.DistrJoin    𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂ using (MCACategory)
open import Categoric.MonCoAlg.P.ObjCompat    𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂ using (𝒞₀; ℳ₀; ℱ₀; objFrom)
open import Categoric.MonCoAlg.P.MorCompat    𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂ using (morFrom)
open Category₀ 𝒞₀
open Category₁ 𝒞₁
open Category₂ 𝒞₂
open HasTerminalObject₂ 𝒞₂ hasTerminal₂
private
  module 𝒞₀ = Category 𝒞₀
  module 𝒞₁ = Category 𝒞₁
  module 𝒞₂ = Category 𝒞₂
  module ℳ = Monad′ ℳ
  module ℱ = Bifunctor ℱ
  module ℳ₀ = Monad′ ℳ₀
  module ℱ₀ = Functor ℱ₀


From : Functor MPCACategory MCACategory
From = record
  {obj = objFrom
  ;mor = morFrom
  ;mor-cong = λ F≈G → F≈G
  ;mor-⨾ = ≈₀-refl
  ;mor-Id = ≈₀-refl
  }
```

8.9 Categoric.MonCoAlg.P.ToFrom

We show one side of the equivalence between the SEAGraph category and the corresponding MonCoAlg category.

```
open import RATH.Level
open import RATH.Data.Product using (_,_)
open import Categoric.Category
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad
open CatFinLimits


module Categoric.MonCoAlg.P.ToFrom
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (𝒞₁ : Category {i₁} j₁ k₁ Obj₁)
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (𝒞₂ : Category {i₂} j₂ k₂ Obj₂)
  (ℳ : Monad 𝒞₂)
  (ℱ : Bifunctor 𝒞₁ 𝒞₂ 𝒞₁)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  where
open import Categoric.MonCoAlg.P.DistrJoin    𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂ using (MCACategory)
open import Categoric.MonCoAlg.P.MorCompat    𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂ using (objToFrom′)
open import Categoric.MonCoAlg.P.From         𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂ using (From)
```

```
open import Categoric.MonCoAlg.P.To          𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂ using (To)
open Category₁ 𝒞₁
open Category₂ 𝒞₂
private
  module 𝒞₁ = Category 𝒞₁
  module 𝒞₂ = Category 𝒞₂
  module ℳ = Monad′ ℳ


ToFrom : NatIso (To ⁘ From) (Identity MCACategory)
ToFrom = IsoNat objToFrom′
  ((𝒞₁.rightId³ ⟨≈₁≈˘⟩ (𝒞₁.leftId ⟨≈₁≈⟩ 𝒞₁.rightId²))
  , (𝒞₂.⨟-cong₂ ℳ.rightUnit ⟨≈₂≈˘⟩ (𝒞₂.⨟-cong₁&₂₁ ℳ.return-naturality ⟨≈₂˘≈⟩ 𝒞₂.⨟-cong₂ ℳ.leftUnit))
  )
```

8.10 Categoric.MonCoAlg.P.FromTo

We show one side of the equivalence between MPCACategory and the corresponding MonCoAlg category.

```
open import RATH.Level
open import Categoric.Category
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad
open CatFinLimits


module Categoric.MonCoAlg.P.FromTo
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (𝒞₁ : Category {i₁} j₁ k₁ Obj₁)
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (𝒞₂ : Category {i₂} j₂ k₂ Obj₂)
  (ℳ : Monad 𝒞₂)
  (ℱ : Bifunctor 𝒞₁ 𝒞₂ 𝒞₁)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  where
open import Categoric.MonCoAlg.P.Cat    𝒞₁ 𝒞₂ ℳ ℱ                 using (MPCACategory)
open import Categoric.MonCoAlg.P.From 𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂ using (From)
open import Categoric.MonCoAlg.P.To   𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂ using (To)
private
  module 𝒞₄ = Category MPCACategory
open Category₄ MPCACategory
FromToFunctor = From ⁘ To
module FromToFunctor = Functor FromToFunctor


FromTo : NatIso (From ⁘ To) (Identity MPCACategory)
FromTo = IsoNat 𝒞₄.IdIso
  (λ {G₁} {G₂} {F} → ≈₄-begin
    FromToFunctor.mor F ⨟₄ Id₄ {G₂}
    ≈₄⟨ rightId₄ {G₁} {G₂} {F} ⟩
      FromToFunctor.mor F
    ≈₄˘⟨ leftId₄ {G₁} {G₂} {F} ⟩
      Id₄ {G₁} ⨟₄ FromToFunctor.mor F
    □₄)
```

8.11 Categoric.MonCoAlg.P.Equiv

We collect all of the equivalence between MPCACategory and the corresponding MonCoAlg category.

```
open import RATH.Level

open import Categoric.Category
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Functor.Equivalence
open import Categoric.Monad

open CatFinLimits


module Categoric.MonCoAlg.P.Equiv
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (𝒞₁ : Category {i₁} j₁ k₁ Obj₁)
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (𝒞₂ : Category {i₂} j₂ k₂ Obj₂)
  (ℳ : Monad 𝒞₂)
  (ℱ : Bifunctor 𝒞₁ 𝒞₂ 𝒞₁)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  where
open import Categoric.MonCoAlg.P.Cat       𝒞₁ 𝒞₂ ℳ ℱ                 using (MPCACategory)
open import Categoric.MonCoAlg.P.DistrJoin 𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂ using (MCACategory)
open import Categoric.MonCoAlg.P.To        𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂ using (To)
open import Categoric.MonCoAlg.P.From      𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂ using (From)
open import Categoric.MonCoAlg.P.ToFrom    𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂ using (ToFrom)
open import Categoric.MonCoAlg.P.FromTo    𝒞₁ 𝒞₂ ℳ ℱ hasTerminal₂ using (FromTo)


MonCoAlg-SEAGraph-Equivalence : CatEquivalence MCACategory MPCACategory
MonCoAlg-SEAGraph-Equivalence = record
  {To     = To
  ;From   = From
  ;ToFrom = ToFrom
  ;FromTo = FromTo
  }
```

8.12 Categoric.MonCoAlg2.P.Pushout2

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

open import Categoric.Category
open import Categoric.Category.FinColimits
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Monad

open CatFinColimits
open CatFinLimits
```

Since it appears impossible to work on the corresponding development formulated directly in terms of the MCACategory of Categoric.MonCoAlg.Cat2 (Sect. 4.2) as instantiated in Categoric.MonCoAlg.P.DistrJoin (Sect. 8.5) with only 13GB of heap, we work in MPCACategory instead.

```
module Categoric.MonCoAlg.P.Pushout2
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (𝒞₁ : Category {i₁} j₁ k₁ Obj₁)
```

```
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (𝒞₂ : Category {i₂} j₂ k₂ Obj₂)
  (ℳ₂ : Monad 𝒞₂)
  (ℱ₁ : Bifunctor 𝒞₁ 𝒞₂ 𝒞₁)
  (hasTerminal₂ : HasTerminalObject 𝒞₂)
  where
open import Categoric.MonCoAlg.P.Obj 𝒞₁ 𝒞₂ ℳ₂ ℱ₁ using (MonProdCoAlg; module MonProdCoAlg)
open import Categoric.MonCoAlg.P.Cat 𝒞₁ 𝒞₂ ℳ₂ ℱ₁
open import Categoric.MonCoAlg.P.ObjCompat 𝒞₁ 𝒞₂ ℳ₂ ℱ₁ hasTerminal₂
open Category₁ 𝒞₁
open Category₂ 𝒞₂
open HasTerminalObject₂ 𝒞₂ hasTerminal₂
private
  module 𝒞₀ = Category 𝒞₀
  module 𝒞₁ = Category 𝒞₁
  module 𝒞₂ = Category 𝒞₂
  module ℳ₂ = Monad′ ℳ₂
  module ℱ₁ = Bifunctor ℱ₁
  module ℳ = Monad′ ℳ₀
  module ℱ = Functor ℱ₀
open Category 𝒞₀
open ℳ using (↼; _⨟∘_)


module MCAMor-Pushout
  {A B C : MonProdCoAlg} {D₀ : Obj₀} (F : MPCAMor A B) (G : MPCAMor A C)
  (let module A = MonProdCoAlg A)
  (let module B = MonProdCoAlg B)
  (let module C = MonProdCoAlg C)
  (let module F = MPCAMor F)
  (let module G = MPCAMor G)
  (H₀ : ℳ.KMor B.Carrier D₀) (K₀ : ℳ.KMor C.Carrier D₀)
  (isPushout : IsPushout ℳ.KleisliCat F.mor G.mor H₀ K₀)
  where
  D₁ = proj₁ D₀
  D₂ = proj₂ D₀
  H₁ = proj₁ H₀
  H₂ = proj₂ H₀
  K₁ = proj₁ K₀
  K₂ = proj₂ K₀
  D′ : Obj₀
  D′ = ℱ₁.obj D₁ (ℳ₂.obj D₂), D₂   -- originally ℱ.obj (ℳ.obj D₀)
  H′ : ℳ.KMor B.Carrier D′
  H′ = B.op ⨟₁ ℱ₁.mor H₁ (ℳ₂.↼ H₂)
     , H₂   -- originally B.op ⨟ ℱ.mor (ℳ.mor H₀) ⨟ ℳ.return
  K′ : ℳ.KMor C.Carrier D′
  K′ = C.op ⨟₁ ℱ₁.mor K₁ (ℳ₂.↼ K₂)
     , K₂          -- originally C.op ⨟ ℱ.mor (ℳ.mor K₀) ⨟ ℳ.return
  module isPO = IsPushout ℳ.KleisliCat isPushout
  isPO-commutes₁ : F.mor₁ ⨟₁ H₁ ≈₁ G.mor₁ ⨟₁ K₁
  isPO-commutes₁ = 𝒞₁.⨟-cong₂ 𝒞₁.rightId² ⟨≈₁˘≈⟩ proj₁ isPO.commutes ⟨≈₁≈⟩ 𝒞₁.⨟-cong₂ 𝒞₁.rightId²
    -- F.commutes : F.mor₁ ⨟₁ B.op ≈₁ A.op ⨟₁ ℱ₁.mor F.mor₁ (ℳ₂.↼ F.mor₂)
  opPOuniv : CoCone2Univ ℳ.KleisliCat H₀ K₀ H′ K′
  opPOuniv = isPO.universal {D′} {H′} {K′}
    ((≈₁-begin
          F.mor₁ ⨟₁ (B.op ⨟₁ ℱ₁.mor H₁ (ℳ₂.↼ H₂)) ⨟₁ Id₁ ⨟₁ Id₁
      ≈₁⟨ 𝒞₁.⨟-cong₂ 𝒞₁.rightId² ⟨≈₁≈⟩ 𝒞₁.⨟-cong₁&₂₁ F.commutes ⟩
          A.op ⨟₁ ℱ₁.mor F.mor₁ (ℳ₂.↼ F.mor₂) ⨟₁ ℱ₁.mor H₁ (ℳ₂.↼ H₂)
      ≈₁⟨ 𝒞₁.⨟-cong₂ (ℱ₁.mor-⨟ ⟨≈₁˘≈⟩ ℱ₁.mor-cong₂ ℳ₂.↼-⨟-↼) ⟩
          A.op ⨟₁ ℱ₁.mor (F.mor₁ ⨟₁ H₁) (ℳ₂.↼ (F.mor₂ ⨟₂ ℳ₂.↼ H₂))
```

```
      ≈₁⟨ C₁.⨟-cong₂ (F₁.mor-cong isPO-commutes₁ (M₂.↢-cong (proj₂ isPO.commutes))) ⟩
            A.op ⨟₁ F₁.mor (G.mor₁ ⨟₁ K₁) (M₂.↢ (G.mor₂ ⨟₂ M₂.↢ K₂))
      ≈₁˘⟨ C₁.⨟-cong₂ (F₁.mor-⨟ ⟨≈₁˘≈⟩ F₁.mor-cong₂ M₂.↢-⨟-↢) ⟩
            A.op ⨟₁ F₁.mor G.mor₁ (M₂.↢ G.mor₂) ⨟₁ F₁.mor K₁ (M₂.↢ K₂)
      ≈₁˘⟨ C₁.⨟-cong₂ C₁.rightId² ⟨≈₁≈⟩ C₁.⨟-cong₁&₂₁ G.commutes ⟩
            G.mor₁ ⨟₁ (C.op ⨟₁ F₁.mor K₁ (M₂.↢ K₂)) ⨟₁ Id₁ ⨟₁ Id₁
      □₁)
    , ((≈₂-begin
            F.mor₂ ⨟₂ M₂.↢ H₂
      ≈₂⟨ proj₂ isPO.commutes ⟩
            G.mor₂ ⨟₂ M₂.↢ K₂
      □₂))
    )
  module opPOuniv = CoCone2Univ M.KleisliCat opPOuniv
    -- opPOuniv.univMor-factors-left:
    -- H₁ ⨟₁ proj₁ opPOuniv.univMor ⨟₁ Id₁ ⨟₁ Id₁) ≈₁ B.op₁ ⨟₁ F₁.mor H₀.mor₁ (M₂.↢ H₂)
    -- H₂ ⨟₂ M₂.↢ (proj₂ opPOuniv.univMor) ≈₂ H₂
  D : MonProdCoAlg
  D = record
      {Carrier = D₀
      ;op      = proj₁ opPOuniv.univMor
      }
  module D = MonProdCoAlg D
  H : MPCAMor B D
  H = record
      {mor = H₀
      ;commutes = ≈₁-begin
                  H₁ ⨟₁ proj₁ opPOuniv.univMor
        ≈₁⟨ C₁.⨟-cong₂ C₁.rightId² ⟨≈₁˘≈⟩ proj₁ opPOuniv.univMor-factors-left ⟩
                  B.op ⨟₁ F₁.mor H₁ (M₂.↢ H₂)
        □₁
      }
  K : MPCAMor C D
  K = record
      {mor = K₀
      ;commutes = ≈₁-begin
                  K₁ ⨟₁ proj₁ opPOuniv.univMor
        ≈₁⟨ C₁.⨟-cong₂ C₁.rightId² ⟨≈₁˘≈⟩ proj₁ opPOuniv.univMor-factors-right ⟩
                  C.op ⨟₁ F₁.mor K₁ (M₂.↢ K₂)
        □₁
      }
  module H = MPCAMor H
  module K = MPCAMor K
  PO : Pushout MPCACategory F G
  PO = record
    {obj = D
    ;left = H
    ;right = K
    ;prf = record
      {commutes = isPO.commutes
      ;universal = λ {Z} {H′} {K′} F⨟∘H′≈G⨟∘K′ → let
          module Z = MonProdCoAlg Z
          module H′ = MPCAMor H′
          module K′ = MPCAMor K′
          uPOuniv : CoCone2Univ M.KleisliCat H₀ K₀ H′.mor K′.mor
          uPOuniv = isPO.universal {Z.Carrier} {H′.mor} {K′.mor} F⨟∘H′≈G⨟∘K′
          module uPOuniv = CoCone2Univ M.KleisliCat uPOuniv
            -- uPOuniv.univMor-factors-left has the following component types:
```

```
      -- H₁ ⨟₂ univMor-N ⨟₂ Id₂ ⨟₂ Id₂ ≈₂ H′.mor₁
      -- H₂ ⨟₂ 𝓜.↼ univMor-V ≈₂ H′.mor₂
  Z′ : Obj₀
  Z′ = 𝓜.obj (ℱ.obj (𝓜.obj Z.Carrier))
  -- [ WK: ] We cannot add a distrJoin at the end since we need to be in Kleisli! [ ]
  ZZ₂ = 𝓜₂.mor (ⓣ₂ {Z.Carrier₂})
  Zop′ = Z.op, ZZ₂
  H′Z : Mor B.Carrier Z′
  H′Z = H′.mor ⨟ Zop′
  K′Z : Mor C.Carrier Z′
  K′Z = K′.mor ⨟ Zop′
  uOPcommutes : F.mor ⨟∘ H′Z ≈ G.mor ⨟∘ K′Z
  uOPcommutes = ≈-begin
      F.mor ⨟∘ (H′.mor ⨟ Zop′)
    ≈⟨ 𝓜.⨟∘-⨟-assocL ⟩
      (F.mor ⨟∘ H′.mor) ⨟ Zop′
    ≈⟨ ⨟-cong₁ F⨟∘H′≈G⨟∘K′ ⟩
      (G.mor ⨟∘ K′.mor) ⨟ Zop′
    ≈⟨ 𝓜.⨟∘-⨟-assoc ⟩
      G.mor ⨟∘ (K′.mor ⨟ Zop′)
    □
  uOPuniv : CoCone2Univ 𝓜.KleisliCat H₀ K₀ H′Z K′Z
  uOPuniv = isPO.universal uOPcommutes
  module uOPuniv = CoCone2Univ 𝓜.KleisliCat uOPuniv
  U : MPCAMor D Z
  U = record
    {mor = uPOuniv.univMor
    ;commutes = let
          univMor-N : _
          univMor-N = proj₁ uPOuniv.univMor
          univMor-V : _
          univMor-V = proj₂ uPOuniv.univMor
    in
    ≈₁-begin
        proj₁ uPOuniv.univMor ⨟₁ Z.op
      ≈₁⟨ proj₁ (uOPuniv.univMor-unique′
        {uPOuniv.univMor ⨟ Zop′}
        {D.op ⨟₁ proj₁ (ℱ.mor (↼ uPOuniv.univMor) ⨟ 𝓜.return), (univMor-V ⨟₂ ZZ₂)}
        ((≈₁-begin
            H₁ ⨟₁ (univMor-N ⨟₁ Z.op) ⨟₁ Id₁ ⨟₁ Id₁
          ≈₁⟨ 𝒞₁.⨟-cong₂ 𝒞₁.rightId² ⟨≈₁≈⟩ 𝒞₁.⨟-assocL ⟩
            (H₁ ⨟₁ univMor-N) ⨟₁ Z.op
          ≈₁⟨ 𝒞₁.⨟-cong₁ (𝒞₁.⨟-cong₂ 𝒞₁.rightId² ⟨≈₁˘≈⟩ proj₁ uPOuniv.univMor-factors-left) ⟩
            H′.mor₁ ⨟₁ Z.op
          □₁)
        ,(≈₂-begin
            H₂ ⨟₂ 𝓜₂.↼ (univMor-V ⨟₂ 𝓜₂.mor ⓣ₂)
          ≈₂⟨ 𝒞₂.⨟-cong₂ 𝓜₂.↼-⨟M-mor ⟩
            H₂ ⨟₂ 𝓜₂.↼ univMor-V ⨟₂ 𝓜₂.mor ⓣ₂
          ≈₂⟨ 𝒞₂.⨟-assocL ⟨≈₂≈⟩ 𝒞₂.⨟-cong₁ (proj₂ uPOuniv.univMor-factors-left) ⟩
            H′.mor₂ ⨟₂ 𝓜₂.mor ⓣ₂
          □₂))
        ((≈₁-begin
            K₁ ⨟₁ (univMor-N ⨟₁ Z.op) ⨟₁ Id₁ ⨟₁ Id₁
          ≈₁⟨ 𝒞₁.⨟-cong₂ 𝒞₁.rightId² ⟨≈₁≈⟩ 𝒞₁.⨟-assocL ⟩
            (K₁ ⨟₁ univMor-N) ⨟₁ Z.op
          ≈₁⟨ 𝒞₁.⨟-cong₁ (𝒞₁.⨟-cong₂ 𝒞₁.rightId² ⟨≈₁˘≈⟩ proj₁ uPOuniv.univMor-factors-right) ⟩
            K′.mor₁ ⨟₁ Z.op
```

```
            □₁)
          ,(≈₂-begin
              K₂ ⨟₂ M₂.↼ (univMor-V ⨟₂ M₂.mor Ⓣ₂)
            ≈₂⟨ C₂.⨟-cong₂ M₂.↼-⨟M-mor ⟩
              K₂ ⨟₂ M₂.↼ univMor-V ⨟₂ M₂.mor Ⓣ₂
            ≈₂⟨ C₂.⨟-assocL ⟨≈₂≈⟩ C₂.⨟-cong₁ (proj₂ uPOuniv.univMor-factors-right) ⟩
              K′.mor₂ ⨟₂ M₂.mor Ⓣ₂
            □₂))
        ((≈₁-begin
              H₁ ⨟₁ (D.op ⨟₁ F₁.mor (univMor-N ⨟₁ Id₁ ⨟₁ Id₁) (M₂.↼ univMor-V) ⨟₁ Id₁)
                 ⨟₁ Id₁ ⨟₁ Id₁
            ≈₁⟨ C₁.⨟-cong₂ C₁.rightId² ⟨≈₁≈⟩ C₁.⨟-cong₂₂ rightId₁
                                       ⟨≈₁≈⟩ C₁.⨟-cong₁&₂₁ H.commutes ⟩
              B.op ⨟₁ F₁.mor H₁ (M₂.↼ H₂)
                   ⨟₁ F₁.mor (univMor-N ⨟₁ Id₁ ⨟₁ Id₁) (M₂.↼ univMor-V)
            ≈₁⟨ C₁.⨟-cong₂ (F₁.mor-⨟ ⟨≈₁˘≈⟩ F₁.mor-cong₂ M₂.↼-⨟-↼) ⟩
              B.op ⨟₁ F₁.mor (H₁ ⨟₁ univMor-N ⨟₁ Id₁ ⨟₁ Id₁) (M₂.↼ (H₂ ⨟₂ M₂.↼ univMor-V))
            ≈₁⟨ C₁.⨟-cong₂ (F₁.mor-cong (proj₁ uPOuniv.univMor-factors-left)
                           (M₂.↼-cong (proj₂ uPOuniv.univMor-factors-left))) ⟩
              B.op ⨟₁ F₁.mor H′.mor₁ (M₂.↼ H′.mor₂)
            ≈₁˘⟨ H′.commutes ⟩
              H′.mor₁ ⨟₁ Z.op
            □₁)
          ,(≈₂-begin
              H₂ ⨟₂ M₂.↼ (univMor-V ⨟₂ M₂.mor Ⓣ₂)
            ≈₂⟨ C₂.⨟-cong₂ M₂.↼-⨟M-mor ⟩
              H₂ ⨟₂ M₂.↼ univMor-V ⨟₂ M₂.mor Ⓣ₂
            ≈₂⟨ C₂.⨟-assocL ⟨≈₂≈⟩ C₂.⨟-cong₁ (proj₂ uPOuniv.univMor-factors-left) ⟩
              H′.mor₂ ⨟₂ M₂.mor Ⓣ₂
            □₂))
        ((≈₁-begin
              K₁ ⨟₁ (D.op ⨟₁ F₁.mor (univMor-N ⨟₁ Id₁ ⨟₁ Id₁) (M₂.↼ univMor-V) ⨟₁ Id₁)
                 ⨟₁ Id₁ ⨟₁ Id₁
            ≈₁⟨ C₁.⨟-cong₂ C₁.rightId² ⟨≈₁≈⟩ C₁.⨟-cong₂₂ rightId₁
                                       ⟨≈₁≈⟩ C₁.⨟-cong₁&₂₁ K.commutes ⟩
              C.op ⨟₁ F₁.mor K₁ (M₂.↼ K₂)
                   ⨟₁ F₁.mor (univMor-N ⨟₁ Id₁ ⨟₁ Id₁) (M₂.↼ univMor-V)
            ≈₁⟨ C₁.⨟-cong₂ (F₁.mor-⨟ ⟨≈₁˘≈⟩ F₁.mor-cong₂ M₂.↼-⨟-↼) ⟩
              C.op ⨟₁ F₁.mor (K₁ ⨟₁ univMor-N ⨟₁ Id₁ ⨟₁ Id₁) (M₂.↼ (K₂ ⨟₂ M₂.↼ univMor-V))
            ≈₁⟨ C₁.⨟-cong₂ (F₁.mor-cong (proj₁ uPOuniv.univMor-factors-right)
                           (M₂.↼-cong (proj₂ uPOuniv.univMor-factors-right))) ⟩
              C.op ⨟₁ F₁.mor K′.mor₁ (M₂.↼ K′.mor₂)
            ≈₁˘⟨ K′.commutes ⟩
              K′.mor₁ ⨟₁ Z.op
            □₁)
          ,(≈₂-begin
              K₂ ⨟₂ M₂.↼ (univMor-V ⨟₂ M₂.mor Ⓣ₂)
            ≈₂⟨ C₂.⨟-cong₂ M₂.↼-⨟M-mor ⟩
              K₂ ⨟₂ M₂.↼ univMor-V ⨟₂ M₂.mor Ⓣ₂
            ≈₂⟨ C₂.⨟-assocL ⟨≈₂≈⟩ C₂.⨟-cong₁ (proj₂ uPOuniv.univMor-factors-right) ⟩
              K′.mor₂ ⨟₂ M₂.mor Ⓣ₂
            □₂))
      ) ⟩
        D.op ⨟₁ F₁.mor (univMor-N ⨟₁ Id₁ ⨟₁ Id₁) (M₂.↼ univMor-V) ⨟₁ Id₁
      ≈₁⟨ C₁.⨟-cong₂ (rightId₁ ⟨≈₁≈⟩ F₁.mor-cong₁ C₁.rightId²) ⟩
        D.op ⨟₁ F₁.mor univMor-N (M₂.↼ univMor-V)
      □₁
    }
in record
```

```
        {univMor = U
        ;univMor-factors-left = uPOuniv.univMor-factors-left
        ;univMor-factors-right = uPOuniv.univMor-factors-right
        ;univMor-unique = uPOuniv.univMor-unique
        }
    }
  }
```

open MCAMor-Pushout **public using** () **renaming** (PO to MonProdCoAlg-extendPushout)

8.13 Categoric.MonCoAlg.P.SEAG

Symbolically edge-attributed graphs.

open import RATH.Level
open import RATH.Data.Product **using** ($\_\times\_$; $\_,\_$; proj_1; proj_2)
open import Categoric.LESGraph **using** (LocalSetoid; **module** LocalEdgeSetoid)
open import Categoric.Semigroupoid
open import Categoric.IdOp
open import Categoric.Category
open import Categoric.Category.FinColimits
open import Categoric.Category.FinLimits
open import Categoric.Functor
open import Categoric.Functor.Product
open import Categoric.Product.Category
open import Categoric.Monad
open import Categoric.Monad.FunctorMonad
open CatFinLimits

module Categoric.MonCoAlg.P.SEAG
  {i j k : Level} {Obj_2 : Set i} ($\mathcal{C}_2$: Category {i} j k Obj_2) ($\mathcal{T}$: Monad $\mathcal{C}_2$)
  ($\mathsf{hasTerminal}_2$: HasTerminalObject $\mathcal{C}_2$)
  ($\mathsf{hasProducts}_2$: HasProducts $\mathcal{C}_2$)
  where
open $\mathsf{Category}_2$ $\mathcal{C}_2$
open $\mathsf{HasTerminalObject}_2$ $\mathcal{C}_2$ $\mathsf{hasTerminal}_2$
open $\mathsf{HasProducts}_2$ $\mathcal{C}_2$ $\mathsf{hasProducts}_2$
module MPCS-SEAG **where**
  Obj_1 = $\mathsf{Obj}_2 \times \mathsf{Obj}_2$
  $\mathcal{C}_1$: Category j k Obj_1
  $\mathcal{C}_1$ = ProductCategory $\mathcal{C}_2$ $\mathcal{C}_2$
  $\otimes_2$: Functor $\mathcal{C}_1$ $\mathcal{C}_2$
  $\otimes_2$ = biFunctor (ProductBifunctor $\mathcal{C}_2$ $\mathsf{hasProducts}_2$)
  module $\mathcal{C}_1$ = Category $\mathcal{C}_1$
  module $\mathcal{C}_2$ = Category $\mathcal{C}_2$
  module $\mathcal{T}$ = Monad′ $\mathcal{T}$
open MPCS-SEAG

The monad $\mathcal{T}$ is exactly the monad we need for the Categoric.MonCoAlg.P setup; the functor is easily constructed: We have, conceptually: $\mathcal{F}\ (\mathsf{N},\mathsf{E})\ \mathcal{T}\mathsf{V}) = (\mathbb{1}, (\mathsf{N} \times \mathsf{N} \times \mathcal{T}\mathsf{V}))$

$\mathcal{F}$: Bifunctor $\mathcal{C}_1$ $\mathcal{C}_2$ $\mathcal{C}_1$
$\mathcal{F}$ = toBifunctor

```
  (ConstFunctor {Src = ProductCategory 𝒞₁ 𝒞₂} {Trg = 𝒞₂} ⓣ₂
    ▼
    ((ProdFunctor (Proj₁ 𝒞₂ 𝒞₂ ⁇ (Identity 𝒞₂ ▼ Identity 𝒞₂) ⁇ ⊗₂)
                  (Identity 𝒞₂)
      ) ⁇ ⊗₂))
module ℱ = Bifunctor ℱ
```

With this, we can instantiate the Categoric.MonCoAlg.P setup, and in particular obtain that pushouts in the Kleisli category for ProductMonad Id$\mathcal{M}$ $\mathcal{T}$ can be extended to pushouts in the SEAGraph category.

```
open import Categoric.MonCoAlg.P.ObjCompat  𝒞₁ 𝒞₂ 𝒯 ℱ hasTerminal₂ public
open import Categoric.MonCoAlg.P.MorCompat  𝒞₁ 𝒞₂ 𝒯 ℱ hasTerminal₂ public
open import Categoric.MonCoAlg.P.Cat        𝒞₁ 𝒞₂ 𝒯 ℱ              public
open import Categoric.MonCoAlg.P.Pushout2   𝒞₁ 𝒞₂ 𝒯 ℱ hasTerminal₂ public
```

Bibliography

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://wiki.portal.chalmers.se/agda/pmwiki.php?n=Libraries.StandardLibrary`.

Wolfram Kahl. Relational semigroupoids: Abstract relation-algebraic interfaces for finite relations between infinite types. *J. Logic and Algebraic Programming*, 76(1):60–89, 2008. doi: 10.1016/jlap.2007.10.008.

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 available for download, with 456 pages literate document output. `http://RelMiCS.McMaster.ca/RATH-Agda/`., February 2014.

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.