Relation-Algebraic Theories in Agda

RATH-Agda-2.0.0

Wolfram Kahl

kahl@cas.mcmaster.ca

with contributions by

Yuhang Zhao

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

5 January 2014

Abstract

This report dcuments the current state of the of basic category and allegory theory library of the RATH-Agda project, containing the typeset versions of (only sporadically truly) literate theories ranging from semigroupoids, which are "categories without identities", to "action lattice categories", which are division allegories that are at the same time Kleene categories (i.e., typed Kleene algebras), including also monoidal categories.

These theories are intended as interfaces for high-level programming; this current collection includes implementations in particular using concrete relations, and a number of constructions, including quotients by (abstractions of) partial equivalence relations.

The features of Agda permit flexible organisation of a fine-grained theory hierarchy, and strongly-typed programming with these nested algebras and with relational homomorphisms between them in a natural mathematical style and with remarkable ease.

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

Contents

Chapter 1

Introduction

Relation-algebraic theories range from full heterogeneous relation algebra with assumed definedness of datatype constructions like direct product and direct sum to subtheories like Kleene algebras (and Kleene algebras with tests) used in control-flow analysis, and allegories that are closer to data-flow considerations.

Nevertheless, while relation-algebraic program derivation, for example as in the "Algebra of Programming" of Bird and de Moor (1997), is enjoying quite some interest, the use of relation-algebraic abstractions for programming is rather limited, at least outside the realm of relational databases.

The present report documents the current state of a longer-term project to establish interfaces and infrastructure that ease programming with relation-algebraic abstractions. One danger of forcing people to use such high-level abstractions for programming is that they feel that they do not have a good model of the resulting performance characteristics of their programs, and in particular insufficient means to achieve high(er) efficiency. Therefore, we believe that, for such programming to be appealing and successful, reasoning capabilities are an essential component of the related infrastructure.

For this reason, we chose the dependently-typed programming language (and proof checker) Agda2 Norell (2007), which allows us to integrate programs and their correctness proof in a single source language.

The bulk of the present report consists of fine-grained, universe-polymorphic formalisations of various categories, allegories, and related structures. We show that the standard dependently-typed concept of concrete relations provides models of our theories, and proceed to construct more complex models from any given base model from algebras that have their signatures interpreted over that base.

In future work, we will extend these theories in particular in the direction of algebraic graph transformation, covering both the theory and executable implementations.

The Agda theories contained in this report are available on-line at the following URL:

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

An overview of version 1.0 of these theories appeared in May 2011 as (Kahl, 2011b); some of the developments reported in (Kahl, 2012) are also included here.

1.1 Introduction to Agda: Types, Sets, Equality

The Agda home page[1] states:

> **Agda is a dependently typed functional programming language.** It has inductive families, i.e., data types which depend on values, such as the type of vectors of a given length. It also has parametrised modules, mixfix operators, Unicode characters, and an interactive Emacs interface which can assist the programmer in writing the program.
>
> **Agda is a proof assistant.** It is an interactive system for writing and checking proofs. Agda is based

[1] `http://wiki.portal.chalmers.se/agda/`

> on intuitionistic type theory, a foundational system for constructive mathematics developed by the Swedish logician Per Martin-Löf. It has many similarities with other proof assistants based on dependent types, such as Coq, Epigram, Matita and NuPRL.

Syntactically and "culturally", Agda is quite close to Haskell. However, since Agda is strongly normalising and has no $\perp$ values, the underlying semantics is quite different. Also, since Agda is dependently typed, it does not have Haskell's distinction between terms, types, and kinds (the "types of the types"). The Agda constant Set corresponds to the Haskell kind *; it is the type of all "normal" datatypes. For example, the Agda standard library defines the type Bool as follows:

```
data Bool : Set where true : Bool
                      false : Bool
```

Since Set needs again a type, there is Set_1, with Set : Set_1, etc., resulting in a hierarchy of "universes". Since Version 2.2.8, Agda supports *universe polymorphism*, with universes Set i where i : Level is an element a special-purpose variant of the natural numbers. we use the following notation, defined in RATH.Level as renaming of the standard-library's Level module:

- ℓ_0 : Level is the lowest level.
- ℓsuc : Level → Level is the Level successor function.
- _⊍_ : Level → Level → Level is the maximum operation.

With this, the conventional usage of Set and Set_1 turns into syntactic sugar, with Set standing for Set ℓ_0, and Set_1 = Set (ℓsuc ℓ_0).[2]

With universe polymorphism, we may quantify over Level-typed variables that occur as Level arguments of Set. Universe polymorphism is essential for being able to talk about both "small" and "large" categories or relation algebras, or, for another example, also for being able to treat diagrams of graphs and graph homomorphisms as graphs again. We therefore use universe polymorphism throughout this paper.

For example, the standard library includes the following definition for the universe-polymorphic parameterised Maybe type:

```
data Maybe {a : Level} (A : Set a) : Set a where just : (x : A) → Maybe A
                                                 nothing : Maybe A
```

Maybe has two parameters, a and A, where dependent typing is used since the type of the second parameter depends on the first parameter. The use of {...} flags a as an *implicit parameter* that can be elided where its type is implied by the call site of Maybe. This happens in the occurrences of Maybe A in the types of the data constructors just and nothing: In Maybe A, the value of the first, implicit parameter of Maybe can only be a, the level of the set A.

The same applies to implicit function arguments, and in most cases, implicit arguments or parameters are determined by later arguments respectively parameters. Frequently, implicit arguments correspond quite precisely to the implicit context of mathematical statements, so that the reader may be advised to skip implicit arguments at first reading of a type, and return to them for clarification where necessary for understanding the types of the explicit parameters.

While the Hindley-Milner typing of Haskell and ML allows function definitions without declaration of the function type, and type signatures without declaration of the universally quantified type variables, in Agda, all types and variables need to be declared, but implicit parameters and the type checking machinery used to resolve them alleviate that burden significantly. For example, the original definition writes only Maybe {a} (A : Set a) : Set a, since the type of a will be inferred from a's use as argument to Set. In this paper, we will rarely use this possibility to elide types of named arguments, since we estimate that the clarity of explicit typing is worth the additional "optical noise" especially for readers who are less familiar with Agda or dependently-typed theories.

[2] The standard library module Level exports zero and suc as Level construction funtions, and uses "⊔" for maximum on Level. In our development, we systematically rename this frequently-used maximum operation to "⊍", so that we can use "⊔" as join in the inclusion order of morphisms, as customary in abstract relation algebra Schmidt and Ströhlein (1993); Schmidt et al. (1997).

The "programming types" like Maybe can be freely mixed with "formula types", inspired by the Curry-Howard-correspondence of "formulae as types, proofs as programs". The formula types of true formulae contain their proofs, while the formula types of false formulae are empty.

The standard library type of propositional equality has (besides two implicit parameters) one explicit parameter and one explicit argument; the definition therefore gives rise to types like the type "2 ≡ 1 + 1", which can be shown to be inhabited using the definition of natural numbers and natural number addition +, and the type "2 ≡ 3", which is an empty type, since it has no proof. [3]

```
data _≡_ {a : Level} {A : Set a} (x : A) : A → Set a where refl : x ≡ x
```

The definition introduces types x ≡ y for any x and y of type A, but only the types x ≡ x are inhabited, and they contain the single element refl {a} {A} {x}.

In Agda, as in other type theories without quotient types, sets with equality are typically modelled as *setoids*, that is, carrier types equipped with an equivalence. This closely corresponds to the non-primitive nature of the "equality" test (==) : Eq a ⇒ a → a → Bool in Haskell.

The standard library defines the following type of homogeneous relations:

```
Rel : {a : Level} → Set a → (ℓ : Level) → Set (a ⊍ ℓsuc ℓ)
Rel A ℓ = A → A → Set ℓ
```

A proof that _≈_ is an equivalence relation is a record containing the proofs of reflexivity, symmetry, and transitivity:

```
record IsEquivalence {a ℓ : Level} {A : Set a} (_≈_ : Rel A ℓ) : Set (a ⊍ ℓ) where
  field refl  : {x : A} → x ≈ x
        sym   : {x y : A} → x ≈ y → y ≈ x
        trans : {x y z : A} → x ≈ y → y ≈ z → x ≈ z
```

A setoid is a dependent record consisting of a Carrier set, a relation _≈_ on that carrier, and a proof that that relation is an equivalence relation:

```
record Setoid c ℓ : Set (ℓsuc (c ⊍ ℓ)) where
  field Carrier : Set c
        _≈_     : Rel Carrier ℓ
        isEquivalence : IsEquivalence _≈_
  open IsEquivalence isEquivalence public
```

An Agda record is also a module that may contain other material besides its **field**s; the "**open**" clause makes the fields of the equivalence proof available as if they were fields of Setoid. This language feature enables incremental extension of smaller theories to larger theories at very low notational cost.

The Preorder type of the Agda standard library adds a second preorder relation to a Setoid, with reflexivity with respect to the setoid equality; for a Poset, that preorder also needs to be antisymmetric with respect to the setoid equality.

1.2 Related Work

Our approach to categories with setoids of morphisms, but not of objects, derives essentially from Kanda's "effective categories" Kanda (1981); it is also used by Huet and Saïbi (1998; 2000) for their formalisation of category theory in Coq, and by Gonzalía (2006), who produced formalisations of concrete heterogeneous binary relations and of the allegory hierarchy of Freyd and Scedrov (1990) in Alf, a predecessor of Agda.

Mu et al. (2009) have contributed Agda2 theories of concrete relations inspired by the *Algebra of Programming* of Bird and de Moor (1997); they note the advantages that Agda2 brought to their formalisations of concrete relations over the Alf formalisations of Gonzalía.

[3] In Agda, almost all lexemes are separated by spaces, since almost all symbol combinations form legal names. Underscores as part of names indicate positions of explicit arguments for mixfix operators.

1.3 Overview

Each chapter contains its own local overview, so we can keep the general overview here quite concise.

Each section is the document view of a literate Agda module file, processed by `lhs2TeX -agda`, but the module head is hidden from the document view. These hidden module heads are contained in the distributed source files; they contain only imports, which are typically clear from the conceptual dependencies of each theory, and therefore do not contribute significantly to understanding, but would instead be rather detracting.

In Chapter 2, we collect our "buffer layer" to the Agda standard library of Danielsson et al. (2013), consisting mostly of minor renamings and extensions.

We use semigroupoids as our most general framework; these are essentially "categories without identities". In Chapter 3 we introduce semigroupoids, categories, and also variants of both with an additional converse operator, turning them into self-dual structures. Foundations and tools for common finite limits and colimits are defined Chapter 4. We add a simple indexed product construction for semigroupoids and categories in Chapter 5. Functors are defined first for semigroupoids and then for categories in Chapter 6; bifunctors and natural transformation (natural isomorphisms) are used in Chapter 7 for monoidal categories.

As preparation for moving from "function theories" to "relation theories", Chapter 8 presents auxiliary definitions mostly concerning posets, including dualisation and lattices.

In Chapter 9, we add local order (on each homset) to semigroupoids and categories, starting with general partial orders, then moving over semi-lattices to (distributive) lattices.

One of the classical relation-algebraic operations that can be characterised in surprisingly simple settings is domain; we present fromalisations of domain concepts in semigroupoids and in ordered semigroupoids in Chapter 10.

Many of the typically relation-algebraic definitions of relation properties can be formalised in ordered categories with converse (OCCs), or even in ordered semigroupoids with converse (OSGCs), where sometimes the absence of identities in the latter imposes "more expensive" definitions. We show both variants, where applicable, in Chapter 11.

In Chapter 12 we formalise allegories, distributive allegories, and related theories. Residuals and division allegories are then the topic of Chapter 13, while Kleene star and the related theories are treated in Chapter 14. Chapters 15 and 16 present direct sums and cotabulations, respectively.

Already in semigroupoids with converse we can characterise "partial equivalence relations" (PER). Taking the "quotient" of an object by a PER essentially means constructing a new semigroupoid (category) that has as objects all PERs on the original objects (Chapter 17).

Part III is devoted to "concrete relations" with a standard type-theoretic definition, compatible with that of the Agda standard library.

Chapter 18 presents a universe-polymorphic formalisation of concrete relations that will later serve in models of the abstract theories.

For many of the abstract theories of Part I and Part II, we show in Chapter 19 that the concrete relations of Chapter 18 populate models.

Moving from Sets to $\mathsf{Setoids}$, we show in Chapter 20 a category of $\mathsf{Setoids}$.

In Chapter 21, we build a modular framework for constructing instances of the abstract theories of Part I and Part II from concrete data types providing an elementship relation.

Part IV first presents the straight-forward construction of product semigroupoids and categories in Chapter 22, and then extends the construction of Chapter 5 also to the interfaces of Part II in Chapter 23.

1.4 RathAgda2All

This module contains entry points from which all modules that are part of the package RATH-Agda-2 are reached.

import Relation.Binary.Heterogeneous
import Categoric
import Relation.Binary.Heterogeneous.Categoric
import Categoric.SortIndexedProduct
import Categoric.PERQ

import Categoric.Semigroupoid.Monolithic
import Categoric.Category.Monolithic
import Categoric.OSGC.Monolithic
import Categoric.USLCCZ.Monolithic

import Categoric.Category.Slice
import Categoric.SGFunctor.Inverse0
import Categoric.Functor.CoEqualiser
import Categoric.MonoidalCategory.Coproducts
import Categoric.MonoidalCategory.ProductGS

import Categoric.Product.Category

import Categoric.Diagram.Examples
import Categoric.Diagram.CompOp

import Relation.Binary.ElemRel.All

The following refers to a 2010 machine running Linux on a six-core 2.8GHZ 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 85 minutes using the following incantation:

```
STDLIB=/usr/local/packages/Agda-2.3.3.8/build/lib/src  # adapt to your installation!

agda +RTS -K64M -S -M10G -H10G -RTS -i . -i $STDLIB  RathAgda2All.lagda
```

For checking this module as a whole, less than 10GB for the heap will not work.

Checking the imports individually however works in 6GB — Categoric.lagda will go through only in the second attempt:

```
for i in Categoric.lagda $(grep '^import ' RathAgda2All.lagda | awk '{print $2}')
do
  agda +RTS -K64M -S -M6G -H6G -RTS -i . -i $STDLIB $(echo $i | tr . /).lagda
done
```

(Some of the individual theories in Categoric probably will not go through in much less. Individually checking RathAgda2All.lagda at the end requires more than 6GB; it works with 6.5GB.)

Chapter 2

Standard Library Wrappers and Extensions

2.1 RATH.Level

```
open import Level public
  renaming (_⊔_ to _⊍_; zero to ℓ₀; suc to ℓsuc)
```

2.2 Relation.Binary.EqReasoning.Extended

We extend Relation.Binary.EqReasoning of the standard library with some additional functionality.

```
open Setoid S
import Relation.Binary.EqReasoning as EqR
open EqR S public renaming (_≡⟨_⟩_ to _≈≡⟨_⟩_; _≡⟨⟩_ to _≈⟨⟩_)
infixr 2 _≈˘⟨_⟩_ _≈≡˘⟨_⟩_
_≈˘⟨_⟩_ : (x : Carrier) {y z : Carrier}
  → y ≈ x → y IsRelatedTo z → x IsRelatedTo z
_ ≈˘⟨ y≈x ⟩ relTo y≈z = relTo (trans (sym y≈x) y≈z)
_≈≡˘⟨_⟩_ : (x : Carrier) → {y z : Carrier}
            → y ≡ x → y IsRelatedTo z → x IsRelatedTo z
_ ≈≡˘⟨ y≡x ⟩ relTo y≈z = relTo (trans (sym (reflexive y≡x)) y≈z)
```

2.3 Relation.Binary.Setoid.Utils

```
⌊_⌋ : {a ℓ : Level} → Setoid a ℓ → Set a
⌊ s ⌋ = Setoid.Carrier s
⌈_⌉ : {a : Level} → Set a → Setoid a a
⌈ A ⌉ = ≡-setoid A
```

```
infix 4 inSetoidEquiv
inSetoidEquiv : {a ℓ : Level} → (S : Setoid a ℓ) → ⌊ S ⌋ → ⌊ S ⌋ → Set ℓ
inSetoidEquiv = Setoid._≈_
syntax inSetoidEquiv S x y = x ≈⌊ S ⌋ y
```

```
Setoid₀₀ = Setoid ℓ₀ ℓ₀
Setoid₁₀ = Setoid (ℓsuc ℓ₀) ℓ₀
```

```
Setoid₀₁ = Setoid ℓ₀ (ℓsuc ℓ₀)
Setoid₁₁ = Setoid (ℓsuc ℓ₀) (ℓsuc ℓ₀)
```

```
retractIsEquivalence : {i₁ i₂ k : Level} {A : Set i₁} {B : Set i₂} (f : B → A)
                     → {_≈_ : Rel A k} → IsEquivalence _≈_ → IsEquivalence (_≈_ on f)
retractIsEquivalence f {_≈_} isEquiv = let module I = IsEquivalence isEquiv
  in record {refl = I.refl; sym = I.sym; trans = I.trans}
retractSetoid : {i₁ i₂ k : Level} {B : Set i₂} (A : Setoid i₁ k) (f : B → ⌊ A ⌋)
  → Setoid i₂ k
retractSetoid {B = B} A f = let module A = Setoid A in record
  {Carrier = B
  ; _≈_ = A._≈_ on f
  ; isEquivalence = retractIsEquivalence f A.isEquivalence
  }
```

```
funRel : {i₁ i₂ k₁ k₂ : Level} {A : Setoid i₁ k₁} {B : Setoid i₂ k₂}
       → (F : A ⟶ B) → ⌊ A ⌋ → ⌊ B ⌋ → Set k₂
funRel {B = B} F x y = F ⟨$⟩ x ≈⌊ B ⌋ y
```

In the “**open** Setoid S **public**” below, everything except Carrier and _≈_ is explicitly renamed.

```
module Setoid′ {i j : Level} (S : Setoid i j) where
  ℓ = i ⊍ j
  open Setoid S public renaming
    (refl           to ≈-refl           -- : {R : Carrier} → R ≈ R
    ; reflexive     to ≈-reflexive      -- : {R S : Carrier} → R ≡ S → R ≈ R
    ; sym           to ≈-sym            -- : {R S : Carrier} → R ≈ S → S ≈ R
    ; trans         to ≈-trans          -- : {Q R S : Carrier} → Q ≈ R → R ≈ S → Q ≈ S
    ; isEquivalence to ≈-isEquivalence  -- : IsEquivalence _≈_
    ; isPreorder    to ≈-isPreorder     -- : IsPreorder _≡_ _≈_
    ; preorder      to ≈-preorder       -- : Preorder i i j
    ; indexedSetoid to ≈-indexedSetoid
    )
  ≈-trans₁ : {Q R S : Carrier} → Q ≈ R → R ≡ S → Q ≈ S
  ≈-trans₁ Q≈R R≡S = ≈-trans Q≈R (≈-reflexive R≡S)
  ≈-trans₂ : {Q R S : Carrier} → Q ≡ R → R ≈ S → Q ≈ S
  ≈-trans₂ Q≡R R≈S = ≈-trans (≈-reflexive Q≡R) R≈S
```

We add a number of infix operators for combining proof steps:

```
  infixl 1 _⟨≈≈⟩_ _⟨≈≈˘⟩_ _⟨≈˘≈⟩_ _⟨≈˘≈˘⟩_
  _⟨≈≈⟩_ : {Q R S : Carrier} → Q ≈ R → R ≈ S → Q ≈ S
  _⟨≈≈⟩_ = ≈-trans
  _⟨≈≈˘⟩_ : {Q R S : Carrier} → Q ≈ R → S ≈ R → Q ≈ S
  _⟨≈≈˘⟩_ x y = ≈-trans x (≈-sym y)
  _⟨≈˘≈⟩_ : {Q R S : Carrier} → R ≈ Q → R ≈ S → Q ≈ S
  _⟨≈˘≈⟩_ x y = ≈-trans (≈-sym x) y
  _⟨≈˘≈˘⟩_ : {Q R S : Carrier} → R ≈ Q → S ≈ R → Q ≈ S
  _⟨≈˘≈˘⟩_ x y = ≈-trans (≈-sym x) (≈-sym y)
  infixl 1 _⟨≡≈⟩_ _⟨≡≈˘⟩_ _⟨≡˘≈⟩_ _⟨≡˘≈˘⟩_
  _⟨≡≈⟩_ : {Q R S : Carrier} → Q ≡ R → R ≈ S → Q ≈ S
  _⟨≡≈⟩_ x y = ≈-reflexive x ⟨≈≈⟩ y
  _⟨≡≈˘⟩_ : {Q R S : Carrier} → Q ≡ R → S ≈ R → Q ≈ S
  _⟨≡≈˘⟩_ x y = ≈-reflexive x ⟨≈≈˘⟩ y
  _⟨≡˘≈⟩_ : {Q R S : Carrier} → R ≡ Q → R ≈ S → Q ≈ S
  _⟨≡˘≈⟩_ x y = ≈-reflexive x ⟨≈˘≈⟩ y
```

```
  _⟨≡˘≈˘⟩_ : {Q R S : Carrier} → R ≡ Q → S ≈ R → Q ≈ S
  _⟨≡˘≈˘⟩_ x y = ≈-reflexive x ⟨≈˘≈˘⟩ y
  infixl 1 _⟨≈≡⟩_ _⟨≈≡˘⟩_ _⟨≈˘≡⟩_ _⟨≈˘≡˘⟩_
  _⟨≈≡⟩_ : {Q R S : Carrier} → Q ≈ R → R ≡ S → Q ≈ S
  _⟨≈≡⟩_ x y = x ⟨≈≈⟩ ≈-reflexive y
  _⟨≈≡˘⟩_ : {Q R S : Carrier} → Q ≈ R → S ≡ R → Q ≈ S
  _⟨≈≡˘⟩_ x y = x ⟨≈≈˘⟩ ≈-reflexive y
  _⟨≈˘≡⟩_ : {Q R S : Carrier} → R ≈ Q → R ≡ S → Q ≈ S
  _⟨≈˘≡⟩_ x y = x ⟨≈˘≈⟩ ≈-reflexive y
  _⟨≈˘≡˘⟩_ : {Q R S : Carrier} → R ≈ Q → S ≡ R → Q ≈ S
  _⟨≈˘≡˘⟩_ x y = x ⟨≈˘≈˘⟩ ≈-reflexive y
```

We also re-export the equational reasoning combinators, with slight renaming. We keep these in separate modules since in the context of preorder reasoning, we use more general implementation of these combinators, but still want to use the Setoid′ material (or its renamings).

```
module SetoidCalc {i j : Level} (S : Setoid i j) =
  Relation.Binary.EqReasoning.Extended S renaming
    (begin_ to ≈-begin_; _■ to _□
    ; _IsRelatedTo_ to _≈-IsRelatedTo_
    ; module _IsRelatedTo_ to _≈-IsRelatedTo_
    )
```

For contexts where calculation in different setoids is necessary, we provide "decorated" versions of the Setoid′ and SetoidCalc interfaces:

```
module SetoidA {i j : Level} (S : Setoid i j) = Setoid′ S renaming
  (ℓ to ℓA; Carrier to A₀; _≈_ to _≈A_; ≈-isEquivalence to ≈A-isEquivalence
  ; ≈-isPreorder to ≈A-isPreorder; ≈-preorder to ≈A-preorder
  ; ≈-indexedSetoid to ≈A-indexedSetoid
  ; ≈-refl to ≈A-refl; ≈-reflexive to ≈A-reflexive; ≈-sym to ≈A-sym
  ; ≈-trans to ≈A-trans; ≈-trans₁ to ≈A-trans₁; ≈-trans₂ to ≈A-trans₂
  ; _⟨≈≈⟩_ to _⟨≈A≈⟩_; _⟨≈≈˘⟩_ to _⟨≈A≈˘⟩_; _⟨≈˘≈⟩_ to _⟨≈A˘≈⟩_; _⟨≈˘≈˘⟩_ to _⟨≈A˘≈˘⟩_
  ; _⟨≡≈⟩_ to _⟨≡≈A⟩_; _⟨≡≈˘⟩_ to _⟨≡≈A˘⟩_; _⟨≡˘≈⟩_ to _⟨≡˘≈A⟩_; _⟨≡˘≈˘⟩_ to _⟨≡˘≈A˘⟩_
  ; _⟨≈≡⟩_ to _⟨≈A≡⟩_; _⟨≈≡˘⟩_ to _⟨≈A≡˘⟩_; _⟨≈˘≡⟩_ to _⟨≈A˘≡⟩_; _⟨≈˘≡˘⟩_ to _⟨≈A˘≡˘⟩_
  )
module SetoidB {i j : Level} (S : Setoid i j) = Setoid′ S renaming
  (ℓ to ℓB; Carrier to B₀; _≈_ to _≈B_; ≈-isEquivalence to ≈B-isEquivalence
  ; ≈-isPreorder to ≈B-isPreorder; ≈-preorder to ≈B-preorder
  ; ≈-indexedSetoid to ≈B-indexedSetoid
  ; ≈-refl to ≈B-refl; ≈-reflexive to ≈B-reflexive; ≈-sym to ≈B-sym
  ; ≈-trans to ≈B-trans; ≈-trans₁ to ≈B-trans₁; ≈-trans₂ to ≈B-trans₂
  ; _⟨≈≈⟩_ to _⟨≈B≈⟩_; _⟨≈≈˘⟩_ to _⟨≈B≈˘⟩_; _⟨≈˘≈⟩_ to _⟨≈B˘≈⟩_; _⟨≈˘≈˘⟩_ to _⟨≈B˘≈˘⟩_
  ; _⟨≡≈⟩_ to _⟨≡≈B⟩_; _⟨≡≈˘⟩_ to _⟨≡≈B˘⟩_; _⟨≡˘≈⟩_ to _⟨≡˘≈B⟩_; _⟨≡˘≈˘⟩_ to _⟨≡˘≈B˘⟩_
  ; _⟨≈≡⟩_ to _⟨≈B≡⟩_; _⟨≈≡˘⟩_ to _⟨≈B≡˘⟩_; _⟨≈˘≡⟩_ to _⟨≈B˘≡⟩_; _⟨≈˘≡˘⟩_ to _⟨≈B˘≡˘⟩_
  )
module SetoidC {i j : Level} (S : Setoid i j) = Setoid′ S renaming
  (ℓ to ℓC; Carrier to C₀; _≈_ to _≈C_; ≈-isEquivalence to ≈C-isEquivalence
  ; ≈-isPreorder to ≈C-isPreorder; ≈-preorder to ≈C-preorder
  ; ≈-indexedSetoid to ≈C-indexedSetoid
  ; ≈-refl to ≈C-refl; ≈-reflexive to ≈C-reflexive; ≈-sym to ≈C-sym
  ; ≈-trans to ≈C-trans; ≈-trans₁ to ≈C-trans₁; ≈-trans₂ to ≈C-trans₂
  ; _⟨≈≈⟩_ to _⟨≈C≈⟩_; _⟨≈≈˘⟩_ to _⟨≈C≈˘⟩_; _⟨≈˘≈⟩_ to _⟨≈C˘≈⟩_; _⟨≈˘≈˘⟩_ to _⟨≈C˘≈˘⟩_
  ; _⟨≡≈⟩_ to _⟨≡≈C⟩_; _⟨≡≈˘⟩_ to _⟨≡≈C˘⟩_; _⟨≡˘≈⟩_ to _⟨≡˘≈C⟩_; _⟨≡˘≈˘⟩_ to _⟨≡˘≈C˘⟩_
  ; _⟨≈≡⟩_ to _⟨≈C≡⟩_; _⟨≈≡˘⟩_ to _⟨≈C≡˘⟩_; _⟨≈˘≡⟩_ to _⟨≈C˘≡⟩_; _⟨≈˘≡˘⟩_ to _⟨≈C˘≡˘⟩_
  )
module SetoidD {i j : Level} (S : Setoid i j) = Setoid′ S renaming
```

```
  (ℓ to ℓD; Carrier to D₀; _≈_ to _≈D_; ≈-isEquivalence to ≈D-isEquivalence
  ; ≈-isPreorder to ≈D-isPreorder; ≈-preorder to ≈D-preorder
  ; ≈-indexedSetoid to ≈D-indexedSetoid
  ; ≈-refl to ≈D-refl; ≈-reflexive to ≈D-reflexive; ≈-sym to ≈D-sym
  ; ≈-trans to ≈D-trans; ≈-trans₁ to ≈D-trans₁; ≈-trans₂ to ≈D-trans₂
  ; _⟨≈≈⟩_ to _⟨≈D≈⟩_; _⟨≈≈˘⟩_ to _⟨≈D≈˘⟩_; _⟨≈˘≈⟩_ to _⟨≈D˘≈⟩_; _⟨≈˘≈˘⟩_ to _⟨≈D˘≈˘⟩_
  ; _⟨≡≈⟩_ to _⟨≡≈D⟩_; _⟨≡≈˘⟩_ to _⟨≡≈D˘⟩_; _⟨≡˘≈⟩_ to _⟨≡˘≈D⟩_; _⟨≡˘≈˘⟩_ to _⟨≡˘≈D˘⟩_
  ; _⟨≈≡⟩_ to _⟨≈D≡⟩_; _⟨≈≡˘⟩_ to _⟨≈D≡˘⟩_; _⟨≈˘≡⟩_ to _⟨≈D˘≡⟩_; _⟨≈˘≡˘⟩_ to _⟨≈D˘≡˘⟩_
  )
module SetoidK {i j : Level} (S : Setoid i j) = Setoid′ S renaming
  (ℓ to ℓK; Carrier to K₀; _≈_ to _≈K_; ≈-isEquivalence to ≈K-isEquivalence
  ; ≈-isPreorder to ≈K-isPreorder; ≈-preorder to ≈K-preorder
  ; ≈-indexedSetoid to ≈K-indexedSetoid
  ; ≈-refl to ≈K-refl; ≈-reflexive to ≈K-reflexive; ≈-sym to ≈K-sym
  ; ≈-trans to ≈K-trans; ≈-trans₁ to ≈K-trans₁; ≈-trans₂ to ≈K-trans₂
  ; _⟨≈≈⟩_ to _⟨≈K≈⟩_; _⟨≈≈˘⟩_ to _⟨≈K≈˘⟩_; _⟨≈˘≈⟩_ to _⟨≈K˘≈⟩_; _⟨≈˘≈˘⟩_ to _⟨≈K˘≈˘⟩_
  ; _⟨≡≈⟩_ to _⟨≡≈K⟩_; _⟨≡≈˘⟩_ to _⟨≡≈K˘⟩_; _⟨≡˘≈⟩_ to _⟨≡˘≈K⟩_; _⟨≡˘≈˘⟩_ to _⟨≡˘≈K˘⟩_
  ; _⟨≈≡⟩_ to _⟨≈K≡⟩_; _⟨≈≡˘⟩_ to _⟨≈K≡˘⟩_; _⟨≈˘≡⟩_ to _⟨≈K˘≡⟩_; _⟨≈˘≡˘⟩_ to _⟨≈K˘≡˘⟩_
  )
module SetoidL {i j : Level} (S : Setoid i j) = Setoid′ S renaming
  (ℓ to ℓL; Carrier to L₀; _≈_ to _≈L_; ≈-isEquivalence to ≈L-isEquivalence
  ; ≈-isPreorder to ≈L-isPreorder; ≈-preorder to ≈L-preorder
  ; ≈-indexedSetoid to ≈L-indexedSetoid
  ; ≈-refl to ≈L-refl; ≈-reflexive to ≈L-reflexive; ≈-sym to ≈L-sym
  ; ≈-trans to ≈L-trans; ≈-trans₁ to ≈L-trans₁; ≈-trans₂ to ≈L-trans₂
  ; _⟨≈≈⟩_ to _⟨≈L≈⟩_; _⟨≈≈˘⟩_ to _⟨≈L≈˘⟩_; _⟨≈˘≈⟩_ to _⟨≈L˘≈⟩_; _⟨≈˘≈˘⟩_ to _⟨≈L˘≈˘⟩_
  ; _⟨≡≈⟩_ to _⟨≡≈L⟩_; _⟨≡≈˘⟩_ to _⟨≡≈L˘⟩_; _⟨≡˘≈⟩_ to _⟨≡˘≈L⟩_; _⟨≡˘≈˘⟩_ to _⟨≡˘≈L˘⟩_
  ; _⟨≈≡⟩_ to _⟨≈L≡⟩_; _⟨≈≡˘⟩_ to _⟨≈L≡˘⟩_; _⟨≈˘≡⟩_ to _⟨≈L˘≡⟩_; _⟨≈˘≡˘⟩_ to _⟨≈L˘≡˘⟩_
  )
module SetoidR {i j : Level} (S : Setoid i j) = Setoid′ S renaming
  (ℓ to ℓR; Carrier to R₀; _≈_ to _≈R_; ≈-isEquivalence to ≈R-isEquivalence
  ; ≈-isPreorder to ≈R-isPreorder; ≈-preorder to ≈R-preorder
  ; ≈-indexedSetoid to ≈R-indexedSetoid
  ; ≈-refl to ≈R-refl; ≈-reflexive to ≈R-reflexive; ≈-sym to ≈R-sym
  ; ≈-trans to ≈R-trans; ≈-trans₁ to ≈R-trans₁; ≈-trans₂ to ≈R-trans₂
  ; _⟨≈≈⟩_ to _⟨≈R≈⟩_; _⟨≈≈˘⟩_ to _⟨≈R≈˘⟩_; _⟨≈˘≈⟩_ to _⟨≈R˘≈⟩_; _⟨≈˘≈˘⟩_ to _⟨≈R˘≈˘⟩_
  ; _⟨≡≈⟩_ to _⟨≡≈R⟩_; _⟨≡≈˘⟩_ to _⟨≡≈R˘⟩_; _⟨≡˘≈⟩_ to _⟨≡˘≈R⟩_; _⟨≡˘≈˘⟩_ to _⟨≡˘≈R˘⟩_
  ; _⟨≈≡⟩_ to _⟨≈R≡⟩_; _⟨≈≡˘⟩_ to _⟨≈R≡˘⟩_; _⟨≈˘≡⟩_ to _⟨≈R˘≡⟩_; _⟨≈˘≡˘⟩_ to _⟨≈R˘≡˘⟩_
  )
module SetoidS {i j : Level} (S : Setoid i j) = Setoid′ S renaming
  (ℓ to ℓS; Carrier to S₀; _≈_ to _≈S_; ≈-isEquivalence to ≈S-isEquivalence
  ; ≈-isPreorder to ≈S-isPreorder; ≈-preorder to ≈S-preorder
  ; ≈-indexedSetoid to ≈S-indexedSetoid
  ; ≈-refl to ≈S-refl; ≈-reflexive to ≈S-reflexive; ≈-sym to ≈S-sym
  ; ≈-trans to ≈S-trans; ≈-trans₁ to ≈S-trans₁; ≈-trans₂ to ≈S-trans₂
  ; _⟨≈≈⟩_ to _⟨≈S≈⟩_; _⟨≈≈˘⟩_ to _⟨≈S≈˘⟩_; _⟨≈˘≈⟩_ to _⟨≈S˘≈⟩_; _⟨≈˘≈˘⟩_ to _⟨≈S˘≈˘⟩_
  ; _⟨≡≈⟩_ to _⟨≡≈S⟩_; _⟨≡≈˘⟩_ to _⟨≡≈S˘⟩_; _⟨≡˘≈⟩_ to _⟨≡˘≈S⟩_; _⟨≡˘≈˘⟩_ to _⟨≡˘≈S˘⟩_
  ; _⟨≈≡⟩_ to _⟨≈S≡⟩_; _⟨≈≡˘⟩_ to _⟨≈S≡˘⟩_; _⟨≈˘≡⟩_ to _⟨≈S˘≡⟩_; _⟨≈˘≡˘⟩_ to _⟨≈S˘≡˘⟩_
  )
module Setoid₀ {i j : Level} (S : Setoid i j) = Setoid′ S renaming
  (ℓ to ℓS₀; Carrier to Carrier₀; _≈_ to _≈₀_; ≈-isEquivalence to ≈₀-isEquivalence
  ; ≈-isPreorder to ≈₀-isPreorder; ≈-preorder to ≈₀-preorder
  ; ≈-indexedSetoid to ≈₀-indexedSetoid
  ; ≈-refl to ≈₀-refl; ≈-reflexive to ≈₀-reflexive; ≈-sym to ≈₀-sym
  ; ≈-trans to ≈₀-trans; ≈-trans₁ to ≈₀-trans₁; ≈-trans₂ to ≈₀-trans₂
  ; _⟨≈≈⟩_ to _⟨≈₀≈⟩_; _⟨≈≈˘⟩_ to _⟨≈₀≈˘⟩_; _⟨≈˘≈⟩_ to _⟨≈₀˘≈⟩_; _⟨≈˘≈˘⟩_ to _⟨≈₀˘≈˘⟩_
  ; _⟨≡≈⟩_ to _⟨≡≈₀⟩_; _⟨≡≈˘⟩_ to _⟨≡≈₀˘⟩_; _⟨≡˘≈⟩_ to _⟨≡˘≈₀⟩_; _⟨≡˘≈˘⟩_ to _⟨≡˘≈₀˘⟩_
  ; _⟨≈≡⟩_ to _⟨≈₀≡⟩_; _⟨≈≡˘⟩_ to _⟨≈₀≡˘⟩_; _⟨≈˘≡⟩_ to _⟨≈₀˘≡⟩_; _⟨≈˘≡˘⟩_ to _⟨≈₀˘≡˘⟩_
```

```
  )
module Setoid₁ {i j : Level} (S : Setoid i j) = Setoid′ S renaming
  (ℓ to ℓS₁; Carrier to Carrier₁; _≈_ to _≈₁_; ≈-isEquivalence to ≈₁-isEquivalence
  ; ≈-isPreorder to ≈₁-isPreorder; ≈-preorder to ≈₁-preorder
  ; ≈-indexedSetoid to ≈₁-indexedSetoid
  ; ≈-refl to ≈₁-refl; ≈-reflexive to ≈₁-reflexive; ≈-sym to ≈₁-sym
  ; ≈-trans to ≈₁-trans; ≈-trans₁ to ≈₁-trans₁; ≈-trans₂ to ≈₁-trans₂
  ; _⟨≈≈⟩_ to _⟨≈₁≈⟩_; _⟨≈≈˘⟩_ to _⟨≈₁≈˘⟩_; _⟨≈˘≈⟩_ to _⟨≈₁˘≈⟩_; _⟨≈˘≈˘⟩_ to _⟨≈₁˘≈˘⟩_
  ; _⟨≡≈⟩_ to _⟨≡≈₁⟩_; _⟨≡≈˘⟩_ to _⟨≡≈₁˘⟩_; _⟨≡˘≈⟩_ to _⟨≡˘≈₁⟩_; _⟨≡˘≈˘⟩_ to _⟨≡˘≈₁˘⟩_
  ; _⟨≈≡⟩_ to _⟨≈₁≡⟩_; _⟨≈≡˘⟩_ to _⟨≈₁≡˘⟩_; _⟨≈˘≡⟩_ to _⟨≈₁˘≡⟩_; _⟨≈˘≡˘⟩_ to _⟨≈₁˘≡˘⟩_
  )
module Setoid₂ {i j : Level} (S : Setoid i j) = Setoid′ S renaming
  (ℓ to ℓS₂; Carrier to Carrier₂; _≈_ to _≈₂_; ≈-isEquivalence to ≈₂-isEquivalence
  ; ≈-isPreorder to ≈₂-isPreorder; ≈-preorder to ≈₂-preorder
  ; ≈-indexedSetoid to ≈₂-indexedSetoid
  ; ≈-refl to ≈₂-refl; ≈-reflexive to ≈₂-reflexive; ≈-sym to ≈₂-sym
  ; ≈-trans to ≈₂-trans; ≈-trans₁ to ≈₂-trans₁; ≈-trans₂ to ≈₂-trans₂
  ; _⟨≈≈⟩_ to _⟨≈₂≈⟩_; _⟨≈≈˘⟩_ to _⟨≈₂≈˘⟩_; _⟨≈˘≈⟩_ to _⟨≈₂˘≈⟩_; _⟨≈˘≈˘⟩_ to _⟨≈₂˘≈˘⟩_
  ; _⟨≡≈⟩_ to _⟨≡≈₂⟩_; _⟨≡≈˘⟩_ to _⟨≡≈₂˘⟩_; _⟨≡˘≈⟩_ to _⟨≡˘≈₂⟩_; _⟨≡˘≈˘⟩_ to _⟨≡˘≈₂˘⟩_
  ; _⟨≈≡⟩_ to _⟨≈₂≡⟩_; _⟨≈≡˘⟩_ to _⟨≈₂≡˘⟩_; _⟨≈˘≡⟩_ to _⟨≈₂˘≡⟩_; _⟨≈˘≡˘⟩_ to _⟨≈₂˘≡˘⟩_
  )
module Setoid₃ {i j : Level} (S : Setoid i j) = Setoid′ S renaming
  (ℓ to ℓS₃; Carrier to Carrier₃; _≈_ to _≈₃_; ≈-isEquivalence to ≈₃-isEquivalence
  ; ≈-isPreorder to ≈₃-isPreorder; ≈-preorder to ≈₃-preorder
  ; ≈-indexedSetoid to ≈₃-indexedSetoid
  ; ≈-refl to ≈₃-refl; ≈-reflexive to ≈₃-reflexive; ≈-sym to ≈₃-sym
  ; ≈-trans to ≈₃-trans; ≈-trans₁ to ≈₃-trans₁; ≈-trans₂ to ≈₃-trans₂
  ; _⟨≈≈⟩_ to _⟨≈₃≈⟩_; _⟨≈≈˘⟩_ to _⟨≈₃≈˘⟩_; _⟨≈˘≈⟩_ to _⟨≈₃˘≈⟩_; _⟨≈˘≈˘⟩_ to _⟨≈₃˘≈˘⟩_
  ; _⟨≡≈⟩_ to _⟨≡≈₃⟩_; _⟨≡≈˘⟩_ to _⟨≡≈₃˘⟩_; _⟨≡˘≈⟩_ to _⟨≡˘≈₃⟩_; _⟨≡˘≈˘⟩_ to _⟨≡˘≈₃˘⟩_
  ; _⟨≈≡⟩_ to _⟨≈₃≡⟩_; _⟨≈≡˘⟩_ to _⟨≈₃≡˘⟩_; _⟨≈˘≡⟩_ to _⟨≈₃˘≡⟩_; _⟨≈˘≡˘⟩_ to _⟨≈₃˘≡˘⟩_
  )
module Setoid₄ {i j : Level} (S : Setoid i j) = Setoid′ S renaming
  (ℓ to ℓS₄; Carrier to Carrier₄; _≈_ to _≈₄_; ≈-isEquivalence to ≈₄-isEquivalence
  ; ≈-isPreorder to ≈₄-isPreorder; ≈-preorder to ≈₄-preorder
  ; ≈-indexedSetoid to ≈₄-indexedSetoid
  ; ≈-refl to ≈₄-refl; ≈-reflexive to ≈₄-reflexive; ≈-sym to ≈₄-sym
  ; ≈-trans to ≈₄-trans; ≈-trans₁ to ≈₄-trans₁; ≈-trans₂ to ≈₄-trans₂
  ; _⟨≈≈⟩_ to _⟨≈₄≈⟩_; _⟨≈≈˘⟩_ to _⟨≈₄≈˘⟩_; _⟨≈˘≈⟩_ to _⟨≈₄˘≈⟩_; _⟨≈˘≈˘⟩_ to _⟨≈₄˘≈˘⟩_
  ; _⟨≡≈⟩_ to _⟨≡≈₄⟩_; _⟨≡≈˘⟩_ to _⟨≡≈₄˘⟩_; _⟨≡˘≈⟩_ to _⟨≡˘≈₄⟩_; _⟨≡˘≈˘⟩_ to _⟨≡˘≈₄˘⟩_
  ; _⟨≈≡⟩_ to _⟨≈₄≡⟩_; _⟨≈≡˘⟩_ to _⟨≈₄≡˘⟩_; _⟨≈˘≡⟩_ to _⟨≈₄˘≡⟩_; _⟨≈˘≡˘⟩_ to _⟨≈₄˘≡˘⟩_
  )

module SetoidCalcA {i j : Level} (S : Setoid i j) where
  open SetoidA S public
  open SetoidCalc S public renaming
    (_□ to _□A
    ; _≈⟨_⟩_ to _≈A⟨_⟩_
    ; _≈˘⟨_⟩_ to _≈A˘⟨_⟩_
    ; _≈≡⟨_⟩_ to _≈A≡⟨_⟩_
    ; _≈⟨⟩_ to _≈A⟨⟩_
    ; _≈≡˘⟨_⟩_ to _≈A≡˘⟨_⟩_
    ; ≈-begin_ to ≈A-begin_
    ; _≈-IsRelatedTo_ to _≈A-IsRelatedTo_
    ; module _≈-IsRelatedTo_ to _≈A-IsRelatedTo_
    )
module SetoidCalcB {i j : Level} (S : Setoid i j) where
  open SetoidB S public
  open SetoidCalc S public renaming
```

```
    ( _□ to _□B
    ; _≈⟨_⟩_ to _≈B⟨_⟩_
    ; _≈˘⟨_⟩_ to _≈B˘⟨_⟩_
    ; _≈≡⟨_⟩_ to _≈B≡⟨_⟩_
    ; _≈⟨⟩_ to _≈B⟨⟩_
    ; _≈≡˘⟨_⟩_ to _≈B≡˘⟨_⟩_
    ; ≈-begin_ to ≈B-begin_
    ; _≈-IsRelatedTo_ to _≈B-IsRelatedTo_
    ; module _≈-IsRelatedTo_ to _≈B-IsRelatedTo_
    )
module SetoidCalcC {i j : Level} (S : Setoid i j) where
  open SetoidC S public
  open SetoidCalc S public renaming
    ( _□ to _□C
    ; _≈⟨_⟩_ to _≈C⟨_⟩_
    ; _≈˘⟨_⟩_ to _≈C˘⟨_⟩_
    ; _≈≡⟨_⟩_ to _≈C≡⟨_⟩_
    ; _≈⟨⟩_ to _≈C⟨⟩_
    ; _≈≡˘⟨_⟩_ to _≈C≡˘⟨_⟩_
    ; ≈-begin_ to ≈C-begin_
    ; _≈-IsRelatedTo_ to _≈C-IsRelatedTo_
    ; module _≈-IsRelatedTo_ to _≈C-IsRelatedTo_
    )
module SetoidCalcD {i j : Level} (S : Setoid i j) where
  open SetoidD S public
  open SetoidCalc S public renaming
    ( _□ to _□D
    ; _≈⟨_⟩_ to _≈D⟨_⟩_
    ; _≈˘⟨_⟩_ to _≈D˘⟨_⟩_
    ; _≈≡⟨_⟩_ to _≈D≡⟨_⟩_
    ; _≈⟨⟩_ to _≈D⟨⟩_
    ; _≈≡˘⟨_⟩_ to _≈D≡˘⟨_⟩_
    ; ≈-begin_ to ≈D-begin_
    ; _≈-IsRelatedTo_ to _≈D-IsRelatedTo_
    ; module _≈-IsRelatedTo_ to _≈D-IsRelatedTo_
    )
module SetoidCalcK {i j : Level} (S : Setoid i j) where
  open SetoidK S public
  open SetoidCalc S public renaming
    ( _□ to _□K
    ; _≈⟨_⟩_ to _≈K⟨_⟩_
    ; _≈˘⟨_⟩_ to _≈K˘⟨_⟩_
    ; _≈≡⟨_⟩_ to _≈K≡⟨_⟩_
    ; _≈⟨⟩_ to _≈K⟨⟩_
    ; _≈≡˘⟨_⟩_ to _≈K≡˘⟨_⟩_
    ; ≈-begin_ to ≈K-begin_
    ; _≈-IsRelatedTo_ to _≈K-IsRelatedTo_
    ; module _≈-IsRelatedTo_ to _≈K-IsRelatedTo_
    )
module SetoidCalcL {i j : Level} (S : Setoid i j) where
  open SetoidL S public
  open SetoidCalc S public renaming
    ( _□ to _□L
    ; _≈⟨_⟩_ to _≈L⟨_⟩_
    ; _≈˘⟨_⟩_ to _≈L˘⟨_⟩_
    ; _≈≡⟨_⟩_ to _≈L≡⟨_⟩_
    ; _≈⟨⟩_ to _≈L⟨⟩_
    ; _≈≡˘⟨_⟩_ to _≈L≡˘⟨_⟩_
    ; ≈-begin_ to ≈L-begin_
```

```
    ; _≈-IsRelatedTo_ to _≈L-IsRelatedTo_
    ; module _≈-IsRelatedTo_ to _≈L-IsRelatedTo_
    )
module SetoidCalcR {i j : Level} (S : Setoid i j) where
  open SetoidR S public
  open SetoidCalc S public renaming
    ( _□ to _□R
    ; _≈⟨_⟩_ to _≈R⟨_⟩_
    ; _≈˘⟨_⟩_ to _≈R˘⟨_⟩_
    ; _≈≡⟨_⟩_ to _≈R≡⟨_⟩_
    ; _≈⟨⟩_ to _≈R⟨⟩_
    ; _≈≡˘⟨_⟩_ to _≈R≡˘⟨_⟩_
    ; ≈-begin_ to ≈R-begin_
    ; _≈-IsRelatedTo_ to _≈R-IsRelatedTo_
    ; module _≈-IsRelatedTo_ to _≈R-IsRelatedTo_
    )
module SetoidCalcS {i j : Level} (S : Setoid i j) where
  open SetoidS S public
  open SetoidCalc S public renaming
    ( _□ to _□S
    ; _≈⟨_⟩_ to _≈S⟨_⟩_
    ; _≈˘⟨_⟩_ to _≈S˘⟨_⟩_
    ; _≈≡⟨_⟩_ to _≈S≡⟨_⟩_
    ; _≈⟨⟩_ to _≈S⟨⟩_
    ; _≈≡˘⟨_⟩_ to _≈S≡˘⟨_⟩_
    ; ≈-begin_ to ≈S-begin_
    ; _≈-IsRelatedTo_ to _≈S-IsRelatedTo_
    ; module _≈-IsRelatedTo_ to _≈S-IsRelatedTo_
    )
module SetoidCalc₀ {i j : Level} (S : Setoid i j) where
  open Setoid₀ S public
  open SetoidCalc S public renaming
    ( _□ to _□₀
    ; _≈⟨_⟩_ to _≈₀⟨_⟩_
    ; _≈˘⟨_⟩_ to _≈₀˘⟨_⟩_
    ; _≈≡⟨_⟩_ to _≈₀≡⟨_⟩_
    ; _≈⟨⟩_ to _≈₀⟨⟩_
    ; _≈≡˘⟨_⟩_ to _≈₀≡˘⟨_⟩_
    ; ≈-begin_ to ≈₀-begin_
    ; _≈-IsRelatedTo_ to _≈₀-IsRelatedTo_
    ; module _≈-IsRelatedTo_ to _≈₀-IsRelatedTo_
    )
module SetoidCalc₁ {i j : Level} (S : Setoid i j) where
  open Setoid₁ S public
  open SetoidCalc S public renaming
    ( _□ to _□₁
    ; _≈⟨_⟩_ to _≈₁⟨_⟩_
    ; _≈˘⟨_⟩_ to _≈₁˘⟨_⟩_
    ; _≈≡⟨_⟩_ to _≈₁≡⟨_⟩_
    ; _≈⟨⟩_ to _≈₁⟨⟩_
    ; _≈≡˘⟨_⟩_ to _≈₁≡˘⟨_⟩_
    ; ≈-begin_ to ≈₁-begin_
    ; _≈-IsRelatedTo_ to _≈₁-IsRelatedTo_
    ; module _≈-IsRelatedTo_ to _≈₁-IsRelatedTo_
    )
module SetoidCalc₂ {i j : Level} (S : Setoid i j) where
  open Setoid₂ S public
  open SetoidCalc S public renaming
    ( _□ to _□₂
```

```
      ; _≈⟨_⟩_ to _≈₂⟨_⟩_
      ; _≈˘⟨_⟩_ to _≈₂˘⟨_⟩_
      ; _≈≡⟨_⟩_ to _≈₂≡⟨_⟩_
      ; _≈⟨⟩_ to _≈₂⟨⟩_
      ; _≈≡˘⟨_⟩_ to _≈₂≡˘⟨_⟩_
      ; ≈-begin_ to ≈₂-begin_
      ; _≈-IsRelatedTo_ to _≈₂-IsRelatedTo_
      ; module _≈-IsRelatedTo_ to _≈₂-IsRelatedTo_
      )
module SetoidCalc₃ {i j : Level} (S : Setoid i j) where
   open Setoid₃ S public
   open SetoidCalc S public renaming
      (_□ to _□₃
      ; _≈⟨_⟩_ to _≈₃⟨_⟩_
      ; _≈˘⟨_⟩_ to _≈₃˘⟨_⟩_
      ; _≈≡⟨_⟩_ to _≈₃≡⟨_⟩_
      ; _≈⟨⟩_ to _≈₃⟨⟩_
      ; _≈≡˘⟨_⟩_ to _≈₃≡˘⟨_⟩_
      ; ≈-begin_ to ≈₃-begin_
      ; _≈-IsRelatedTo_ to _≈₃-IsRelatedTo_
      ; module _≈-IsRelatedTo_ to _≈₃-IsRelatedTo_
      )
module SetoidCalc₄ {i j : Level} (S : Setoid i j) where
   open Setoid₄ S public
   open SetoidCalc S public renaming
      (_□ to _□₄
      ; _≈⟨_⟩_ to _≈₄⟨_⟩_
      ; _≈˘⟨_⟩_ to _≈₄˘⟨_⟩_
      ; _≈≡⟨_⟩_ to _≈₄≡⟨_⟩_
      ; _≈⟨⟩_ to _≈₄⟨⟩_
      ; _≈≡˘⟨_⟩_ to _≈₄≡˘⟨_⟩_
      ; ≈-begin_ to ≈₄-begin_
      ; _≈-IsRelatedTo_ to _≈₄-IsRelatedTo_
      ; module _≈-IsRelatedTo_ to _≈₄-IsRelatedTo_
      )
```

2.4 RATH.PropositionalEquality

This module is our wrapper to the propositional equality of the standard library, and we use this throughout. By refraining from using of "_≡_" in any other way, and by prefixing most of the names here with "≡-", we thus avoid one source of name ambiguity, at lesser cost than using qualified names. The current module is therefore used in the RATH-Agda project as the sole interface to the standard library module Relation.Binary.PropositionalEquality. For the most part, we only re-export material from there:

```
open import Relation.Binary.PropositionalEquality public
   using
      (_≡_; _≢_
      ; _≗_              -- {a b : Level} {A : Set a} {B : Set b} (f g : A → B) → Set (a ⊍ b)
      ; _→-setoid_       -- {a b : Level} (A : Set a) (B : Set b) → Setoid (a ⊍ b) (a ⊍ b)
      ; :→-to-Π          -- {a b₁ b₂ : Level} {A : Set a} {B : I.Setoid A b₁ b₂}
                         -- → ((x : A) → I.Setoid.Carrier B x) → Π A B
      ; →-to-⟶          -- {a b₁ b₂ : Level} {A : Set a} {B : Setoid b₁ b₂}
                         -- → (A → Setoid.Carrier B) → A ⟶ B
      ; inspect; Reveal_is_
      ; Extensionality; extensionality-for-lower-levels; ∀-extensionality
      )
   renaming
```

```
  (refl to ≡-refl           -- : {a : Level} {A : Set a} {x : A} → x ≡ x
  ; sym to ≡-sym            -- : {a : Level} {A : Set a} {i j : A} → i ≡ j → j ≡ i
  ; trans to ≡-trans        -- : {a : Level} {A : Set a} {i j k : A} → i ≡ j → j ≡ k → i ≡ k
  ; subst to ≡-subst        -- : {a p : Level} {A : Set a} (P : A → Set p) {x y : A}
                            -- → x ≡ y → P x → P y
  ; subst₂ to ≡-subst₂      -- : {a b p : Level} {A : Set a} {B : Set b} (P : A → B → Set p)
                            -- → {x₁ x₂ : A} {y₁ y₂ : B} → x₁ ≡ x₂ → y₁ ≡ y₂ → P x₁ y₁ → P x₂ y₂
  ; cong to ≡-cong          -- : {a b : Level} {A : Set a} {B : Set b}
                            -- → (f : A → B) {x y : A}
                            -- → x ≡ y → f x ≡ f y
  ; cong₂ to ≡-cong₂        -- : {a b c : Level} {A : Set a} {B : Set b} {C : Set c}
                            -- → (f : A → B → C) {x y : A} {u v : B}
                            -- → x ≡ y → u ≡ v → f x u ≡ f y v
  ; resp₂ to ≡-resp₂        -- : {a ℓ : Level} {A : Set a} (_~_ : A → A → Set ℓ)
                            -- → ({x y z : A} → y ≡ z → x ~ y → x ~ z)
                            -- × ({x y z : A} → y ≡ z → y ~ x → z ~ x)
  ; proof-irrelevance to ≡-irrelevance   -- : {a : Level} {A : Set a} {x y : A} (p q : x ≡ y) → p ≡ q
  ; isEquivalence to ≡-isEquivalence
  ; setoid to ≡-setoid
  ; decSetoid to ≡-decSetoid
  ; isPreorder to ≡-isPreorder
  ; preorder to ≡-preorder
  ; [_] to ≡[_]≡
  )
```

For propositional equality _≡_, instead of the calculational reasoning interface provided by the standard library in Relation.Binary.PropositionalEquality.≡-Reasoning, we avoid having to duplicate our definition of _≈˘⟨_⟩_ for _≡˘⟨_⟩_ by instead providing a renamed version of the interface of Relation.Binary.EqReasoning.Extended that is also implicitly parameterised by the underlying set S (but we currently do not provide _≅⟨_⟩_ for heterogeneous equality):

```
module _ {ℓ : Level} {S : Set ℓ} where
  open import Relation.Binary.EqReasoning.Extended (≡-setoid S) public using () renaming
    (begin_     to ≡-begin_
    ; _≈⟨_⟩_    to _≡⟨_⟩_
    ; _≈⟨⟩_ to _≡⟨⟩_
    ; _≈˘⟨_⟩_ to _≡˘⟨_⟩_
    ; _■        to _≡■
    )
  infixr 1 _⟨≡≡⟩_ _⟨≡˘≡⟩_ _⟨≡≡⟩_ _⟨≡≡˘⟩_
  _⟨≡≡⟩_ : {i j k : S} → i ≡ j → j ≡ k → i ≡ k
  _⟨≡≡⟩_ = ≡-trans
  _⟨≡≡˘⟩_ : {i j k : S} → i ≡ j → k ≡ j → i ≡ k
  p ⟨≡≡˘⟩ q = p ⟨≡≡⟩ ≡-sym q
  _⟨≡˘≡⟩_ : {i j k : S} → j ≡ i → j ≡ k → i ≡ k
  p ⟨≡˘≡⟩ q = ≡-sym p ⟨≡≡⟩ q
  _⟨≡˘≡˘⟩_ : {i j k : S} → j ≡ i → k ≡ j → i ≡ k
  p ⟨≡˘≡˘⟩ q = ≡-sym p ⟨≡≡⟩ ≡-sym q
```

We add names for the properties of pointwise function equality:

```
module _ {ℓa ℓb : Level} {A : Set ℓa} {B : Set ℓb} where
  open Setoid (A →-setoid B) public using () renaming
    (refl to ≗-refl; reflexive to ≗-reflexive; sym to ≗-sym; trans to ≗-trans
    ; isEquivalence to ≗-isEquivalence)
```

We also add a datatype for isomorphisms up to propositional equality:

```
record ≡-Iso {ℓa ℓb : Level} (A : Set ℓa) (B : Set ℓb) : Set (ℓa ⊍ ℓb) where
  field
    fun : A → B
    inv : B → A
    left : fun ∘ inv ≗ id
    right : inv ∘ fun ≗ id
```

2.5 Relation.Binary.PropositionalEquality.Utils

≡-subst Simplification

This, and many of the propositional equalities below, are intended for type adaptations using ≡-subst from RATH.PropositionalEquality (Sect. 2.4); for reference:

```
≡-subst : {a b : Level} {A : Set a} (P : A → Set b) {x y : A} → x ≡ y → P x → P y
≡-subst ≡-refl p = p
```

Where the type adaptation has come full circle, subst becomes superfluous:

```
≡-subst-contract : {s ℓ : Level} {S : Set s} (P : S → Set ℓ) {x : S}
  → (x≡x : x ≡ x) → (p : P x) → ≡-subst P x≡x p ≡ p
≡-subst-contract P ≡-refl p = ≡-refl
```

Certain nested applications of subst can be turned into a single composed application:

```
≡-subst-comp : {a b c : Level} {A : Set a} {B : Set b}
             → (Q : A → B) (P : B → Set c) {u v : A} {x : B}
             → (u≡v : u ≡ v) → (x≡Qu : x ≡ Q u)
             → (p : P x)
             → ≡-subst (λ u → P (Q u)) {u} {v} u≡v
                ( ≡-subst P {x} {Q u} x≡Qu p)
                ≡ ≡-subst P {x} {Q v} (≡-trans x≡Qu (≡-cong Q u≡v)) p
≡-subst-comp Q P ≡-refl ≡-refl p = ≡-refl
```

A simple special case:

```
≡-subst-cancel : {a b : Level} {A : Set a} (P : A → Set b) {x y : A} {p : P x}
               → (x≡y : x ≡ y) (y≡x : y ≡ x) → ≡-subst P y≡x (≡-subst P x≡y p) ≡ p
≡-subst-cancel P ≡-refl ≡-refl = ≡-refl
```

The following were motivated by Data.Fin.Utils.FinIso-≡; only subst-≡-sym is used by the first finished version:

```
subst-≡-sym : {ℓa : Level} {A : Set ℓa} {x y : A} (eq : x ≡ y)
            → ≡-subst (_≡_ x) (≡-sym eq) eq ≡ ≡-refl
subst-≡-sym ≡-refl = ≡-refl

subst-≡-cong-sym : {ℓa ℓb : Level} {A : Set ℓa} {B : Set ℓb} {F : A → B} {x y : A} (eq : x ≡ y)
                 → ≡-refl ≡ ≡-subst (λ z → F x ≡ F z) (≡-sym eq) (≡-cong F eq)
subst-≡-cong-sym ≡-refl = ≡-refl

subst-≡-sym-cong : {ℓa ℓb : Level} {A : Set ℓa} {B : Set ℓb} {F : A → B} {x y : A} {z : B}
                 → (eq₁ : x ≡ y) (eq₂ : F y ≡ z)
                 → ≡-cong F eq₁ ≡ ≡-subst (λ z → F x ≡ z) (≡-sym eq₂) (≡-trans (≡-cong F eq₁) eq₂)
subst-≡-sym-cong ≡-refl ≡-refl = ≡-refl
```

More Complex Substitutions

```
≡-subst₂D : {a b p : Level} {A : Set a} {B : A → Set b} (P : (x : A) → B x → Set p)
            → {x₁ x₂ : A} {y₁ : B x₁} {y₂ : B x₂}
            → (x₁≡x₂ : x₁ ≡ x₂) → y₁ ≡ ≡-subst B (≡-sym x₁≡x₂) y₂ → P x₁ y₁ → P x₂ y₂
≡-subst₂D P ≡-refl ≡-refl p = p
```

```
≡-cong-Σ : {a b : Level} {A : Set a} {B : A → Set b}
           → {x₁ x₂ : A} {y₁ : B x₁} {y₂ : B x₂}
           → (x₁≡x₂ : x₁ ≡ x₂) → y₁ ≡ ≡-subst B (≡-sym x₁≡x₂) y₂ → (x₁, y₁) ≡ (x₂, y₂)
≡-cong-Σ ≡-refl ≡-refl = ≡-refl
```

```
≡-cong₂D : {a b c : Level} {A : Set a} {B : A → Set b} {C : Set c} (f : (x : A) → B x → C)
           → {x₁ x₂ : A} {y₁ : B x₁} {y₂ : B x₂}
           → (x₁≡x₂ : x₁ ≡ x₂) → y₁ ≡ ≡-subst B (≡-sym x₁≡x₂) y₂ → f x₁ y₁ ≡ f x₂ y₂
≡-cong₂D f ≡-refl ≡-refl = ≡-refl
```

2.6 Relation.Decidable.Utils

Thu utility functions collected in this modules are motivated by their use with decidable propositional equality in Sect. 23.32.

Since the standard library (as of version 0.5) does not provide an eliminator for Dec, we provide our own:

```
withDec : {p : Level} {P : Set p} → Dec P
          → {r : Level} {R : Set r} → (P → R) → (¬ P → R) → R
withDec (yes p)  T E = T p
withDec (no ¬p) T E = E ¬p
```

For the special case of decidable propositional equality, the type Dec (x ≡ x) only contains the element yes refl, but the Agda typechecker (as of version 2.2.10) does not use this fact automatically, so we explicitly provide the resulting equality:

```
withDec-contract : {ℓ : Level} {S : Set ℓ} {x : S} {d : Dec (x ≡ x)}
  → {r : Level} {R : Set r} {T : x ≡ x → R} {E : ¬ (x ≡ x) → R}
  → withDec d T E ≡ T refl
withDec-contract {d = yes refl}   = refl
withDec-contract {d = no ¬x≡x} = ⊥-elim (¬x≡x refl)
```

A dependent variant of withDec:

```
withDec-subst : {ℓ : Level} {S : Set ℓ} {x y : S}
  → (d : Dec (x ≡ y))
  → {r p : Level} {R : Set r} {P : R → Set p}
  → (T : x ≡ y → R) → (E : ¬ (x ≡ y) → R)
  → (t : (x≡y : x ≡ y) → P (T x≡y))
  → (e : (¬x≡y : ¬ (x ≡ y)) → P (E ¬x≡y))
  → P (withDec d T E)
withDec-subst (yes x≡y) T E t e = t x≡y
withDec-subst (no ¬x≡y) T E t e = e ¬x≡y
```

Again the case where d : Dec (x ≡ x) has to be yes refl:

```
withDec-subst-contract : {ℓ : Level} {S : Set ℓ} {x : S}
  → {d : Dec (x ≡ x)}
```

```
  → {r p : Level} {R : Set r} {P : R → Set p}
  → {T : x ≡ x → R} → {E : ¬ (x ≡ x) → R}
  → {t : (x≡x : x ≡ x) → P (T x≡x)}
  → {e : (¬x≡x : ¬ (x ≡ x)) → P (E ¬x≡x)}
  → withDec-subst d {P = P} T E t e
    ≡ subst P (sym (withDec-contract {d = d} {T = T} {E = E})) (t refl)
withDec-subst-contract {d = yes refl}  = refl
withDec-subst-contract {d = no ¬x≡x} = ⊥-elim (¬x≡x refl)
```

The fact that d : Dec (x ≡ x) has to be yes refl as an equality:

```
decide-x≡x : {ℓ : Level} {S : Set ℓ} {x : S} {d : Dec (x ≡ x)} → d ≡ yes refl
decide-x≡x {d = yes refl} = refl
decide-x≡x {d = no ¬x≡x} = ⊥-elim (¬x≡x refl)
```

As a consequence, any two elements of Dec (x ≡ x) are equal:

```
Dec-x≡x-irrelevance : {ℓ : Level} {S : Set ℓ} {x : S} {d e : Dec (x ≡ x)} → d ≡ e
Dec-x≡x-irrelevance {d = yes refl} {e = yes refl} = refl
Dec-x≡x-irrelevance {d = no ¬x≡x} = ⊥-elim (¬x≡x refl)
Dec-x≡x-irrelevance {e = no ¬x≡x} = ⊥-elim (¬x≡x refl)
```

2.7 Data.Empty.Generalised

```
data ⊥ {k : Level} : Set k where


⊥-elim : {k w : Level} {Whatever : Set w} → ⊥ {k} → Whatever
⊥-elim ()
```

2.8 Data.Empty.Setoid

To make this fully Level-polymorphic, we use our Level-polymorphic empty type from Sect. 2.7.

```
import Data.Empty.Generalised as E using (⊥; ⊥-elim)
⊥ : {k ℓ : Level} → Setoid k ℓ
⊥ = record
  {Carrier = E.⊥
  ; _≈_ = E.⊥-elim
  ; isEquivalence = record
    {refl  = λ {x} → E.⊥-elim x
    ; sym   = λ {x} → E.⊥-elim x
    ; trans = λ {x} → E.⊥-elim x
    }
  }
```

For every setoid A, there is exactly one setoid homomorphism from ⊥ to A:

```
⊥-elim : {k ℓ ℓ₁ ℓ₂ : Level} {A : Setoid ℓ₁ ℓ₂} → ⊥ {k} {ℓ} ⟶ A
⊥-elim = record
  {_⟨$⟩_ = E.⊥-elim
  ; cong   = λ {i} → E.⊥-elim i
  }
```

2.9 Data.Unit.Generalised

As in Data.Unit, this is \top:

```
record ⊤ {k : Level} : Set k where
  constructor tt
```

2.10 Data.Unit.Setoid

To make this fully Level-polymorphic, we use our Level-polymorphic unit type from Sect. 2.9.

```
open import Data.Unit.Generalised as U using (tt)
⊤ : {k ℓ : Level} → Setoid k ℓ
⊤ = record
  {Carrier = U.⊤
  ; _≈_ = λ _ _ → U.⊤
  ; isEquivalence = record
    {refl   = λ {_} → tt
    ; sym   = λ {_} {_} p → p
    ; trans = λ {_} {_} {_} p q → tt
    }
  }
```

From any setoid A, there is exactly one setoid homomorphism to ⊤:

```
! : {a k ℓ₁ ℓ₂ : Level} {A : Setoid a ℓ₁} → A ⟶ ⊤ {k} {ℓ₂}
! = record
  {_⟨$⟩_ = λ _ → tt
  ; cong = λ {i} {j} p → tt
  }
```

2.11 RATH.Data.Product

This is a wrapper around Data.Product of the standard library, since in late November 2012, the "fake colon" in the syntax definition for Σ there has been replaced with ∈, and we want to keep that symbol available for element relations as in Categoric.Membership.

Therefore this module re-exports everything from the standard library's Data.Product, except for the Σ-syntax alias to which the syntax definition using ∈ is now bound, and instead adds its own alias to which the old syntax definition is attached.

```
open import Data.Product public hiding (Σ-syntax)
Σ: : {a b : Level} (A : Set a) (B : A → Set b) → Set (a ⊍ b)
Σ: = Data.Product.Σ
syntax Σ: A (λ x → B) = Σ [x : A] B
```

We also introduce a variant that uses the "bullet" separator of the Z notation (Spivey, 1989); we may decide to fully switch to that at some point in the future.

```
Σ:• : {a b : Level} (A : Set a) (B : A → Set b) → Set (a ⊍ b)
Σ:• = Data.Product.Σ
syntax Σ:• A (λ x → B) = Σ x : A • B
```

For convenience, we add one-sided versions of Product.map:

```
map₁ :  {ℓa₁ ℓa₂ ℓb : Level} {A₁ : Set ℓa₁} {A₂ : Set ℓa₂} {B : A₂ → Set ℓb}
     →  (f : A₁ → A₂) → Σ [ a₁ : A₁ ] B (f a₁) → Σ A₂ B
map₁ f (x, y) = f x, y
map₂ :  {ℓa ℓb₁ ℓb₂ : Level} {A : Set ℓa} {B₁ : A → Set ℓb₁} {B₂ : A → Set ℓb₂}
     →  ({a : A} → B₁ a → B₂ a) → Σ A B₁ → Σ A B₂
map₂ g (x, y) = x, g y
map₁₁ : {ℓa₁ ℓa₂ ℓb ℓc : Level} {A₁ : Set ℓa₁} {A₂ : Set ℓa₂} {B : A₂ → Set ℓb} {C : Σ A₂ B → Set ℓc}
     →  (f : A₁ → A₂) → Σ (Σ [ a₁ : A₁ ] B (f a₁)) (λ {(a, b) → C (f a, b)}) → Σ (Σ A₂ B) C
map₁₁ f ((x, y), z) = (f x, y), z
map₁₂ : {ℓa ℓb₁ ℓb₂ ℓc : Level}
        {A : Set ℓa} {B₁ : A → Set ℓb₁} {B₂ : A → Set ℓb₂} {C : Σ A B₂ → Set ℓc}
     →  (g : {a : A} → B₁ a → B₂ a) → Σ (Σ A B₁) (λ {(a, b) → C (a, g b)}) → Σ (Σ A B₂) C
map₁₂ g ((x, y), z) = (x, g y), z
map₂₁ : {ℓa ℓb₁ ℓb₂ ℓc : Level}
        {A : Set ℓa} {B₁ : A → Set ℓb₁} {B₂ : A → Set ℓb₂} {C : {a : A} → B₂ a → Set ℓc}
     →  (g : {a : A} → B₁ a → B₂ a) → Σ [ a : A ] Σ [ b : B₁ a ] C (g b) → Σ [ a : A ] Σ (B₂ a) C
map₂₁ g (x, y, z) = x, g y, z
map₂₂ : {ℓa ℓb ℓc₁ ℓc₂ : Level}
        {A : Set ℓa} {B : A → Set ℓb} {C₁ : {a : A} → B a → Set ℓc₁} {C₂ : {a : A} → B a → Set ℓc₂}
     →  ({a : A} {b : B a} → C₁ b → C₂ b) → Σ [ a : A ] Σ (B a) C₁ → Σ [ a : A ] Σ (B a) C₂
map₂₂ h (x, y, z) = x, y, h z
```

We also add nested projections:

```
proj₁₁ :  {ℓa ℓb ℓc : Level} {A : Set ℓa} {B : A → Set ℓb} {C : Σ A B → Set ℓc}
       → Σ (Σ A B) C → A
proj₁₁ ((x, y), z) = x
proj₁₂ :  {ℓa ℓb ℓc : Level} {A : Set ℓa} {B : A → Set ℓb} {C : Σ A B → Set ℓc}
       → (t : Σ (Σ A B) C) → B (proj₁₁ t)
proj₁₂ ((x, y), z) = y
proj₂₁ :  {ℓa ℓb ℓc : Level} {A : Set ℓa} {B : A → Set ℓb} {C : {a : A} → B a → Set ℓc}
       → (t : Σ [ a : A ] Σ (B a) C) → B (proj₁ t)
proj₂₁ (x, y, z) = y
proj₂₂ :  {ℓa ℓb ℓc : Level} {A : Set ℓa} {B : A → Set ℓb} {C : {a : A} → B a → Set ℓc}
       → (t : Σ [ a : A ] Σ (B a) C) → C {proj₁ t} (proj₂₁ t)
proj₂₂ (x, y, z) = z
```

2.12 Function.Composition

We will occasionally need composition after two-argument functions, with:

$$(f \circ_2 g)\, x\, y \;=\; f\, (g\, x\, y)$$

```
_∘₂_ : {a b c d : Level}
       {A : Set a}
       {B : A → Set b}
       {C : (x : A) → B x → Set c}
       {D : (x : A) → (y : B x) → C x y → Set d} →
  ({x : A} {y : B x} (z : C x y) → D x y z) → (g : (x : A) → (y : B x) → C x y) →
  ((x : A) → (y : B x) → D x y (g x y))
f ∘₂ g = λ x y → f (g x y)
```

2.13 Relation.Binary.Conversions

posetSetoid extracts the underlying Setoid of a Poset:

```
posetSetoid : {c ℓ₁ ℓ₂ : Level} → Poset c ℓ₁ ℓ₂ → Setoid c ℓ₁
posetSetoid P = record
  {Carrier = Poset.Carrier P
  ; _≈_ = Poset._≈_ P
  ; isEquivalence = IsPreorder.isEquivalence (Preorder.isPreorder (Poset.preorder P))
  }
```

Given a relation _≈_ and a proof that it is an equivalence relation as an IsEquivalence record, trivPreorder produces the IsPreorder record that proves that this equivalence relation is a preorder over itself as underlying equality:

```
trivPreorder : {c ℓ : Level} {A : Set c} {_≈_ : Rel A ℓ}
             → IsEquivalence {c} {ℓ} _≈_ → IsPreorder {c} {ℓ} {ℓ} _≈_ _≈_
trivPreorder {_≈_} isEq = let open IsEquivalence isEq in record
  {isEquivalence = isEq
  ; reflexive     = id
  ; trans         = trans
  }
```

We use this to show how a setoid can be considered as a discrete preorder, that is, as a preorder that coincides with its underlying equality:

```
setoidPreorder : {c ℓ : Level} → Setoid c ℓ → Preorder c ℓ ℓ
setoidPreorder S = let open Setoid S in record
  {Carrier    = Carrier
  ; _≈_       = _≈_
  ; _∼_       = _≈_
  ; isPreorder = trivPreorder isEquivalence
  }
setoidPoset : {c ℓ : Level} → Setoid c ℓ → Poset c ℓ ℓ
setoidPoset S = let open Setoid S in record
  {Carrier = Carrier; _≈_ = _≈_; _≤_ = _≈_
  ; isPartialOrder = record {isPreorder = trivPreorder isEquivalence; antisym = λ x≤y _ → x≤y}
  }
```

Note that the standard library provides Setoid.preorder, which contains a IsPreorder _≡_ _≈_, and therefore produces a preorder with propositional equality as underlying equivalence.

Part I

Semigroupoids and Categories

Chapter 3

Semigroupoids and Categories: Composition, Identity, Converse

In this chapter, we include the theory of semigroupoids as our starting point, and separate theories adding identities, which yields categories, and adding converse to both semigroupoids and categories.

We start with two "demo modules": Monolithic definitions of semigroupoids (Sect. 3.1) and categories (Sect. 3.2). Due to their monolithic nature, these are used for explaining the general approach of the RATH-Agda formalisations in publications where there is insufficient space to present the whole fine-grained development starting in Sect. 3.4 that spreads the definition of categories over six top-level modules.

The interface module Categoric (Sect. 3.3) re-exports all the foundational relation-algebraic theories from semigroupoids to (currently) distributive action allegories $\boxed{\textbf{???}}$, which all reside directly below Categoric in the module hierarchy.

3.1 Categoric.Semigroupoid.Monolithic

A semigroupoid can be considered as a "category without identities", namely consisting of

- a collection Obj of objects,
- for any two objects A and B, a collection Mor A B of "morphisms from A to B",
- and a composition operation _⨾_ that takes two "consecutive" mrophisms $A \xrightarrow{f} B \xrightarrow{g} C$ to a composed morphism f ⨾ g from A to C, where this composition is associative.

We allow Obj to be an arbitrary Set, but we use Setoids for the morphism collections Hom A B, and therefore need to add the constraint ⨾-cong that the composition respects the setoid equivalences.

```
import Categoric.Semigroupoid as Semigroupoid₀

record Semigroupoid′ {ℓi ℓj ℓk : Level} {Obj : Set ℓi} (Hom : Obj → Obj → Setoid ℓj ℓk)
                     : Set (ℓi ⊍ ℓj ⊍ ℓk) where
  Mor : Obj → Obj → Set ℓj
  Mor = λ A B → Setoid.Carrier (Hom A B)
  infix 4 _≈_; infixr 9 _⨾_
  _≈_ = λ {A} {B} → Setoid._≈_ (Hom A B)
  field _⨾_     : {A B C : Obj} → Mor A B → Mor B C → Mor A C
        ⨾-cong  : {A B C : Obj} {f₁ f₂ : Mor A B} {g₁ g₂ : Mor B C}
                → f₁ ≈ f₂ → g₁ ≈ g₂ → (f₁ ⨾ g₁) ≈ (f₂ ⨾ g₂)
        ⨾-assoc : {A B C D : Obj} {f : Mor A B} {g : Mor B C} {h : Mor C D}
                → ((f ⨾ g) ⨾ h) ≈ (f ⨾ (g ⨾ h))
    -- Up to here, monolithic presentation of semigroupoids.
    -- From here, compatibility with Categoric.Semigroupoid:
```

```
  compOp : CompOp Hom
  compOp = record {_⨾_ = _⨾_; ⨾-cong = ⨾-cong; ⨾-assoc = ⨾-assoc}
  semigroupoid : Semigroupoid₀.Semigroupoid ℓj ℓk Obj
  semigroupoid = record {Hom = Hom; compOp = compOp}
  open Semigroupoid₀.Semigroupoid semigroupoid public hiding
    (Hom; Mor; _≈_; _⨾_; ⨾-cong; ⨾-assoc; compOp)
```

Taking the above Semigroupoid′ as starting point, a category in addition needs "identity morphisms", i.e., an operation Id that associates with each object A a morphism Id {A} from A to A, satisfying left- and right-identity laws for composition:

```
record Category′ {ℓi ℓj ℓk : Level} {Obj : Set ℓi} (Hom : Obj → Obj → Setoid ℓj ℓk)
                        : Set (ℓi ⊍ ℓj ⊍ ℓk) where
  field semigroupoid : Semigroupoid′ Hom
  open Semigroupoid′ semigroupoid hiding (semigroupoid)
  field Id        : {A : Obj} → Mor A A
        leftId    : {A : Obj} → isLeftIdentity   (Id {A})
        rightId   : {A : Obj} → isRightIdentity (Id {A})
```

3.2 Categoric.Category.Monolithic

```
import Categoric.Category as Category₀
```

Similar to Semigroupoid′ in Categoric.Semigroupoid.Monolithic (Sect. 3.1), we show here a monolithic characterisation of categories.

Strictly speaking, if $\mathcal{C}$ is of type Category′ {Obj = Obj} Hom, then this means that "$\mathcal{C}$ is a category with elements of Obj as objects and elements of Hom as morphisms", since Obj and Hom are parameters of the record type Category′ instead of **field**s.

The main RATH-Agda development starting with Categoric.LESGraph will use a different parameterisation, namely turn Hom into a **field**. (The only context we have seen so far where turning also Obj into a **field** would be advantagous is for defining a category of categories, which for the time being is not yet needed by the envisaged applications of RATH-Agda.)

The type chosen here and in Categoric.Semigroupoid.Monolithic (Sect. 3.1) is more flexible in certain contexts, but we still don't see sufficient reasons to abandon the choice of having exactly Obj as parameter for the main RATH-Agda development.

```
record Category′ {i j k : Level} {Obj : Set i} (Hom : Obj → Obj → Setoid j k) : Set (i ⊍ j ⊍ k) where
  Mor : Obj → Obj → Set j
  Mor = λ A B → Setoid.Carrier (Hom A B)
  infix 4 _≈_; infixr 9 _⨾_
  _≈_ = λ {A} {B} → Setoid._≈_ (Hom A B)
  field _⨾_     : {A B C : Obj} → Mor A B → Mor B C → Mor A C
        ⨾-cong  : {A B C : Obj} {f₁ f₂ : Mor A B} {g₁ g₂ : Mor B C}
                → f₁ ≈ f₂ → g₁ ≈ g₂ → (f₁ ⨾ g₁) ≈ (f₂ ⨾ g₂)
        ⨾-assoc : {A B C D : Obj} {f : Mor A B} {g : Mor B C} {h : Mor C D}
                → ((f ⨾ g) ⨾ h) ≈ (f ⨾ (g ⨾ h))
        Id      : {A : Obj} → Mor A A
        leftId  : {A B : Obj} → {f : Mor A B} → (Id ⨾ f) ≈ f
        rightId : {A B : Obj} → {f : Mor A B} → (f ⨾ Id) ≈ f
    -- Up to here, monolithic presentation of categories.
    -- From here, compatibility with Categoric.Category:
  compOp : CompOp Hom
  compOp = record {_⨾_ = _⨾_; ⨾-cong = ⨾-cong; ⨾-assoc = ⨾-assoc}
```

```
  semigroupoid : Semigroupoid j k Obj
  semigroupoid = record {Hom = Hom; compOp = compOp}
  idOp : SGIdOp semigroupoid
  idOp = record {Id = Id; leftId = leftId; rightId = rightId}
  category : Category₀.Category j k Obj
  category = record {semigroupoid = semigroupoid; idOp = idOp}

  open Category₀.Category category public hiding
    (Hom; Mor; _≈_; _⨾_; ⨾-cong; ⨾-assoc
    ; Id; leftId; rightId
    ; compOp; semigroupoid; idOp
    )
```

3.3 Categoric

Re-export only:

```
open import Categoric.LESGraph                        public -- Sect. 3.4
open import Categoric.CompOp                          public -- Sect. 3.5
open import Categoric.CompOpProps1                    public -- Sect. 3.6
open import Categoric.LESGraph.Examples               public -- Sect. 3.7
open import Categoric.Semigroupoid                    public -- Sect. 3.8
open import Categoric.Semigroupoid.SGIso              public -- Sect. 3.11
open import Categoric.Semigroupoid.Factoring          public -- Sect. 3.12
open import Categoric.IdOp                            public -- Sect. 3.13
open import Categoric.Category                        public -- Sect. 3.14
open import Categoric.ConvSemigroupoid                public -- Sect. 3.15
open import Categoric.ConvCategory                    public -- Sect. 3.16
open import Categoric.Diagram                         public -- Sect. 3.17

import Categoric.FinColimits                                 -- Sect. 4.7
import Categoric.FinLimits                                   -- Sect. 4.8
import Categoric.FinColimits.Pushout-Coproduct               -- Sect. 4.6
import Categoric.Semigroupoid.FinColimits                    -- Sect. 4.9
import Categoric.Semigroupoid.FinLimits                      -- Sect. 4.10
import Categoric.Category.FinColimits                        -- Sect. 4.11
import Categoric.Category.FinLimits                          -- Sect. 4.12

open import Categoric.OrderedSemigroupoid             public -- Sect. 9.1
open import Categoric.OrderedCategory                 public -- Sect. 9.2
open import Categoric.OrderedSemigroupoid.Lattice     public -- Sect. 9.3
open import Categoric.LSLSemigroupoid                 public -- Sect. 9.4
open import Categoric.USLSemigroupoid                 public -- Sect. 9.5
open import Categoric.USLCategory                     public -- Sect. 9.6
open import Categoric.LatticeSemigroupoid             public -- Sect. 9.7
open import Categoric.DistrLatSemigroupoid            public -- Sect. 9.8
open import Categoric.ZeroMor                         public -- Sect. 9.9

open import Categoric.DomainSemigroupoid              public -- Sect. 10.1
open import Categoric.OSGD                            public -- Sect. 10.2
open import Categoric.OCD                             public -- Sect. 10.3

open import Categoric.OSGC                            public -- Sect. 11.2
open import Categoric.OCC                             public -- Sect. 11.10
open import Categoric.MapSG                           public -- Sect. 11.17
open import Categoric.MapCat                          public -- Sect. 11.18

open import Categoric.SemiAllegory                    public -- Sect. 12.2
open import Categoric.Allegory                        public -- Sect. 12.4
open import Categoric.USLSGC                          public -- Sect. 12.5
open import Categoric.USLCC                           public -- Sect. 12.6
open import Categoric.DistrLatSGC                     public -- Sect. 12.8
```

```
open import Categoric.DistrLatCC                       public  -- Sect. 12.9
open import Categoric.OSGC.LeastMor                    public  -- Sect. 12.1

open import Categoric.SemiCollagory                    public  -- Sect. 12.10
open import Categoric.Collagory                        public  -- Sect. 12.11
open import Categoric.DistrSemiAllegory                public  -- Sect. 12.12
open import Categoric.DistrAllegory                    public  -- Sect. 12.13

open import Categoric.OrderedSemigroupoid.Residuals    public  -- Sect. 13.1
open import Categoric.OrderedCategory.Residuals        public  -- Sect. 13.2
open import Categoric.OSGC.Residuals                   public  -- Sect. 13.3
open import Categoric.OSGD.RestrictedResiduals         public  -- Sect. 13.4
open import Categoric.OSGD.Residuals                   public  -- Sect. 13.5
open import Categoric.OSGC.SyQ                         public  -- Sect. 13.6
open import Categoric.SemiAllegory.Residuals           public  -- Sect. 13.7
open import Categoric.DivSemiAllegory                  public  -- Sect. 13.8
open import Categoric.DivAllegory                      public  -- Sect. 13.9

open import Categoric.KleeneSemigroupoid               public  -- Sect. 14.1
open import Categoric.KSGC                             public  -- Sect. 14.2
open import Categoric.KleeneCategory                   public  -- Sect. 14.3
open import Categoric.KCC                              public  -- Sect. 14.4
open import Categoric.KleeneCollagory                  public  -- Sect. 14.5
open import Categoric.ActLatSemigroupoid               public  -- Sect. 14.6
open import Categoric.ActLatCategory                   public  -- Sect. 14.7
open import Categoric.DistrActAllegory                 public  -- Sect. 14.8

open import Categoric.DirectSum                        public  -- Sect. 15.1
open import Categoric.KleeneCategory.DirectSum         public  -- Sect. 15.2
```

3.4 Categoric.LESGraph

A natural way to think about the underlying graph of a category of semigroupoid is to consider not a global collection of edges, but local collections. That is, a graph has, for any pair (x, y) of vertices, such a local collection of edges from x to y.

For different purposes, different kinds of collection are appropriate:

- A semigroupoid is a graph where the vertices are called objects and the edges are called morphisms, and equality reasoning on morphisms is available, so assigning a Setoid of morphisms (or edges) to any two objects (or vertices) is natural.
- In a locally ordered semigroupoid or category, a Poset of morphisms is available, which can be viewed as a Setoid enriched with an additional ordering relation compatible with the Setoid structure.
- In a 2-category, the collection of morphisms from x to y provides the objects of a local category, with two-cells as morphisms.

Since graphs are the most general structures of this kind, they should have at most Setoids of edges for pairs of nodes; we currently do not see any reason to only consider Sets of edges.

Therefore, we introduce “LESGraph”s as graphs based on “local edge setoids”.

```
LocalSetoid : {i : Level} (Node : Set i) (j k : Level) → Set (i ⊍ ℓsuc (j ⊍ k))
LocalSetoid {i} Node j k = Node → Node → Setoid j k
```

```
module LocalEdgeSetoid {i j k : Level} {Node : Set i} (LES : LocalSetoid Node j k) where
  Edge : Node → Node → Set j
  Edge = Setoid.Carrier ∘₂ LES
```

We provide variants of the Setoid components taking two objects as additional implicit arguments, including in particular

```
_≈_ : {A B : Obj} → Edge A B → Edge A B → Set k.
```

We choose to achieve this via a parameterised anonymous module (also know as "section"):

```
module _ {A B : Node} where
  open Setoid′ (LES A B) public hiding (Carrier)
```

Note that the hidden item has the type Carrier : {A : Node} → {B : Node} → Set j with *implicit* arguments, while we want to use Edge as defined above with *explicit* arguments.

```
module LocalEdgeSetoidAux where
```

The source and target mappings are included here for the sake of completeness; they are rarely useful in our dependently-typed setting since their results are typically already available as (implicit) arguments.

```
source : {A B : Node} → Edge A B → Node
source {A} {_} _ = A
target : {A B : Node} → Edge A B → Node
target {_} {B} _ = B
```

Occasionally we need to adapt source and/or target via propositional equality; the following properties help reasoning in such cases:

```
≡-substSrc :  {A A′ B : Node} (A≡A′ : A ≡ A′) (F : Edge A B) → Edge A′ B
≡-substSrc {B = B} = ≡-subst (flip Edge B)
≡-substTrg :  {A B B′ : Node} (B≡B′ : B ≡ B′) (F : Edge A B) → Edge A B′
≡-substTrg {A} = ≡-subst (Edge A)
≡˘-substSrc :  {A A′ B : Node} (A′≡A : A′ ≡ A) (F : Edge A B) → Edge A′ B
≡˘-substSrc A′≡A  = ≡-substSrc (≡-sym A′≡A)
≡˘-substTrg :  {A B B′ : Node} (B′≡B : B′ ≡ B) (F : Edge A B) → Edge A B′
≡˘-substTrg B′≡B = ≡-substTrg (≡-sym B′≡B)
≡-substSrc-contract :  {A B : Node} (A≡A : A ≡ A) {F : Edge A B} → ≡-substSrc A≡A F ≡ F
≡-substSrc-contract {B = B} ≡-refl {F} = ≡-subst-contract (flip Edge B) ≡-refl F
≡-substTrg-contract :  {A B : Node} (B≡B : B ≡ B) {F : Edge A B} → ≡-substTrg B≡B F ≡ F
≡-substTrg-contract {A} ≡-refl {F} = ≡-subst-contract (Edge A) ≡-refl F
≡-substSrc-cong : {A A′ B : Node} (A≡A′ : A ≡ A′) {F G : Edge A B} → F ≈ G
                  → ≡-substSrc A≡A′ F ≈ ≡-substSrc A≡A′ G
≡-substSrc-cong ≡-refl F≈G = F≈G
≡-substTrg-cong : {A B B′ : Node} (B≡B′ : B ≡ B′) {F G : Edge A B} → F ≈ G
                  → ≡-substTrg B≡B′ F ≈ ≡-substTrg B≡B′ G
≡-substTrg-cong ≡-refl F≈G = F≈G
≡˘-substSrc-cong  : {A A′ B : Node} (A′≡A : A′ ≡ A) {F G : Edge A B} → F ≈ G
                  → ≡˘-substSrc A′≡A F ≈ ≡˘-substSrc A′≡A G
≡˘-substSrc-cong ≡-refl F≈G = F≈G
≡˘-substTrg-cong  : {A B B′ : Node} (B′≡B : B′ ≡ B) {F G : Edge A B} → F ≈ G
                  → ≡˘-substTrg B′≡B F ≈ ≡˘-substTrg B′≡B G
≡˘-substTrg-cong ≡-refl F≈G = F≈G
≡-substSrc-irr   : {A A′ B : Node} (p q : A ≡ A′) {F : Edge A B}
                  → ≡-substSrc p F ≡ ≡-substSrc q F
≡-substSrc-irr ≡-refl ≡-refl = ≡-refl
≡-substTrg-irr   : {A B B′ : Node} (p q : B ≡ B′) {F : Edge A B}
                  → ≡-substTrg p F ≡ ≡-substTrg q F
≡-substTrg-irr ≡-refl ≡-refl = ≡-refl
≡-substSrcSrc :   {A A′ A″ B : Node} (A≡A′ : A ≡ A′) (A′≡A″ : A′ ≡ A″) {F : Edge A B}
              →   ≡-substSrc A′≡A″ (≡-substSrc A≡A′ F)
              ≡   ≡-substSrc (A≡A′ ⟨≡≡⟩ A′≡A″) F
```

```
  ≡-substSrcSrc ≡-refl ≡-refl  =  ≡-refl
  ≡-substSrcSrc-irr  :  {A A′ A″ B : Node} (A≡A′ : A ≡ A′) (A′≡A″ : A′ ≡ A″) (A≡A″ : A ≡ A″)
                      → {F : Edge A B} → ≡-substSrc A′≡A″ (≡-substSrc A≡A′ F)
                                        ≡  ≡-substSrc A≡A″  F
  ≡-substSrcSrc-irr ≡-refl ≡-refl ≡-refl  =  ≡-refl
  ≡-substTrgTrg :   {A B B′ B″ : Node} (B≡B′ : B ≡ B′) (B′≡B″ : B′ ≡ B″) {F : Edge A B}
                 →   ≡-substTrg B′≡B″ (≡-substTrg B≡B′ F)
                 ≡   ≡-substTrg (B≡B′ ⟨≡≡⟩ B′≡B″) F
  ≡-substTrgTrg ≡-refl ≡-refl  =  ≡-refl
  ≡-substTrgTrg-irr  :  {A B B′ B″ : Node} (B≡B′ : B ≡ B′) (B′≡B″ : B′ ≡ B″) (B≡B″ : B ≡ B″)
                      → {F : Edge A B} → ≡-substTrg B′≡B″ (≡-substTrg B≡B′ F)
                                        ≡  ≡-substTrg B≡B″  F
  ≡-substTrgTrg-irr ≡-refl ≡-refl ≡-refl  =  ≡-refl
  ≡-substSrcSrc-contract  :  {A A′ B : Node} (A≡A′ : A ≡ A′) (A′≡A : A′ ≡ A) {F : Edge A B}
                            → ≡-substSrc A′≡A (≡-substSrc A≡A′ F)  ≡ F
  ≡-substSrcSrc-contract ≡-refl ≡-refl  =  ≡-refl
  ≡-substTrgTrg-contract :  {A B B′ : Node} (B≡B′ : B ≡ B′) (B′≡B : B′ ≡ B) {F : Edge A B}
                            → ≡-substTrg B′≡B (≡-substTrg B≡B′ F) ≡ F
  ≡-substTrgTrg-contract ≡-refl ≡-refl  =  ≡-refl
  ≡-substTrgSrc :   {A A′ B B′ : Node} (A≡A′ : A ≡ A′) (B≡B′ : B ≡ B′) {F : Edge A B}
                 →   ≡-substTrg B≡B′ (≡-substSrc A≡A′ F)
                 ≡   ≡-substSrc A≡A′ (≡-substTrg B≡B′ F)
  ≡-substTrgSrc ≡-refl ≡-refl  =  ≡-refl
  ≡-substSrcTrg :   {A A′ B B′ : Node} (A≡A′ : A ≡ A′) (B≡B′ : B ≡ B′) {F : Edge A B}
                 →   ≡-substSrc A≡A′ (≡-substTrg B≡B′ F)
                 ≡   ≡-substTrg B≡B′ (≡-substSrc A≡A′ F)
  ≡-substSrcTrg ≡-refl ≡-refl  =  ≡-refl
  ≡-substSrcTrg²  :  {A₁ A₂ A₃ B₁ B₂ B₃ : Node}
                      (A₂≡A₃ : A₂ ≡ A₃) (B₂≡B₃ : B₂ ≡ B₃) (A₁≡A₂ : A₁ ≡ A₂) (B₁≡B₂ : B₁ ≡ B₂)
                      {F : Edge A₁ B₁}
                    → ≡-substSrc A₂≡A₃ (≡-substTrg B₂≡B₃ (≡-substSrc A₁≡A₂ (≡-substTrg B₁≡B₂ F)))
                    ≡  ≡-substSrc (A₁≡A₂ ⟨≡≡⟩ A₂≡A₃) (≡-substTrg (B₁≡B₂ ⟨≡≡⟩ B₂≡B₃) F)
  ≡-substSrcTrg² ≡-refl ≡-refl ≡-refl ≡-refl  =  ≡-refl

 open LocalEdgeSetoidAux public
```

To save having to write **open** Eq (Hom A B) or similar for each calculational proof, we introduce a generalisation of the standard library's Relation.Binary.PreorderReasoning that is parameterised with a LocalSetoid instead of just with a Setoid or Preorder, with the result that the exported symbols have two additional object parameters. By using our SetoidCalc wrapper over the standard library's Relation.Binary.PreorderReasoning, we also add some commonly used proof-step abbreviations for "backwards" steps, and for steps involving propositional equality.

```
module LocalSetoidCalc {i j k : Level} {Node : Set i} (LES : LocalSetoid Node j k) where
  module _ {A B : Node} where
    open SetoidCalc (LES A B) public
```

```
retractLES :  {i₁ i₂ j k : Level} {Node₁ : Set i₁} {Node₂ : Set i₂}
           → (F : Node₂ → Node₁) → LocalSetoid Node₁ j k → LocalSetoid Node₂ j k
retractLES F LES x y  =  LES (F x) (F y)
```

```
retract²LES :  {i₁ i₂ j₁ j₂ k : Level} {Node₁ : Set i₁} {Node₂ : Set i₂}
            → (LES₁ : LocalSetoid Node₁ j₁ k)
            → {LEs₂ : Node₂ → Node₂ → Set j₂}
            → (FN : Node₂ → Node₁)
```

```
          → (FE : {n₁ n₂ : Node₂} → LEs₂ n₁ n₂ → ⌊ LES₁ (FN n₁) (FN n₂) ⌋)
          → LocalSetoid Node₂ j₂ k
retract²LES LES₁ FN FE x y = retractSetoid (LES₁ (FN x) (FN y)) (FE {x} {y})
```

Where a LocalSetoid is not injective, it can be useful to "attach" the two end nodes to edges.

```
data Attach {i j k : Level} {Node : Set i} (LES : LocalSetoid Node j k)
          : Node → Node → Set (i ⊍ j)
  where
    ATTACH : (x : Node) → (y : Node) → Setoid.Carrier (LES x y) → Attach LES x y
srcᵃ  : {i j k : Level} {Node : Set i} {LES : LocalSetoid Node j k} {x y : Node}
      → Attach LES x y → Node
srcᵃ  (ATTACH x y e) = x
trgᵃ  : {i j k : Level} {Node : Set i} {LES : LocalSetoid Node j k} {x y : Node}
      → Attach LES x y → Node
trgᵃ  (ATTACH x y e) = y
edgeᵃ : {i j k : Level} {Node : Set i} {LES : LocalSetoid Node j k} {x y : Node}
      → Attach LES x y → Setoid.Carrier (LES x y)
edgeᵃ (ATTACH x y e) = e

Attach-≈ : {i j k : Level} {Node : Set i} {LES : LocalSetoid Node j k}
         → {x y : Node} → Rel (Attach LES x y) k
Attach-≈ {LES = LES} {x} {y} a₁ a₂ = Setoid._≈_ (LES x y) (edgeᵃ a₁) (edgeᵃ a₂)

attachLES : {i j k : Level} {Node : Set i}
          → LocalSetoid Node j k → LocalSetoid Node (i ⊍ j) k
attachLES LES x y = let open LocalEdgeSetoid LES in record
  {Carrier        = Attach LES x y
  ; _≈_           = Attach-≈
  ; isEquivalence = record {refl = ≈-refl; sym = ≈-sym; trans = ≈-trans}
  }

srcᵃ≡source : {i j k : Level} {Node : Set i} {LES : LocalSetoid Node j k} {x y : Node}
            → (a : Attach LES x y) → srcᵃ a ≡ LocalEdgeSetoid.source (attachLES LES) a
srcᵃ≡source {x = x} {y} (ATTACH .x .y e) = ≡-refl
trgᵃ≡target : {i j k : Level} {Node : Set i} {LES : LocalSetoid Node j k} {x y : Node}
            → (a : Attach LES x y) → trgᵃ a ≡ LocalEdgeSetoid.target (attachLES LES) a
trgᵃ≡target {x = x} {y} (ATTACH .x .y e) = ≡-refl
```

A homomorphism between LES graphs consists of a node mapping, and a family of edge mappings with types determined by the node mappings. The edge mappings also need to respect the local equivalence relations.

```
record LESHom {i₁ j₁ k₁ : Level} {Node₁ : Set i₁} (LES₁ : LocalSetoid Node₁ j₁ k₁)
              {i₂ j₂ k₂ : Level} {Node₂ : Set i₂} (LES₂ : LocalSetoid Node₂ j₂ k₂)
              : Set (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂ ⊍ k₁ ⊍ k₂) where
  open LocalEdgeSetoid LES₁ using () renaming (Edge to Edge₁; _≈_ to _≈₁_)
  open LocalEdgeSetoid LES₂ using () renaming (Edge to Edge₂; _≈_ to _≈₂_)
  field
    mapN : Node₁ → Node₂
    mapE : {X Y : Node₁} → Edge₁ X Y → Edge₂ (mapN X) (mapN Y)
    congE : {X Y : Node₁} → {e₁ e₂ : Edge₁ X Y} → e₁ ≈₁ e₂ → mapE e₁ ≈₂ mapE e₂
```

Renamings

For contexts where reasoning in different LocalSetoids is required, we add "decorated" variants of the $\mathsf{LocalEdgeSetoid}_0$ interface as defined in Relation.Binary.Setoid.Utils (Sect. 2.3), and also of the LocalSetoidCalc interface:

```
module LocalEdgeSetoid₀ {i j k : Level} {Node : Set i} (LES : LocalSetoid Node j k) where
  Edge₀ : Node → Node → Set j
  Edge₀ = Setoid.Carrier ∘₂ LES
  module _ {A B : Node} where
    open Setoid₀ (LES A B) public hiding (Carrier₀)

module LocalEdgeSetoid₁ {i j k : Level} {Node : Set i} (LES : LocalSetoid Node j k) where
  Edge₁ : Node → Node → Set j
  Edge₁ = Setoid.Carrier ∘₂ LES
  module _ {A B : Node} where
    open Setoid₁ (LES A B) public hiding (Carrier₁)

module LocalEdgeSetoid₂ {i j k : Level} {Node : Set i} (LES : LocalSetoid Node j k) where
  Edge₂ : Node → Node → Set j
  Edge₂ = Setoid.Carrier ∘₂ LES
  module _ {A B : Node} where
    open Setoid₂ (LES A B) public hiding (Carrier₂)

module LocalEdgeSetoid₃ {i j k : Level} {Node : Set i} (LES : LocalSetoid Node j k) where
  Edge₃ : Node → Node → Set j
  Edge₃ = Setoid.Carrier ∘₂ LES
  module _ {A B : Node} where
    open Setoid₃ (LES A B) public hiding (Carrier₃)

module LocalEdgeSetoid₄ {i j k : Level} {Node : Set i} (LES : LocalSetoid Node j k) where
  Edge₄ : Node → Node → Set j
  Edge₄ = Setoid.Carrier ∘₂ LES
  module _ {A B : Node} where
    open Setoid₄ (LES A B) public hiding (Carrier₄)

module LocalEdgeSetoidR {i j k : Level} {Node : Set i} (LES : LocalSetoid Node j k) where
  EdgeR : Node → Node → Set j
  EdgeR = Setoid.Carrier ∘₂ LES
  module _ {A B : Node} where
    open SetoidR (LES A B) public hiding (R₀)

module LocalSetoidCalc₀ {i j k : Level} {Node : Set i} (LES : LocalSetoid Node j k) where
  Edge₀ : Node → Node → Set j
  Edge₀ = Setoid.Carrier ∘₂ LES
  module _ {A B : Node} where
    open SetoidCalc₀ (LES A B) public hiding (Carrier₀)

module LocalSetoidCalc₁ {i j k : Level} {Node : Set i} (LES : LocalSetoid Node j k) where
  Edge₁ : Node → Node → Set j
  Edge₁ = Setoid.Carrier ∘₂ LES
  module _ {A B : Node} where
    open SetoidCalc₁ (LES A B) public hiding (Carrier₁)

module LocalSetoidCalc₂ {i j k : Level} {Node : Set i} (LES : LocalSetoid Node j k) where
  Edge₂ : Node → Node → Set j
  Edge₂ = Setoid.Carrier ∘₂ LES
  module _ {A B : Node} where
    open SetoidCalc₂ (LES A B) public hiding (Carrier₂)
```

```
module LocalSetoidCalc₃ {i j k : Level} {Node : Set i} (LES : LocalSetoid Node j k) where
  Edge₃ : Node → Node → Set j
  Edge₃ = Setoid.Carrier ∘₂ LES
  module _ {A B : Node} where
    open SetoidCalc₃ (LES A B) public hiding (Carrier₃)
```

```
module LocalSetoidCalc₄ {i j k : Level} {Node : Set i} (LES : LocalSetoid Node j k) where
  Edge₄ : Node → Node → Set j
  Edge₄ = Setoid.Carrier ∘₂ LES
  module _ {A B : Node} where
    open SetoidCalc₄ (LES A B) public hiding (Carrier₄)
```

```
module LocalSetoidCalcR {i j k : Level} {Node : Set i} (LES : LocalSetoid Node j k) where
  EdgeR : Node → Node → Set j
  EdgeR = Setoid.Carrier ∘₂ LES
  module _ {A B : Node} where
    open SetoidCalcR (LES A B) public hiding (R₀)
```

3.5 Categoric.CompOp

One way to think about generalising a graph to a locally ordered category, or even to a 2-category, is the following:

- A graph has, for any pair (x, y) of vertices, a set of edges from x to y.
- A semigroupoid is a graph where the vertices are called objects and the edges are called morphisms, and equality reasoning on morphisms is available, so assigning a Setoid of morphisms (or edges) to any two objects (or vertices) is natural.

Since the term "homset" is customarily used for the individual collection of morphisms from one particular object to one second object, we use the expression "Hom A B" for the setoid-structured collection of morphisms from A to B, and "Mor A B" for the type of morphisms from A to B.

```
module LocalHomSetoid {i j k : Level} {Obj : Set i} (Hom : LocalSetoid Obj j k) where
  open LocalEdgeSetoid Hom public renaming
    (Edge to Mor   -- : Obj → Obj → Set j
    )
```

A semigroupoid composition operator in the context of such a LocalHomSetoid needs to respect the equivalences on the individual setoids, and needs to be associative. We first define appropriate type synonyms:

```
module _ {i j k : Level} {Obj : Set i} (Hom : LocalSetoid Obj j k) where
  open LocalHomSetoid Hom

  Comp : Set (i ⊍ j)
  Comp = {A B C : Obj} → Mor A B → Mor B C → Mor A C

  module _ (_⨾_ : Comp) where
    ⨾-Cong  : Set (i ⊍ j ⊍ k)
    ⨾-Cong  = {A B C : Obj} {f₁ f₂ : Mor A B} {g₁ g₂ : Mor B C}
            → f₁ ≈ f₂ → g₁ ≈ g₂ → (f₁ ⨾ g₁) ≈ (f₂ ⨾ g₂)
    ⨾-Assoc : Set (i ⊍ j ⊍ k)
    ⨾-Assoc = {A B C D : Obj} {f : Mor A B} {g : Mor B C} {h : Mor C D}
            → ((f ⨾ g) ⨾ h) ≈ (f ⨾ (g ⨾ h))
```

```
record CompOp {i j k : Level} {Obj : Set i} (Hom : LocalSetoid Obj j k)
              : Set (i ⊍ j ⊍ k) where
  open LocalHomSetoid Hom
```

```
  infixr 9 _⨾_
  field
    _⨾_     : Comp Hom
    ⨾-cong  : ⨾-Cong  Hom _⨾_
    ⨾-assoc : ⨾-Assoc Hom _⨾_
```

For the category-theoretic concept of duality, wich involves reversing the direction of the morphisms, we will use the word "opposite". (The word "dual" will be used for order duality.) Given a CompOp over a LocalHomSetoid, it is straight-forward to define the opposite CompOp over the opposite LocalHomSetoid, that is, the LocalHomSetoid resulting from flipping the arguments.

```
oppositeCompOp : {i j k : Level} {Obj : Set i} {Hom : LocalSetoid Obj j k}
               → CompOp Hom → CompOp (λ A B → Hom B A)
oppositeCompOp {Hom = Hom} compOp = let open CompOp compOp in record
  {_⨾_ = λ f g → g ⨾ f
  ;⨾-cong  = λ f₁≈f₂ g₁≈g₂ → ⨾-cong g₁≈g₂ f₁≈f₂
  ;⨾-assoc = λ {A} {B} {C} {D} → Setoid.sym (Hom D A) ⨾-assoc
  }
```

Adding retract and attach combinators for CompOp is straight-forward:

```
retractCompOp : {i₁ i₂ j k : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
              → (F : Obj₂ → Obj₁)
              → {Hom : LocalSetoid Obj₁ j k} → CompOp Hom → CompOp (retractLES F Hom)
retractCompOp F compOp = let open CompOp compOp
  in record {_⨾_ = _⨾_; ⨾-cong = ⨾-cong; ⨾-assoc = ⨾-assoc}


Attach-⨾ : {i j k : Level} {Obj : Set i}
           (Hom : LocalSetoid Obj j k)
           (_⨾_ : Transitive (Setoid.Carrier ∘₂ Hom))
  →        Transitive (Setoid.Carrier ∘₂ attachLES Hom)
Attach-⨾ Hom _⨾_ {A} {B} {C} a₁ a₂ = ATTACH A C (edgeᵃ a₁ ⨾ edgeᵃ a₂)
attachCompOp : {i j k : Level} {Obj : Set i} {Hom : LocalSetoid Obj j k}
             → CompOp Hom → CompOp (attachLES Hom)
attachCompOp {Hom = Hom} compOp = let open CompOp compOp
  in record {_⨾_ = Attach-⨾ Hom _⨾_; ⨾-cong = ⨾-cong; ⨾-assoc = ⨾-assoc}
```

3.6 Categoric.CompOpProps1

3.6.1 Simplification of ≡-substSrc and ≡-substTrg in Compositions

We provide a few simplification rules for reasoning with propositional equality on objects in the context of morphism composition. Although many of the special cases listed here can be obtained easily from more primitive items, the special cases still serve to considerably abbreviate more involved substitution reasoning.

```
module Comp-≡-substSrcTrg {i j k : Level} {Obj : Set i}
                          (Hom : LocalSetoid Obj j k) (_⨾_ : Comp Hom) where
  open LocalHomSetoid Hom
  ⨾-≡-subst₁ : {A B C : Obj} {F : Mor A B} {G : Mor B C} {A′ : Obj}
             → (A≡A′ : A ≡ A′)
             → (≡-substSrc A≡A′ F ⨾ G) ≡ ≡-substSrc A≡A′ (F ⨾ G)
  ⨾-≡-subst₁ ≡-refl = ≡-refl
```

```
  ⨾-≡˘-subst₁ : {A B C : Obj} {F : Mor A B} {G : Mor B C} {A′ : Obj}
              → (A′≡A : A′ ≡ A)
              → (≡˘-substSrc A′≡A F ⨾ G) ≡ ≡˘-substSrc A′≡A (F ⨾ G)
  ⨾-≡˘-subst₁ ≡-refl = ≡-refl

  ⨾-≡-subst₂ : {A B C : Obj} {F : Mor A B} {G : Mor B C} {B′ : Obj}
              → (B≡B′ : B ≡ B′)
              → (≡-substTrg B≡B′ F ⨾ ≡-substSrc B≡B′ G) ≡ (F ⨾ G)
  ⨾-≡-subst₂ ≡-refl = ≡-refl

  ⨾-≡˘-subst₂ : {A B C : Obj} {F : Mor A B} {G : Mor B C} {B′ : Obj}
              → (B′≡B : B′ ≡ B)
              → (≡˘-substTrg B′≡B F ⨾ ≡˘-substSrc B′≡B G) ≡ (F ⨾ G)
  ⨾-≡˘-subst₂ ≡-refl = ≡-refl

  ⨾-≡-subst₂-irr : {A B C : Obj} {F : Mor A B} {G : Mor B C} {B′ : Obj}
              → (B≡₁B′ B≡₂B′ : B ≡ B′)
              → (≡-substTrg B≡₁B′ F ⨾ ≡-substSrc B≡₂B′ G) ≡ (F ⨾ G)
  ⨾-≡-subst₂-irr ≡-refl ≡-refl = ≡-refl

  ≡-substTrg-⨾ : {A B B′ C : Obj} {F : Mor A B} {G : Mor B′ C}
                 → (B≡B′ : B ≡ B′)
                 → (≡-substTrg B≡B′ F ⨾ G) ≡ (F ⨾ ≡˘-substSrc B≡B′ G)
  ≡-substTrg-⨾ ≡-refl = ≡-refl

  ⨾-≡-substSrc : {A B B′ C : Obj} {F : Mor A B′} {G : Mor B C}
                 → (B≡B′ : B ≡ B′)
                 → (F ⨾ ≡-substSrc B≡B′ G) ≡ (≡˘-substTrg B≡B′ F ⨾ G)
  ⨾-≡-substSrc ≡-refl = ≡-refl

  ⨾-≡-subst₃ : {A B C : Obj} {F : Mor A B} {G : Mor B C} {C′ : Obj}
              → (C≡C′ : C ≡ C′)
              → (F ⨾ ≡-substTrg C≡C′ G) ≡ ≡-substTrg C≡C′ (F ⨾ G)
  ⨾-≡-subst₃ ≡-refl = ≡-refl

  ⨾-≡˘-subst₃ : {A B C : Obj} {F : Mor A B} {G : Mor B C} {C′ : Obj}
              → (C′≡C : C′ ≡ C)
              → (F ⨾ ≡˘-substTrg C′≡C G) ≡ ≡˘-substTrg C′≡C (F ⨾ G)
  ⨾-≡˘-subst₃ ≡-refl = ≡-refl

  ≡-substSrcTrg-⨾ : {A₁ A₂ B₁ B₂ B₃ C₁ C₂ : Obj} {F : Mor A₁ B₁} {G : Mor B₃ C₁}
                   → (A₁≡A₂ : A₁ ≡ A₂) (B₁≡B₂ : B₁ ≡ B₂) (B₃≡B₂ : B₃ ≡ B₂) (C₁≡C₂ : C₁ ≡ C₂)
                   → (≡-substSrc A₁≡A₂ (≡-substTrg B₁≡B₂ F) ⨾ ≡-substSrc B₃≡B₂ (≡-substTrg C₁≡C₂ G))
                   ≡ (≡-substSrc A₁≡A₂ (≡-substTrg C₁≡C₂ (F ⨾ ≡-substSrc (B₃≡B₂ ⟨≡≡˘⟩ B₁≡B₂) G)))
  ≡-substSrcTrg-⨾ ≡-refl ≡-refl ≡-refl ≡-refl = ≡-refl

  ≡-substSrcTrg-⨾-irr : {A₁ A₂ B₁ B₂ C₁ C₂ : Obj} {F : Mor A₁ B₁} {G : Mor B₁ C₁}
                   → (A₁≡A₂ : A₁ ≡ A₂) (B₁≡B₂ B₁≡′B₂ : B₁ ≡ B₂) (C₁≡C₂ : C₁ ≡ C₂)
                   → (≡-substSrc A₁≡A₂ (≡-substTrg B₁≡B₂ F) ⨾ ≡-substSrc B₁≡′B₂ (≡-substTrg C₁≡C₂ G))
                   ≡ (≡-substSrc A₁≡A₂ (≡-substTrg C₁≡C₂ (F ⨾ G)))
  ≡-substSrcTrg-⨾-irr ≡-refl ≡-refl ≡-refl ≡-refl = ≡-refl
```

3.6.2 Abbreviations for Applications of ⨾-cong

The following abbreviations for applications of ⨾-cong are useful in proofs:

```
module CompCongProps {i j k : Level} {Obj : Set i} (Hom : LocalSetoid Obj j k)
                     (_⨾_ : Comp Hom) (⨾-cong : ⨾-Cong Hom _⨾_) where
  open LocalHomSetoid Hom
  ⨾-cong₁ : {A B C : Obj} {g : Mor B C}
          → ((λ (f : Mor A B) → f ⨾ g) Preserves (_≈_ {A} {B}) ⟶ (_≈_ {A} {C}))
  ⨾-cong₁ {_} {B} {C} eq = ⨾-cong eq (Setoid.refl (Hom B C))

  ⨾-cong₂ : {A B C : Obj} {f : Mor A B}
          → ((_⨾_ {A} {B} {C} f) Preserves (_≈_ {B} {C}) ⟶ (_≈_ {A} {C}))
```

```
  ⨾-cong₂ {A} {B} = ⨾-cong (Setoid.refl (Hom A B))
  ⨾-cong₁₁ : {A B C D : Obj} {g : Mor B C} {h : Mor C D}
           → ((λ (f : Mor A B) → (f ⨾ g) ⨾ h) Preserves (_≈_ {A} {B}) ⟶ (_≈_ {A} {D}))
  ⨾-cong₁₁ eq = ⨾-cong₁ (⨾-cong₁ eq)
  ⨾-cong₁₂ : {A B C D : Obj} {f : Mor A B} {h : Mor C D}
           → ((λ (g : Mor B C) → (f ⨾ g) ⨾ h) Preserves (_≈_ {B} {C}) ⟶ (_≈_ {A} {D}))
  ⨾-cong₁₂ eq = ⨾-cong₁ (⨾-cong₂ eq)
  ⨾-cong₂₁ : {A B C D : Obj} {f : Mor A B} {h : Mor C D}
           → ((λ (g : Mor B C) → f ⨾ (g ⨾ h)) Preserves (_≈_ {B} {C}) ⟶ (_≈_ {A} {D}))
  ⨾-cong₂₁ eq = ⨾-cong₂ (⨾-cong₁ eq)
  ⨾-cong₂₂ : {A B C D : Obj} {f : Mor A B} {g : Mor B C}
           → ((λ (h : Mor C D) → f ⨾ (g ⨾ h)) Preserves (_≈_ {C} {D}) ⟶ (_≈_ {A} {D}))
  ⨾-cong₂₂ eq = ⨾-cong₂ (⨾-cong₂ eq)
  ⨾-cong₁₁₁ : {A B C D E : Obj} {g : Mor B C} {h : Mor C D} {j : Mor D E}
            → ((λ (f : Mor A B) → ((f ⨾ g) ⨾ h) ⨾ j) Preserves (_≈_ {A} {B}) ⟶ (_≈_ {A} {E}))
  ⨾-cong₁₁₁ eq = ⨾-cong₁₁ (⨾-cong₁ eq)
  ⨾-cong₁₁₂ : {A B C D E : Obj} {f : Mor A B} {h : Mor C D} {j : Mor D E}
            → ((λ (g : Mor B C) → ((f ⨾ g) ⨾ h) ⨾ j) Preserves (_≈_ {B} {C}) ⟶ (_≈_ {A} {E}))
  ⨾-cong₁₁₂ eq = ⨾-cong₁₁ (⨾-cong₂ eq)
  ⨾-cong₁₂₁ : {A B C D E : Obj} {f : Mor A B} {h : Mor C D} {j : Mor D E}
            → ((λ (g : Mor B C) → (f ⨾ (g ⨾ h)) ⨾ j) Preserves (_≈_ {B} {C}) ⟶ (_≈_ {A} {E}))
  ⨾-cong₁₂₁ eq = ⨾-cong₁₂ (⨾-cong₁ eq)
  ⨾-cong₁₂₂ : {A B C D E : Obj} {f : Mor A B} {g : Mor B C} {j : Mor D E}
            → ((λ (h : Mor C D) → (f ⨾ (g ⨾ h)) ⨾ j) Preserves (_≈_ {C} {D}) ⟶ (_≈_ {A} {E}))
  ⨾-cong₁₂₂ eq = ⨾-cong₁₂ (⨾-cong₂ eq)
  ⨾-cong₂₁₁ : {A B C D E : Obj} {f : Mor A B} {h : Mor C D} {j : Mor D E}
            → ((λ (g : Mor B C) → f ⨾ ((g ⨾ h) ⨾ j)) Preserves (_≈_ {B} {C}) ⟶ (_≈_ {A} {E}))
  ⨾-cong₂₁₁ eq = ⨾-cong₂₁ (⨾-cong₁ eq)
  ⨾-cong₂₁₂ : {A B C D E : Obj} {f : Mor A B} {g : Mor B C} {j : Mor D E}
            → ((λ (h : Mor C D) → f ⨾ ((g ⨾ h) ⨾ j)) Preserves (_≈_ {C} {D}) ⟶ (_≈_ {A} {E}))
  ⨾-cong₂₁₂ eq = ⨾-cong₂₁ (⨾-cong₂ eq)
  ⨾-cong₂₂₁ : {A B C D E : Obj} {f : Mor A B} {g : Mor B C} {j : Mor D E}
            → ((λ (h : Mor C D) → f ⨾ (g ⨾ (h ⨾ j))) Preserves (_≈_ {C} {D}) ⟶ (_≈_ {A} {E}))
  ⨾-cong₂₂₁ eq = ⨾-cong₂₂ (⨾-cong₁ eq)
  ⨾-cong₂₂₂ : {A B C D E : Obj} {f : Mor A B} {g : Mor B C} {h : Mor C D}
            → ((λ (j : Mor D E) → f ⨾ (g ⨾ (h ⨾ j))) Preserves (_≈_ {D} {E}) ⟶ (_≈_ {A} {E}))
  ⨾-cong₂₂₂ eq = ⨾-cong₂₂ (⨾-cong₂ eq)
```

3.6.3 module CompOpProps$_1$

We now define a single module parameterised over a CompOp re-exporting the above properties and collecting further basic composition-related properties and derived definitions.

```
module CompOpProps₁ {i j k : Level} {Obj : Set i} {Hom : LocalSetoid Obj j k}
                    (compOp : CompOp Hom) where
  open LocalHomSetoid    Hom
  open LocalSetoidCalc   Hom
  open CompOp            compOp

  open Comp-≡-substSrcTrg Hom _⨾_          public
  open CompCongProps     Hom _⨾_ ⨾-cong  public
```

Abbreviations for Applications of ⨾-cong and ⨾-assoc

The following abbreviations for applications of ⨾-cong and ⨾-assoc are useful in proofs:

```
⨾-assocL  : {A B C D : Obj} {f : Mor A B} {g : Mor B C} {h : Mor C D}
          → f ⨾ (g ⨾ h) ≈ (f ⨾ g) ⨾ h
⨾-assocL {A} {B} {C} {D} = Setoid.sym (Hom A D) ⨾-assoc
⨾-assoc₄  : {A B C D E : Obj} {f : Mor A B} {g : Mor B C} {h : Mor C D} {j : Mor D E}
          → ((f ⨾ g) ⨾ h) ⨾ j ≈ f ⨾ (g ⨾ (h ⨾ j))
⨾-assoc₄  = ⨾-assoc ⟨≈≈⟩ ⨾-assoc
⨾-assocL₄ : {A B C D E : Obj} {f : Mor A B} {g : Mor B C} {h : Mor C D} {j : Mor D E}
          → f ⨾ (g ⨾ (h ⨾ j)) ≈ ((f ⨾ g) ⨾ h) ⨾ j
⨾-assocL₄ = ⨾-assocL ⟨≈≈⟩ ⨾-assocL
⨾-assoc₃₊₁  : {A B C D E : Obj} {f : Mor A B} {g : Mor B C} {h : Mor C D} {j : Mor D E}
          → (f ⨾ g ⨾ h) ⨾ j ≈ f ⨾ (g ⨾ (h ⨾ j))
⨾-assoc₃₊₁  = ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⨾-assoc
⨾-assocL₃₊₁  : {A B C D E : Obj} {f : Mor A B} {g : Mor B C} {h : Mor C D} {j : Mor D E}
          →  f ⨾ (g ⨾ (h ⨾ j)) ≈ (f ⨾ g ⨾ h) ⨾ j
⨾-assocL₃₊₁  = ⨾-cong₂ ⨾-assocL ⟨≈≈⟩ ⨾-assocL

⨾-₂₂assoc₁₂₁     :     {A B C D E : Obj} {f : Mor A B} {g : Mor B C} {h : Mor C D} {j : Mor D E}
                 → (f ⨾ g) ⨾ (h ⨾ j) ≈ f ⨾ (g ⨾ h) ⨾ j
⨾-₂₂assoc₁₂₁ = ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⨾-assocL
⨾-₁₂₁assoc₂₂     :     {A B C D E : Obj} {f : Mor A B} {g : Mor B C} {h : Mor C D} {j : Mor D E}
                 → f ⨾ (g ⨾ h) ⨾ j ≈ (f ⨾ g) ⨾ (h ⨾ j)
⨾-₁₂₁assoc₂₂ = ⨾-cong₂ ⨾-assoc ⟨≈≈⟩ ⨾-assocL

on-⨾-assoc   : {A B B′ C C′ D : Obj}
               → {f : Mor A B} {g : Mor B C} {h : Mor C D}
               → {f′ : Mor A B′} {g′ : Mor B′ C′} {h′ : Mor C′ D}
               → (f ⨾ (g ⨾ h)) ≈ (f′ ⨾ (g′ ⨾ h′))
               → ((f ⨾ g) ⨾ h) ≈ ((f′ ⨾ g′) ⨾ h′)
on-⨾-assoc eq = ⨾-assoc ⟨≈≈⟩ eq ⟨≈≈⟩ ⨾-assocL
on-⨾-assocL   : {A B B′ C C′ D : Obj}
                → {f : Mor A B} {g : Mor B C} {h : Mor C D}
                → {f′ : Mor A B′} {g′ : Mor B′ C′} {h′ : Mor C′ D}
                → (f ⨾ g) ⨾ h ≈ (f′ ⨾ g′) ⨾ h′
                → f ⨾ (g ⨾ h) ≈ f′ ⨾ (g′ ⨾ h′)
on-⨾-assocL eq = ⨾-assocL ⟨≈≈⟩ eq ⟨≈≈⟩ ⨾-assoc

⨾-cong₁₂&₂ : {A B C C′ D : Obj} {f : Mor A B} {g : Mor B C} {h : Mor C D}
           → {g′ : Mor B C′} {h′ : Mor C′ D}
           → g ⨾ h ≈ g′ ⨾ h′
           → (f ⨾ g) ⨾ h ≈ (f ⨾ g′) ⨾ h′
⨾-cong₁₂&₂ eq = on-⨾-assoc (⨾-cong₂ eq)
⨾-cong₁&₂₁ : {A B B′ C D : Obj} {f : Mor A B} {g : Mor B C} {h : Mor C D}
           → {f′ : Mor A B′} {g′ : Mor B′ C}
           → f ⨾ g ≈ f′ ⨾ g′
           → f ⨾ (g ⨾ h) ≈ f′ ⨾ (g′ ⨾ h)
⨾-cong₁&₂₁ eq = on-⨾-assocL (⨾-cong₁ eq)
⨾-cong₁₂&₂₁ : {A B C D   E : Obj} {f : Mor A B} {g : Mor B C} {h : Mor C D} {j : Mor D E}
              → {gh′ : Mor B D}
              → g ⨾ h ≈ gh′
              → (f ⨾ g) ⨾ (h ⨾ j) ≈ f ⨾ gh′ ⨾ j
⨾-cong₁₂&₂₁ eq = ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ eq)
```

Idempotence and Identity Predicates

Since we have morphism equality, we can formalise the properties of idempotence and (left- and right-) identities. We also show that each of these properties is invariant under morphism equality.

```
isIdempotent : {A : Obj} → (f : Mor A A) → Set k
isIdempotent f = (f ⨾ f) ≈ f

isIdempotent-subst : {A : Obj} → {f g : Mor A A} → f ≈ g → isIdempotent f → isIdempotent g
isIdempotent-subst {A} {f} {g} f≈g f⨾f≈f = ≈-begin
    g ⨾ g
  ≈˘⟨ ⨾-cong f≈g f≈g ⟩
    f ⨾ f
  ≈⟨ f⨾f≈f ⟩
      f
  ≈⟨ f≈g ⟩
      g
  □

isLeftIdentity : {A : Obj} → Mor A A → Set (i ⊍ j ⊍ k)
isLeftIdentity {A} I   = {B : Obj} {R : Mor A B} → I ⨾ R ≈ R
isRightIdentity : {A : Obj} → Mor A A → Set (i ⊍ j ⊍ k)
isRightIdentity {A} I = {B : Obj} {R : Mor B A} → R ⨾ I ≈ R
isIdentity : {A : Obj} → Mor A A → Set (i ⊍ j ⊍ k)
isIdentity I = isLeftIdentity I × isRightIdentity I

isLeftIdentity-subst : {A : Obj} → {I J : Mor A A} → I ≈ J → isLeftIdentity I → isLeftIdentity J
isLeftIdentity-subst {A} {I} {J} I≈J left {B} {R} = ≈-begin
    J ⨾ R
  ≈˘⟨ ⨾-cong₁ I≈J ⟩
    I ⨾ R
  ≈⟨ left ⟩
      R
  □
isRightIdentity-subst : {A : Obj} → {I J : Mor A A} → I ≈ J → isRightIdentity I → isRightIdentity J
isRightIdentity-subst {A} {I} {J} I≈J right {B} {R} = ≈-begin
    R ⨾ J
  ≈˘⟨ ⨾-cong₂ I≈J ⟩
    R ⨾ I
  ≈⟨ right ⟩
      R
  □
isIdentity-subst : {A : Obj} → {I J : Mor A A} → I ≈ J → isIdentity I → isIdentity J
isIdentity-subst {A} {I} {J} I≈J (left, right) = isLeftIdentity-subst   I≈J left
                                               ,  isRightIdentity-subst I≈J right
```

Splittings

Freyd and Scedrov (1990, 1.281) propose the concept of *splitting* idempotents, where idempotence is actually a consequence of the splitting properties:

```
record Splitting {A : Obj} (e : Mor A A) : Set (i ⊍ j ⊍ k) where
  field
    {obj}    : Obj
    mor₁  : Mor A obj
    mor₂  : Mor obj A
    factors  : mor₁ ⨾ mor₂ ≈ e
    splitId   : isIdentity (mor₂ ⨾ mor₁)
  idempotent : isIdempotent e
  idempotent = ≈-begin
```

```
    e ⨾ e
  ≈˘⟨ ⨾-cong factors factors ⟩
    (mor₁ ⨾ mor₂) ⨾ (mor₁ ⨾ mor₂)
  ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⨾-assocL ⟩
    mor₁ ⨾ (mor₂ ⨾ mor₁) ⨾ mor₂
  ≈⟨ ⨾-cong₂ (proj₁ splitId) ⟩
    mor₁ ⨾ mor₂
  ≈⟨ factors ⟩
    e
  □
```

Monomorphism and Epimorphism Predicates

The standard category-theoretic definitions of monos and epis can be expressed already in semigrouoids:

```
isMono : {A B : Obj} → Mor A B → Set (i ⊍ j ⊍ k)
isMono {A} {B} F = {Z : Obj} {R S : Mor Z A} → (R ⨾ F) ≈ (S ⨾ F) → R ≈ S

isEpi : {A B : Obj} → Mor A B → Set (i ⊍ j ⊍ k)
isEpi {A} {B} F = {Z : Obj} {R S : Mor B Z} → (F ⨾ R) ≈ (F ⨾ S) → R ≈ S

isMono-subst : {A B : Obj} {F G : Mor A B} → F ≈ G → isMono F → isMono G
isMono-subst {A} {B} {F} {G} F≈G F-isMono {Z} {R} {S} R⨾G≈S⨾G = F-isMono {Z} {R} {S}
  (⨾-cong₂ F≈G ⟨≈≈⟩ R⨾G≈S⨾G ⟨≈≈˘⟩ ⨾-cong₂ F≈G)

isEpi-subst : {A B : Obj} {F G : Mor A B} → F ≈ G → isEpi F → isEpi G
isEpi-subst {A} {B} {F} {G} F≈G F-isEpi {Z} {R} {S} G⨾R≈G⨾S = F-isEpi {Z} {R} {S}
  (⨾-cong₁ F≈G ⟨≈≈⟩ G⨾R≈G⨾S ⟨≈≈˘⟩ ⨾-cong₁ F≈G)

⨾-isMono : {A B C : Obj} {F : Mor A B} {G : Mor B C}
         → isMono F → isMono G → isMono (F ⨾ G)
⨾-isMono isMono-F isMono-G R⨾F⨾G≈S⨾F⨾G = isMono-F (isMono-G (on-⨾-assoc R⨾F⨾G≈S⨾F⨾G))

⨾-isMono-decomp : {A B C : Obj} {F : Mor A B} {G : Mor B C}
                → isMono (F ⨾ G) → isMono F
⨾-isMono-decomp isMono-F⨾G R⨾F≈S⨾F = isMono-F⨾G (⨾-cong₁&₂₁ R⨾F≈S⨾F)

⨾-isEpi : {A B C : Obj} {F : Mor A B} {G : Mor B C}
        → isEpi F → isEpi G → isEpi (F ⨾ G)
⨾-isEpi isEpi-F isEpi-G F⨾G⨾R≈F⨾G⨾S = isEpi-G (isEpi-F (on-⨾-assocL F⨾G⨾R≈F⨾G⨾S))

⨾-isEpi-decomp   : {A B C : Obj} {F : Mor A B} {G : Mor B C}
  → isEpi (F ⨾ G) → isEpi G
⨾-isEpi-decomp isEpi-F⨾G G⨾R≈G⨾S = isEpi-F⨾G (⨾-cong₁₂&₂ G⨾R≈G⨾S)
```

(If we had chosen to obtain ⨾-isEpi by renaming of ⨾-isMono from the opposite semigroupoid, it would have ended up with reversed argument order: ⨾-isEpi : isEpi F → isEpi G → isEpi (G ⨾ F).)

Morphism Retraction

```
retract²CompOp :     {i₂ j₂ : Level} {Obj₂ : Set i₂} {Mor₂ : Obj₂ → Obj₂ → Set j₂}
               →     (_⨾₂_ : {A B C : Obj₂} → Mor₂ A B → Mor₂ B C → Mor₂ A C)
               →     (FO : Obj₂ → Obj)
               →     (FM : {A B : Obj₂} → Mor₂ A B → Mor (FO A) (FO B))
               →     (FM-⨾₂ : {A B C : Obj₂} {f : Mor₂ A B} {g : Mor₂ B C}
                           → FM (f ⨾₂ g) ≈ FM f ⨾ FM g)
               →     CompOp (retract²LES Hom FO FM)
retract²CompOp _⨾₂_ FO FM FM-⨾₂ = record
  {_⨾_ = _⨾₂_
  ; ⨾-cong = λ {A B C f₁ f₂ g₁ g₂} f₁≈f₂ g₁≈g₂ → ≈-begin
        FM (f₁ ⨾₂ g₁)
    ≈⟨ FM-⨾₂ ⟩
```

```
        FM f₁ ⨾ FM g₁
      ≈⟨ ⨾-cong f₁≈f₂ g₁≈g₂ ⟩
        FM f₂ ⨾ FM g₂
      ≈˘⟨ FM-⨾₂ ⟩
        FM (f₂ ⨾₂ g₂)
      □
    ;⨾-assoc = λ {A B C D f g h} → ≈-begin
        FM ((f ⨾₂ g) ⨾₂ h)
      ≈⟨ FM-⨾₂ ⟩
        FM (f ⨾₂ g) ⨾ FM h
      ≈⟨   ⨾-cong₁ FM-⨾₂ ⟩
        (FM f ⨾ FM g) ⨾ FM h
      ≈⟨ ⨾-assoc ⟩
        FM f ⨾ FM g ⨾ FM h
      ≈˘⟨ ⨾-cong₂ FM-⨾₂ ⟩
        FM f ⨾ FM (g ⨾₂ h)
      ≈˘⟨ FM-⨾₂ ⟩
        FM (f ⨾₂ (g ⨾₂ h))
      □
    }
```

Pre- and Post-Composition Setoid Homomorphisms

Pre-composition and post-composition give rise to Setoid homomorphisms between the respective hom-setoids:

```
  pre-⨾ : {A B C : Obj} (F : Mor A B) → Hom B C ⟶ Hom A C
  pre-⨾ F = record {_⟨$⟩_ = _⨾_ F; cong = ⨾-cong₂}
  post-⨾ : {A B C : Obj} (G : Mor B C) → Hom A B ⟶ Hom A C
  post-⨾ G = record {_⟨$⟩_ = λ F → F ⨾ G; cong = ⨾-cong₁}
```

3.6.4 module SemigroupoidCore

All material up to here is gathered into the module SemigroupoidCore that takes only one explicit CompOp parameter:

```
module SemigroupoidCore {i j k : Level} {Obj : Set i} {Hom : LocalSetoid Obj k j}
                        (compOp : CompOp Hom) where
  open LocalHomSetoid Hom    public
  open LocalSetoidCalc Hom   public
  open CompOp        compOp public
  open CompOpProps₁  compOp public
```

3.7 Categoric.LESGraph.Examples

A diagram, defined in the next section, Sect. 3.17, is a graph homomorphism into the underlying graph of a category or semigroupoid. The source graph of that homomorphism is called the *shape* of the diagram.

First recall the definition viewing a graph as a function assigning any two nodes the setoids of edges from the first to the second:

```
        LocalSetoid : {i : Level} (Node : Set i) (j k : Level) → Set (i ⊍ ℓsuc (j ⊍ k))
        LocalSetoid Node j k = Node → Node → Setoid j k
```

For typical small diagram shape graphs, we will need only small local edge setoids:

```
LocalSetoid₀ : (Node : Set₀) → Set₁
LocalSetoid₀ Node = LocalSetoid Node ℓ₀ ℓ₀
```

3.7.1 Shape Graph for Spans

For example, we use the following object and morphism names for spans:

$$\mathsf{Left} \xleftarrow{\mathsf{left}} \mathsf{Centre} \xrightarrow{\mathsf{right}} \mathsf{Right}$$

```
data SpanObjName : Set₀ where Left Centre Right : SpanObjName
data SpanMorName : SpanObjName → SpanObjName → Set₀ where
  left : SpanMorName Centre Left
  right : SpanMorName Centre Right
spanLES : LocalSetoid₀ SpanObjName
spanLES X Y = ≡-setoid (SpanMorName X Y)
cospanLES : LocalSetoid₀ SpanObjName
cospanLES X Y = ≡-setoid (SpanMorName Y X)
```

3.7.2 Shape Graph for Unlabelled Graphs

For plain, unlabelled graphs, we use:

```
data GraphObjName : Set₀ where N E : GraphObjName
data GraphMorName : GraphObjName → GraphObjName → Set₀ where
  src : GraphMorName E N
  trg : GraphMorName E N
graphLES : LocalSetoid₀ GraphObjName
graphLES X Y = ≡-setoid (GraphMorName X Y)
```

3.8 Categoric.Semigroupoid

For ease of working with CompOp properties in contexts other than a Semigroupoid, we also create a module CompOpProps that re-exports both CompOp itself and the properties module CompOpProps$_1$. (This is currently used to fake inheritance from Semigroupoid in OrderedSemigroupoid.)

```
module CompOpProps {i j k : Level} {Obj : Set i} {Hom : LocalSetoid Obj k j}
                   (compOp : CompOp Hom) where
  open CompOp        compOp public
  open CompOpProps₁  compOp public
```

The module SemigroupoidExports collects all semigroupoid exports except those from FinColimits and FinLimits. The original motivation for this was to ease the replacement of pieces of the latter in Categoric.Category; however, a direct **hiding** from the Semigroupoid re-export also works.

```
module SemigroupoidExports {i j k : Level} {Obj : Set i} {Hom : LocalSetoid Obj k j}
  (compOp : CompOp Hom) where
  open LocalHomSetoid Hom    public
  open LocalSetoidCalc Hom   public
  open CompOpProps compOp public
```

A semigroupoid consists of a LocalHomSetoid together with a CompOp over it, and exports all the derived material presented up to now for these two components.

```
record Semigroupoid {i : Level} (j k : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k)) where
  field
    Hom : LocalSetoid Obj j k
    compOp : CompOp Hom
  open SemigroupoidExports compOp public

module _ {i j k : Level} {Obj : Set i} (Base : Semigroupoid j k Obj) where
  open Semigroupoid Base using (Hom)
  module HomSetoidCalc₀ = LocalSetoidCalc₀ Hom renaming (Edge₀ to Mor₀)
  module HomSetoidCalc₁ = LocalSetoidCalc₁ Hom renaming (Edge₁ to Mor₁)
  module HomSetoidCalc₂ = LocalSetoidCalc₂ Hom renaming (Edge₂ to Mor₂)
  module HomSetoidCalc₃ = LocalSetoidCalc₃ Hom renaming (Edge₃ to Mor₃)
  module HomSetoidCalc₄ = LocalSetoidCalc₄ Hom renaming (Edge₄ to Mor₄)

module Semigroupoid₀ {i j k : Level} {Obj : Set i} (Base : Semigroupoid j k Obj) where
  open HomSetoidCalc₀ Base public
  open Semigroupoid Base public using () renaming
    (_⨾_ to _⨾₀_; Hom to Hom₀; compOp to compOp₀)

module Semigroupoid₁ {i j k : Level} {Obj : Set i} (Base : Semigroupoid j k Obj) where
  open HomSetoidCalc₁ Base public
  open Semigroupoid Base public using () renaming
    (_⨾_ to _⨾₁_; Hom to Hom₁; compOp to compOp₁)

module Semigroupoid₂ {i j k : Level} {Obj : Set i} (Base : Semigroupoid j k Obj) where
  open HomSetoidCalc₂ Base public
  open Semigroupoid Base public using () renaming
    (_⨾_ to _⨾₂_; Hom to Hom₂; compOp to compOp₂)

module Semigroupoid₃ {i j k : Level} {Obj : Set i} (Base : Semigroupoid j k Obj) where
  open HomSetoidCalc₃ Base public
  open Semigroupoid Base public using () renaming
    (_⨾_ to _⨾₃_; Hom to Hom₃; compOp to compOp₃)

module Semigroupoid₄ {i j k : Level} {Obj : Set i} (Base : Semigroupoid j k Obj) where
  open HomSetoidCalc₄ Base public
  open Semigroupoid Base public using () renaming
    (_⨾_ to _⨾₄_; Hom to Hom₄; compOp to compOp₄)
```

Freyd and Scedrov (1990, 1.243) call retractSemigroupoid F B "founded on B".

```
retractSemigroupoid : {i₁ i₂ j k : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                    → (F : Obj₂ → Obj₁) → Semigroupoid j k Obj₁ → Semigroupoid j k Obj₂
retractSemigroupoid F sg = let open Semigroupoid sg in record
  {Hom    = retractLES F Hom
  ;compOp = retractCompOp F compOp
  }

module _ {i j k : Level} {Obj : Set i} (Base : Semigroupoid j k Obj) where
  open Semigroupoid Base
  retract²Semigroupoid : {i₂ j₂ : Level} {Obj₂ : Set i₂} {Mor₂ : Obj₂ → Obj₂ → Set j₂}
                       → (_⨾₂_ : {A B C : Obj₂} → Mor₂ A B → Mor₂ B C → Mor₂ A C)
                       → (FO : Obj₂ → Obj)
                       → (FM : {A B : Obj₂} → Mor₂ A B → Mor (FO A) (FO B))
```

```
                          → (FM-⨾₂ : {A B C : Obj₂} {f : Mor₂ A B} {g : Mor₂ B C}
                              → FM (f ⨾₂ g) ≈ FM f ⨾ FM g)
                          → Semigroupoid j₂ k Obj₂
  retract²Semigroupoid _⨾₂_ FO FM FM-⨾₂ = record
    {Hom = retract²LES Hom FO FM
    ;compOp = retract²CompOp _⨾₂_ FO FM FM-⨾₂
    }
```

```
attachSemigroupoid : {i j k : Level} {Obj : Set i} → Semigroupoid j k Obj → Semigroupoid (i ⊍ j) k Obj
attachSemigroupoid sg = let open Semigroupoid sg in record
  {Hom    = attachLES Hom
  ;compOp = attachCompOp compOp
  }
```

```
oppositeSemigroupoid : {i j k : Level} {Obj : Set i} → Semigroupoid j k Obj → Semigroupoid j k Obj
oppositeSemigroupoid sg = let open Semigroupoid sg in record
  {Hom    = λ A B → Hom B A
  ;compOp = oppositeCompOp compOp
  }
```

3.9 Categoric.Span

We define spans in a separate module MkSpan parameterised over a LocalSetoid, so that we can easily obtain co-spans via dualisation.

3.9.1 Spans

Together with the Span datatype we define also a number of utility functions.

```
module MkSpan {i j k : Level} {Obj : Set i} (Hom : LocalSetoid Obj j k)
  where
  open LocalEdgeSetoid Hom using (_≈_) renaming (Edge to Mor)
  record Span (A B C : Obj) : Set j where
    field left  : Mor A B
          right : Mor A C
  _\≈/_ : {A B C : Obj} → Span A B C → Span A B C → Set k
  s \≈/ t = Span.left s ≈ Span.left t × Span.right s ≈ Span.right t
  mkSpan : {A B C : Obj} → Mor A B → Mor A C → Span A B C
  mkSpan F G = record {left = F; right = G}
  currySpan : {ℓ : Level} {S : Set ℓ} {A B C : Obj} → (Span A B C → S) → Mor A B → Mor A C → S
  currySpan fun F G = fun (mkSpan F G)
  uncurrySpan : {ℓ : Level} {S : Set ℓ} {A B C : Obj} → (Mor A B → Mor A C → S) → Span A B C → S
  uncurrySpan fun FG = fun (Span.left FG) (Span.right FG)
```

3.9.2 Adding Co-Spans by Dualisation of Spans

```
module MkCospan {i j k : Level} {Obj : Set i} (Hom : LocalSetoid Obj j k)
   = MkSpan (λ A B → Hom B A)
    renaming (Span to Cospan; module Span to Cospan; _\≈/_ to _/≈\_
             ; mkSpan to mkCospan; currySpan to curryCospan; uncurrySpan to uncurryCospan)
```

3.10 Categoric.Semigroupoid.Span

```
module Categoric.Semigroupoid.Span {i j k : Level} {Obj : Set i}
                                   (SG : Semigroupoid j k Obj) where
  open Semigroupoid SG using (Hom)
  open MkSpan    Hom public
  open MkCospan Hom public
```

3.11 Categoric.Semigroupoid.SGIso

We provide a definition of isomorphisms in the semigroupoid setting, where identities are not assumed.

```
module Categoric.Semigroupoid.SGIso {i j k : Level} {Obj : Set i} {Hom : LocalSetoid Obj k j}
                                    (compOp : CompOp Hom) where
  open LocalHomSetoid Hom
  open LocalSetoidCalc Hom
  open CompOp compOp
  open CompOpProps₁ compOp
```

When discussing inverses in the semigroupoid context, we need to use isIdentity, since we do not assume that each object has an identity *a priori*. Since we encounter situations, where the same definitions that give rise to isomorphisms in categories only give rise to one-sided identites in semigroupoids, as for example in Categoric.FinColimits.Initial (Sect. 4.1), we also provide definitions for such "inverses yielding only one-sided identities".

```
  record _hasSGInverseL_ {A B : Obj} (F : Mor A B) (G : Mor B A) : Set (i ⊍ j ⊍ k) where
    field
      rightSGInverseL : isLeftIdentity (F ⨾ G)
      leftSGInverseL  : isLeftIdentity (G ⨾ F)
  record _hasSGInverseR_ {A B : Obj} (F : Mor A B) (G : Mor B A) : Set (i ⊍ j ⊍ k) where
    field
      rightSGInverseR : isRightIdentity (F ⨾ G)
      leftSGInverseR  : isRightIdentity (G ⨾ F)
  record _hasSGInverse_ {A B : Obj} (F : Mor A B) (G : Mor B A) : Set (i ⊍ j ⊍ k) where
    field
      rightSGInverse : isIdentity (F ⨾ G)
      leftSGInverse  : isIdentity (G ⨾ F)
    rightSGInverseL  : isLeftIdentity (F ⨾ G)
    rightSGInverseL  = proj₁ rightSGInverse
    rightSGInverseR  : isRightIdentity (F ⨾ G)
    rightSGInverseR  = proj₂ rightSGInverse
    leftSGInverseL   : isLeftIdentity (G ⨾ F)
    leftSGInverseL   = proj₁ leftSGInverse
    leftSGInverseR   : isRightIdentity (G ⨾ F)
    leftSGInverseR   = proj₂ leftSGInverse
```

For an isomorphism, an inverse morphism exists such that isIdentity holds for both compositions:

```
  record SGIso (A B : Obj) : Set (i ⊍ j ⊍ k) where
    field
      mor : Mor A B
      inv  : Mor B A
      prf : mor hasSGInverse inv
    open _hasSGInverse_ prf public
  record SGIsoL (A B : Obj) : Set (i ⊍ j ⊍ k) where
    field
```

```
      mor : Mor A B
      inv  : Mor B A
      prf : mor hasSGInverseL inv
    open _hasSGInverseL_ prf public
  record SGIsoR (A B : Obj) : Set (i ⊍ j ⊍ k) where
    field
      mor : Mor A B
      inv  : Mor B A
      prf : mor hasSGInverseR inv
    open _hasSGInverseR_ prf public
```

3.12 Categoric.Semigroupoid.Factoring

We now explore different approach to isomorphisms in semigroupoids via factoring properties.

```
module Categoric.Semigroupoid.Factoring {i j k : Level} {Obj : Set i} {Hom : LocalSetoid Obj k j}
  (compOp : CompOp Hom) where
  open LocalHomSetoid Hom
  open LocalSetoidCalc Hom
  open CompOp compOp
  open CompOpProps₁ compOp
  open import Categoric.Semigroupoid.SGIso compOp
```

For the usual characterisation of isomorphisms, identities are required; here we provide definitions that can be used as an alternative to SGIso.

- F : Mor A B is called *left-factoring* iff isLeftFactoring F, which documents a meta-level isomorphism between Mor A Z and Mor B Z for each object Z, and
- F : Mor A B is called *right-factoring* iff isRightFactoring F, which documents a meta-level isomorphism between Mor Z A and Mor Z B for each object Z.

```
  isLeftFactoring : {A B : Obj} → Mor A B → Set (i ⊍ j ⊍ k)
  isLeftFactoring {A} {B} F = {Z : Obj} (S : Mor A Z) → ∃! _≈_ (λ R → S ≈ F ⨾ R)
  LeftFactoring-isEpi : {A B : Obj} {F : Mor A B} → isLeftFactoring F → isEpi F
  LeftFactoring-isEpi {A} {B} {F} left {Z} {R} {S} F⨾R≈F⨾S = ≈-begin
      R
    ≈˘⟨ P-unique ≈-refl ⟩
      P
    ≈⟨ P-unique F⨾R≈F⨾S ⟩
      S
    □
    where
      P : Mor B Z
      P = proj₁ (left (F ⨾ R))
      P-unique :  {U : Mor B Z} → F ⨾ R ≈ F ⨾ U → P ≈ U
      P-unique = proj₂ (proj₂ (left (F ⨾ R)))
  isRightFactoring : {A B : Obj} → Mor A B → Set (i ⊍ j ⊍ k)
  isRightFactoring {A} {B} F = {Z : Obj} (S : Mor Z B) → ∃! _≈_ (λ P → S ≈ P ⨾ F)
  RightFactoring-isMono : {A B : Obj} {F : Mor A B} → isRightFactoring F → isMono F
  RightFactoring-isMono {A} {B} {F} right {Z} {R} {S} R⨾F≈S⨾F = ≈-begin
      R
    ≈˘⟨ P-unique ≈-refl ⟩
      P
    ≈⟨ P-unique R⨾F≈S⨾F ⟩
      S
    □
```

```
    where
      P : Mor Z A
      P = proj₁ (right (R ⨾ F))
      P-unique : {U : Mor Z A} → R ⨾ F ≈ U ⨾ F → P ≈ U
      P-unique = proj₂ (proj₂ (right (R ⨾ F)))
```

Morphisms that are both left- and right-factoring are isos, and their source and target objects have identities (for the time being, we leave this development in the shape of an exploration; it will be properly packaged if actual uses for it arise):

```
module IsFactoring {A B : Obj} {F : Mor A B} (left : isLeftFactoring F) (right : isRightFactoring F)
  where
  F-isMono : isMono F
  F-isMono = RightFactoring-isMono right

  F-isEpi : isEpi F
  F-isEpi = LeftFactoring-isEpi left

  iB : Mor B B
  iB = proj₁ (left F)
  F≈F⨾iB : F ≈ F ⨾ iB
  F≈F⨾iB = proj₁ (proj₂ (left F))
  iB-unique : {U : Mor B B} → F ≈ F ⨾ U → iB ≈ U
  iB-unique = proj₂ (proj₂ (left F))

  iA : Mor A A
  iA = proj₁ (right F)
  F≈iA⨾F : F ≈ iA ⨾ F
  F≈iA⨾F = proj₁ (proj₂ (right F))
  iA-unique : {U : Mor A A} → F ≈ U ⨾ F → iA ≈ U
  iA-unique = proj₂ (proj₂ (right F))

  G : Mor B A
  G = proj₁ (left iA)
  iA≈F⨾G : iA ≈ F ⨾ G
  iA≈F⨾G = proj₁ (proj₂ (left iA))
  G-unique : {U : Mor B A} → iA ≈ F ⨾ U → G ≈ U
  G-unique = proj₂ (proj₂ (left iA))

  H : Mor B A
  H = proj₁ (right iB)
  iB≈H⨾F : iB ≈ H ⨾ F
  iB≈H⨾F = proj₁ (proj₂ (right iB))

  G≈H : G ≈ H
  G≈H = G-unique (F-isMono (≈-begin
        iA ⨾ F
      ≈˘⟨ F≈iA⨾F ⟩
        F
      ≈⟨ F≈F⨾iB ⟩
        F ⨾ iB
      ≈⟨ ⨾-cong₂ iB≈H⨾F ⟨≈≈⟩ ⨾-assocL ⟩
        (F ⨾ H) ⨾ F
      □))

  iB≈G⨾F : iB ≈ G ⨾ F
  iB≈G⨾F = ≈-begin
        iB
      ≈⟨ iB≈H⨾F ⟩
        H ⨾ F
      ≈˘⟨ ⨾-cong₁ G≈H ⟩
        G ⨾ F
      □

  iA-leftId : isLeftIdentity iA
```

```
iA-leftId {C} {R} = ≈-begin
    iA ⨾ R
  ≈⟨ ⨾-cong₂ R≈F⨾S ⟩
    iA ⨾ F ⨾ S
  ≈˘⟨ ⨾-cong₁ F≈iA⨾F ⟨≈≈⟩ ⨾-assoc ⟩
    F ⨾ S
  ≈˘⟨ R≈F⨾S ⟩
    R
  □
  where
    S : Mor B C
    S = proj₁ (left R)
    R≈F⨾S : R ≈ F ⨾ S
    R≈F⨾S = proj₁ (proj₂ (left R))
iA-rightId : isRightIdentity iA
iA-rightId {C} {R} = F-isMono (≈-begin
    (R ⨾ iA) ⨾ F
  ≈˘⟨ ⨾-cong₂ F≈iA⨾F ⟨≈≈⟩ ⨾-assocL ⟩
    R ⨾ F
  □)
iB-leftId : isLeftIdentity iB
iB-leftId {C} {R} = F-isEpi (≈-begin
    F ⨾ iB ⨾ R
  ≈˘⟨ ⨾-cong₁ F≈F⨾iB ⟨≈≈⟩ ⨾-assoc ⟩
    F ⨾ R
  □)
iB-rightId : isRightIdentity iB
iB-rightId {C} {R} = ≈-begin
    R ⨾ iB
  ≈⟨ ⨾-cong₁ R≈S⨾F ⟨≈≈⟩ ⨾-assoc ⟩
    S ⨾ F ⨾ iB
  ≈˘⟨ ⨾-cong₂ F≈F⨾iB ⟩
    S ⨾ F
  ≈˘⟨ R≈S⨾F ⟩
    R
  □
  where
    S : Mor C A
    S = proj₁ (right R)
    R≈S⨾F : R ≈ S ⨾ F
    R≈S⨾F = proj₁ (proj₂ (right R))
FinvG : F hasSGInverse G
FinvG = record
  {rightSGInverse = isIdentity-subst iA≈F⨾G (iA-leftId, iA-rightId)
  ; leftSGInverse  = isIdentity-subst iB≈G⨾F (iB-leftId, iB-rightId)
  }
F-SGIso : SGIso A B
F-SGIso = record
  {mor = F
  ; inv = G
  ; prf = FinvG
  }
```

3.13 Categoric.IdOp

Since identity relations frequently occur in contexts where their type can be inferred, it is worth making it an implicit argument.

```
record IdOp {i j k : Level} {Obj : Set i}
            (Hom : LocalSetoid Obj j k)
            (_⨾_ : Transitive (Setoid.Carrier ○₂ Hom))
            : Set (i ⊍ j ⊍ k) where
  open LocalHomSetoid Hom
  open LocalSetoidCalc Hom
  field
    Id : {A : Obj} → Mor A A
    leftId   : {A B : Obj} → {f : Mor A B} → (Id ⨾ f) ≈ f
    rightId  : {A B : Obj} → {f : Mor A B} → (f ⨾ Id) ≈ f
```

In contexts where propositional equality reasoning on objects is necessary, identities can change their type:

```
  ≡-substSrc-Id : {A B : Obj} (A≡B : A ≡ B)
                → ≡-substSrc A≡B (Id {A}) ≡ ≡-substTrg (≡-sym A≡B) (Id {B})
  ≡-substSrc-Id ≡-refl = ≡-refl
  ≡-substTrg-Id : {A B : Obj} (A≡B : A ≡ B)
                → ≡-substTrg A≡B (Id {A}) ≡ ≡-substSrc (≡-sym A≡B) (Id {B})
  ≡-substTrg-Id ≡-refl = ≡-refl
  Id-≡-subst-Trg-src : {A B C : Obj} (A≡B : A ≡ B) (A≡C : A ≡ C)
                     → ≡-substTrg A≡C (≡-substSrc A≡B (Id {A}))
                     ≡ ≡-substTrg (≡-trans (≡-sym A≡B) A≡C) (Id {B})
  Id-≡-subst-Trg-src ≡-refl ≡-refl = ≡-refl
  Id-≡-subst-trg-Src : {A B C : Obj} (A≡B : A ≡ B) (A≡C : A ≡ C)
                     → ≡-substTrg A≡C (≡-substSrc A≡B (Id {A}))
                     ≡ ≡-substSrc (≡-trans (≡-sym A≡C) A≡B) (Id {C})
  Id-≡-subst-trg-Src ≡-refl ≡-refl = ≡-refl
  Id-≡-subst-Trg≡Src : {A B C : Obj} (A≡B : A ≡ B) (A≡B′ : A ≡ B)
                     → ≡-substTrg A≡B′ (≡-substSrc A≡B (Id {A})) ≡ Id {B}
  Id-≡-subst-Trg≡Src ≡-refl ≡-refl = ≡-refl
  Id-≡-subst-Src-trg : {A B C : Obj} (A≡B : A ≡ B) (A≡C : A ≡ C)
                     → ≡-substSrc A≡B (≡-substTrg A≡C (Id {A}))
                     ≡ ≡-substSrc (≡-trans (≡-sym A≡C) A≡B) (Id {C})
  Id-≡-subst-Src-trg ≡-refl ≡-refl = ≡-refl
  Id-≡-subst-src-Trg : {A B C : Obj} (A≡B : A ≡ B) (A≡C : A ≡ C)
                     → ≡-substSrc A≡B (≡-substTrg A≡C (Id {A}))
                     ≡ ≡-substTrg (≡-trans (≡-sym A≡B) A≡C) (Id {B})
  Id-≡-subst-src-Trg ≡-refl ≡-refl = ≡-refl
  Id-≡-subst-Src≡Trg : {A B : Obj} (A≡B : A ≡ B) (A≡B′ : A ≡ B)
                     → ≡-substSrc A≡B (≡-substTrg A≡B′ (Id {A})) ≡ Id {B}
  Id-≡-subst-Src≡Trg ≡-refl ≡-refl = ≡-refl
```

```
retractIdOp : {i₁ i₂ j k : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
            → (F : Obj₂ → Obj₁)
            → {Hom : LocalSetoid Obj₁ j k}
              {_⨾_ : Transitive (Setoid.Carrier ○₂ Hom)}
            → IdOp Hom _⨾_ → IdOp (retractLES F Hom) _⨾_
retractIdOp F idOp = let open IdOp idOp in record
  {Id      = Id
  ; leftId  = leftId
  ; rightId = rightId
  }
```

```
module _ {i j k : Level} {Obj : Set i} (Base : Semigroupoid j k Obj) where
    open Semigroupoid Base
    retract²IdOp : (idOp : IdOp Hom _⨾_)
                → {i₂ j₂ : Level} {Obj₂ : Set i₂} {Mor₂ : Obj₂ → Obj₂ → Set j₂}
                → (Id₂ : {A : Obj₂} → Mor₂ A A)
                → (_⨾₂_ : {A B C : Obj₂} → Mor₂ A B → Mor₂ B C → Mor₂ A C)
                → (FO : Obj₂ → Obj)
                → (FM : {A B : Obj₂} → Mor₂ A B → Mor (FO A) (FO B))
                → (FM-Id₂ : {A : Obj₂} → FM Id₂ ≈ IdOp.Id idOp {FO A})
                → (FM-⨾₂ : {A B C : Obj₂} {f : Mor₂ A B} {g : Mor₂ B C}
                      → FM (f ⨾₂ g) ≈ FM f ⨾ FM g)
                → IdOp (retract²LES Hom FO FM) _⨾₂_
    retract²IdOp idOp Id₂ _⨾₂_ FO FM FM-Id₂ FM-⨾₂ = let open IdOp idOp in record
      {Id = Id₂
      ; leftId = λ {A B f} → ≈-begin
            FM (Id₂ ⨾₂ f)
        ≈⟨ FM-⨾₂ ⟩
            FM Id₂ ⨾ FM f
        ≈⟨ ⨾-cong₁ FM-Id₂ ⟨≈≈⟩ leftId ⟩
            FM f
        □
      ; rightId = λ {A B f} → ≈-begin
            FM (f ⨾₂ Id₂)
        ≈⟨ FM-⨾₂ ⟩
            FM f ⨾ FM Id₂
        ≈⟨ ⨾-cong₂ FM-Id₂ ⟨≈≈⟩ rightId ⟩
            FM f
        □
      }

attachIdOp : {i j k : Level} {Obj : Set i}
             {Hom : LocalSetoid Obj j k}
             {_⨾_ : Transitive (Setoid.Carrier ∘₂ Hom)}
           → IdOp Hom _⨾_ → IdOp (attachLES Hom) (Attach-⨾ Hom _⨾_)
attachIdOp idOp = let open IdOp idOp in record
  {Id      = λ {A} → ATTACH A A Id
  ; leftId  = leftId
  ; rightId = rightId
  }

oppositeIdOp : {i j k : Level} {Obj : Set i}
               {Hom : LocalSetoid Obj j k}
               {_⨾_ : Transitive (Setoid.Carrier ∘₂ Hom)}
             → IdOp Hom _⨾_ → IdOp (λ A B → Hom B A) (λ f g → g ⨾ f)
oppositeIdOp idOp = let open IdOp idOp in record
  {Id      = Id
  ; leftId  = rightId
  ; rightId = leftId
  }
```

Frequently it will be more convenient to base an IdOp directly on a semigroupoid instead of on Hom and _⨾_:

```
SGIdOp : {i j k : Level} {Obj : Set i} (SG : Semigroupoid j k Obj) → Set (i ⊍ j ⊍ k)
SGIdOp SG = let open Semigroupoid SG in IdOp Hom _⨾_
```

Given all the constituents of a category together, we can derive the following additional properties:

```
module CategoryProps {i j k : Level} {Obj : Set i}
    (SG : Semigroupoid j k Obj)
```

```
    (idOp : SGIdOp SG)
  where
    open Semigroupoid SG
    open IdOp         idOp
    ≈Id-isLeftIdentity   : {A : Obj} {i : Mor A A} → (i ≈ Id)
                          → {B : Obj} → {f : Mor A B} → (i ⨾ f) ≈ f
    ≈Id-isLeftIdentity   i≈Id = ≈-trans (⨾-cong₁ i≈Id) leftId
    ≈Id-isRightIdentity  : {B : Obj} {i : Mor B B} → (i ≈ Id)
                          → {A : Obj} → {f : Mor A B} → (f ⨾ i) ≈ f
    ≈Id-isRightIdentity  i≈Id = ≈-trans (⨾-cong₂ i≈Id) rightId
    ≈Id-isIdentity   : {B : Obj} {i : Mor B B} → (i ≈ Id) → isIdentity i
    ≈Id-isIdentity i≈Id   = (λ {B} {f} → ≈Id-isLeftIdentity i≈Id)
                          , (λ {A} {F} → ≈Id-isRightIdentity i≈Id)
    Id-isIdentity : {A : Obj} → isIdentity (Id {A})
    Id-isIdentity = ≈Id-isIdentity ≈-refl
    isLeftIdentity-≈Id : {B : Obj} {i : Mor B B} → isLeftIdentity i → i ≈ Id
    isLeftIdentity-≈Id {B} {i} left = ≈-begin
        i
      ≈˘⟨ rightId ⟩
        i ⨾ Id
      ≈⟨ left ⟩
        Id
      □
    isRightIdentity-≈Id  : {B : Obj} {i : Mor B B} → isRightIdentity i → i ≈ Id
    isRightIdentity-≈Id {B} {i} right = ≈-begin
        i
      ≈˘⟨ leftId ⟩
        Id ⨾ i
      ≈⟨ right ⟩
        Id
      □
    isIdentity-≈Id   : {B : Obj} {i : Mor B B} → isIdentity i → i ≈ Id
    isIdentity-≈Id {B} {i} (left, right) = isLeftIdentity-≈Id left


    Id-isMono : {A : Obj} → isMono (Id {A})
    Id-isMono {A} {Z} {R} {S} R⨾Id≈S⨾Id = rightId ⟨≈˘≈⟩ R⨾Id≈S⨾Id ⟨≈≈⟩ rightId
    Id-isEpi : {A : Obj} → isEpi (Id {A})
    Id-isEpi {A} {Z} {R} {S} Id⨾R≈Id⨾S = leftId ⟨≈˘≈⟩ Id⨾R≈Id⨾S ⟨≈≈⟩ leftId


infix 1 _hasRightInverse_ _hasLeftInverse_ _hasInverse_
_hasRightInverse_ : {A B : Obj} (f : Mor A B) (g : Mor B A) → Set k
f hasRightInverse g = f ⨾ g ≈ Id
_hasLeftInverse_ : {A B : Obj} (f : Mor A B) (g : Mor B A) → Set k
f hasLeftInverse g = g ⨾ f ≈ Id
_hasInverse_ : {A B : Obj} (f : Mor A B) (g : Mor B A) → Set k
f hasInverse g = (f hasLeftInverse g) × (f hasRightInverse g)


hasInverse-cong : {A B : Obj} {f g : Mor A B} {f⁻¹ g⁻¹ : Mor B A}
                → f hasLeftInverse f⁻¹
                → g hasRightInverse g⁻¹
                → f ≈ g → f⁻¹ ≈ g⁻¹
hasInverse-cong {f = f} {g} {f⁻¹} {g⁻¹} fLInv gRInv f≈g = ≈-begin
    f⁻¹
  ≈˘⟨ ⨾-cong₂ gRInv ⟨≈≈⟩ rightId ⟩
    f⁻¹ ⨾ g ⨾ g⁻¹
```

```
  ≈˘⟨ ⨾-cong₂₁ f≈g ⟩
    f⁻¹ ⨾ f ⨾ g⁻¹
  ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ fLInv ⟨≈≈⟩ leftId ⟩
    g⁻¹
  □

pairedToIdMor³ : {A B C D : Obj}
  {f : Mor A B} {g : Mor B C} {h : Mor C D}
  {f⁻¹ : Mor B A} {g⁻¹ : Mor C B} {h⁻¹ : Mor D C}
  → (f ⨾ f⁻¹ ≈ Id) → (g ⨾ g⁻¹ ≈ Id) → (h ⨾ h⁻¹ ≈ Id)
  → (f ⨾ g ⨾ h) ⨾ (h⁻¹ ⨾ g⁻¹ ⨾ f⁻¹) ≈ Id
pairedToIdMor³ {f = f} {g} {h} {f⁻¹} {g⁻¹} {h⁻¹} ff⁻¹≈Id gg⁻¹≈Id hh⁻¹≈Id = ≈-begin
    (f ⨾ g ⨾ h) ⨾ (h⁻¹ ⨾ g⁻¹ ⨾ f⁻¹)
  ≈⟨ ≈-refl ⟩
    (f ⨾ (g ⨾ h)) ⨾ (h⁻¹ ⨾ (g⁻¹ ⨾ f⁻¹))
  ≈⟨ ⨾-cong₁ ⨾-assocL ⟨≈≈⟩ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⨾-assocL ⟨≈≈⟩ ⨾-cong₂₁ hh⁻¹≈Id ⟨≈≈⟩ ⨾-cong₂ leftId ⟩
    (f ⨾ g) ⨾ (g⁻¹ ⨾ f⁻¹)
  ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⨾-assocL ⟨≈≈⟩ ⨾-cong₂₁ gg⁻¹≈Id ⟨≈≈⟩ ⨾-cong₂ leftId ⟩
    f ⨾ f⁻¹
  ≈⟨ ff⁻¹≈Id ⟩
    Id
  □

    record IsIso {A B : Obj} (f : Mor A B) : Set (j ⊍ k) where
      field
        _⁻¹ : Mor B A
        rightInverse : f ⨾ _⁻¹ ≈ Id
        leftInverse : _⁻¹ ⨾ f ≈ Id
    record Iso (A B : Obj) : Set (j ⊍ k) where
      field
        isoMor : Mor A B
        isIso : IsIso isoMor
      open IsIso isIso public
    open Iso public
    Id-isIso : {A : Obj} → IsIso (Id {A})
    Id-isIso = record
      { _⁻¹ = Id
      ; rightInverse = leftId
      ; leftInverse = rightId
      }
    IdIso : {A : Obj} → Iso A A
    IdIso {A} = record {isoMor = Id; isIso = Id-isIso}
    invIso : {A B : Obj} → Iso A B → Iso B A
    invIso f = record
      {isoMor = f ⁻¹
      ; isIso = record
        {_⁻¹ = isoMor f
        ; rightInverse = leftInverse f
        ; leftInverse = rightInverse f
        }
      }
    _⨾Iso_ : {A B C : Obj} → Iso A B → Iso B C → Iso A C
    _⨾Iso_ f g = record
      {isoMor = isoMor f ⨾ isoMor g
      ; isIso = record
        {_⁻¹ = g ⁻¹ ⨾ f ⁻¹
```

```
      ; rightInverse = ≈-begin
              (isMor f ⨾ isoMor g) ⨾ (g ⁻¹ ⨾ f ⁻¹)
          ≈⟨ ⨾-cong₁₂&₂₁ (rightInverse g) ⟩
              isoMor f ⨾ Id ⨾ f ⁻¹
          ≈⟨ ⨾-cong₂ leftId ⟨≈≈⟩ rightInverse f ⟩
              Id
          □
      ; leftInverse = ≈-begin
              (g ⁻¹ ⨾ f ⁻¹) ⨾ (isoMor f ⨾ isoMor g)
          ≈⟨ ⨾-cong₁₂&₂₁ (leftInverse f) ⟩
              g ⁻¹ ⨾ Id ⨾ isoMor g
          ≈⟨ ⨾-cong₂ leftId ⟨≈≈⟩ leftInverse g ⟩
              Id
          □
      }
    }
  _≈Iso_ : {A B : Obj} → Iso A B → Iso A B → Set k
  f ≈Iso g = isoMor f ≈ isoMor g
  Iso≈ : (A B : Obj) → Setoid (j ⊍ k) k
  Iso≈ A B = record
    {Carrier = Iso A B
    ; _≈_ = _≈Iso_
    ; isEquivalence = record {refl = ≈-refl; sym = ≈-sym; trans = ≈-trans}
    }
  ⨾Iso-cong    : {A B C : Obj} {F₁ F₂ : Iso A B} {G₁ G₂ : Iso B C}
               → F₁ ≈Iso F₂ → G₁ ≈Iso G₂ → (F₁ ⨾Iso G₁) ≈Iso (F₂ ⨾Iso G₂)
  ⨾Iso-cong F₁≈F₂ G₁≈G₂ = ⨾-cong F₁≈F₂ G₁≈G₂
  IsoCompOp : CompOp Iso≈
  IsoCompOp = record
    {_⨾_ = _⨾Iso_
    ; ⨾-cong = λ {A} {B} {C} {F₁} {F₂} {G₁} {G₂} → ⨾Iso-cong {F₁ = F₁} {F₂} {G₁} {G₂}
    ; ⨾-assoc = ⨾-assoc
    }
  IsoIdOp : IdOp Iso≈ _⨾Iso_
  IsoIdOp = record
    {Id = λ {A} → record
      {isoMor = Id
      ; isIso = record {_⁻¹ = Id; rightInverse = leftId; leftInverse = leftId}
      }
    ; leftId = leftId
    ; rightId = rightId
    }
```

Existence of isomorphisms induces a setoid of objects:

```
  IsoSetoid : Setoid i (j ⊍ k)
  IsoSetoid = record
    {Carrier = Obj; _≈_ = Iso
    ; isEquivalence = record {refl = IdIso; sym = invIso; trans = _⨾Iso_}
    }


  isMono-isoMor : {A B : Obj} (I : Iso A B) → isMono (isoMor I)
  isMono-isoMor I {Z} {R} {S} R⨾I≈S⨾I = ≈-begin
        R
    ≈⟨ rightId ⟨≈˘≈˘⟩ ⨾-cong₂ (rightInverse I) ⟩
        R ⨾ isoMor I ⨾ I ⁻¹
    ≈⟨ ⨾-cong₁&₂₁ R⨾I≈S⨾I ⟩
```

```
      S ⨾ isMor I ⨾ I ⁻¹
    ≈⟨ ⨾-cong₂ (rightInverse I) ⟨≈≈⟩ rightId ⟩
      S
    □
  isMono-⁻¹ : {A B : Obj} (I : Iso A B) → isMono (I ⁻¹)
  isMono-⁻¹ I {Z} {R} {S} R⨾I⁻¹≈S⨾I⁻¹ = ≈-begin
      R
    ≈⟨ rightId ⟨≈˘≈˘⟩ ⨾-cong₂ (leftInverse I) ⟩
      R ⨾ I ⁻¹ ⨾ isMor I
    ≈⟨ ⨾-cong₁&₂₁ R⨾I⁻¹≈S⨾I⁻¹ ⟩
      S ⨾ I ⁻¹ ⨾ isMor I
    ≈⟨ ⨾-cong₂ (leftInverse I) ⟨≈≈⟩ rightId ⟩
      S
    □
  isMono-⨾isoMor : {A B C : Obj} {F : Mor A B}
                 → isMono F → (I : Iso B C) → isMono (F ⨾ isoMor I)
  isMono-⨾isoMor F-isMono I = ⨾-isMono F-isMono (isMono-isoMor I)
  isMono-isoMor⨾ : {A B C : Obj} (I : Iso A B) {F : Mor B C}
                 → isMono F → isMono (isoMor I ⨾ F)
  isMono-isoMor⨾ I F-isMono = ⨾-isMono (isMono-isoMor I) F-isMono
  isMono-⨾⁻¹ : {A B C : Obj} {F : Mor A B}
             → isMono F → (I : Iso C B) → isMono (F ⨾ I ⁻¹)
  isMono-⨾⁻¹ F-isMono I = ⨾-isMono F-isMono (isMono-⁻¹ I)
  isMono-⁻¹⨾ : {A B C : Obj} (I : Iso B A) {F : Mor B C}
             → isMono F → isMono (I ⁻¹ ⨾ F)
  isMono-⁻¹⨾ I F-isMono = ⨾-isMono (isMono-⁻¹ I) F-isMono


  Iso-isEpi : {A B : Obj} (I : Iso A B) → isEpi (isoMor I)
  Iso-isEpi I {Z} {R} {S} I⨾R≈I⨾S = ≈-begin
      R
    ≈⟨ leftId ⟨≈˘≈˘⟩ ⨾-cong₁ (leftInverse I) ⟩
      (I ⁻¹ ⨾ isoMor I) ⨾ R
    ≈⟨ ⨾-cong₁₂&₂ I⨾R≈I⨾S ⟩
      (I ⁻¹ ⨾ isoMor I) ⨾ S
    ≈⟨ ⨾-cong₁ (leftInverse I) ⟨≈≈⟩ leftId ⟩
      S
    □
  isEpi-⨾Iso : {A B C : Obj} {F : Mor A B}
             → isEpi F → (I : Iso B C) → isEpi (F ⨾ isoMor I)
  isEpi-⨾Iso F-isEpi I = ⨾-isEpi F-isEpi (Iso-isEpi I)
  isEpi-Iso⨾ : {A B C : Obj} (I : Iso A B) {F : Mor B C}
             → isEpi F → isEpi (isoMor I ⨾ F)
  isEpi-Iso⨾ I F-isEpi = ⨾-isEpi (Iso-isEpi I) F-isEpi
```

An isomorphism in the opposite category is, from the point of view of Agda's type system, *not* the same as an inverse isomorphism. Therefore, we provide adaptation functions that also will be re-exported in categories.

```
module OppositeIsos {i j k : Level} {Obj : Set i}
    (SG : Semigroupoid j k Obj)
    (idOp : SGIdOp SG)
  where
  private
    module C = CategoryProps SG idOp
    module opC = CategoryProps (oppositeSemigroupoid SG) (oppositeIdOp idOp)
  open Semigroupoid SG
```

```
  open IdOp           idOp
  opIsIso : {A B : Obj} {F : Mor A B} → C.IsIso F → opC.IsIso F
  opIsIso F-isIso = let module F = C.IsIso F-isIso in record
    {_⁻¹ = F._⁻¹
    ; rightInverse = F.leftInverse
    ; leftInverse = F.rightInverse
    }
  unopIsIso : {A B : Obj} {F : Mor A B} → opC.IsIso F → C.IsIso F
  unopIsIso F-isIso = let module F = opC.IsIso F-isIso in record
    {_⁻¹ = F._⁻¹
    ; rightInverse = F.leftInverse
    ; leftInverse = F.rightInverse
    }
  opIso : {A B : Obj} → C.Iso A B → opC.Iso B A
  opIso J = record {isoMor = C.isoMor J; isIso = opIsIso (C.isIso J)}
  opIso⁻¹ : {A B : Obj} → C.Iso A B → opC.Iso A B
  opIso⁻¹ = opIso ∘ C.invIso
  unopIso : {A B : Obj} → opC.Iso B A → C.Iso A B
  unopIso J = record {isoMor = opC.isoMor J; isIso = unopIsIso (opC.isIso J)}
  unopIso⁻¹ : {A B : Obj} → opC.Iso A B → C.Iso A B
  unopIso⁻¹ = C.invIso ∘ unopIso
```

3.14 Categoric.Category

```
record Category {i : Level} (j k : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k)) where
  field semigroupoid : Semigroupoid j k Obj
        idOp         : SGIdOp semigroupoid
  open Semigroupoid semigroupoid public
  open IdOp         idOp         public
  open module CatProps = CategoryProps semigroupoid idOp public
  open OppositeIsos                    semigroupoid idOp public

module Category₀ {i j k : Level} {Obj : Set i} (Base : Category j k Obj) where
  open Category Base
  open Semigroupoid₀ semigroupoid public
  SG₀      = semigroupoid
  Id₀      = Id
  leftId₀  = leftId
  rightId₀ = rightId

module Category₁ {i j k : Level} {Obj : Set i} (Base : Category j k Obj) where
  open Category Base
  open Semigroupoid₁ semigroupoid public
  SG₁      = semigroupoid
  Id₁      = Id
  leftId₁  = leftId
  rightId₁ = rightId

module Category₂ {i j k : Level} {Obj : Set i} (Base : Category j k Obj) where
  open Category Base
  open Semigroupoid₂ semigroupoid public
  SG₂      = semigroupoid
```

```
  Id₂       = Id
  leftId₂   = leftId
  rightId₂  = rightId


module Category₃ {i j k : Level} {Obj : Set i} (Base : Category j k Obj) where
  open Category Base
  open Semigroupoid₃ semigroupoid public
  SG₃       = semigroupoid
  Id₃       = Id
  leftId₃   = leftId
  rightId₃  = rightId


module Category₄ {i j k : Level} {Obj : Set i} (Base : Category j k Obj) where
  open Category Base
  open Semigroupoid₄ semigroupoid public
  SG₄       = semigroupoid
  Id₄       = Id
  leftId₄   = leftId
  rightId₄  = rightId


retractCategory : {i₁ i₂ j k : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                → (F : Obj₂ → Obj₁) → Category j k Obj₁ → Category j k Obj₂
retractCategory F cat = let open Category cat in record
  {semigroupoid = retractSemigroupoid F semigroupoid
  ; idOp         = retractIdOp F idOp
  }


module _ {i j k : Level} {Obj : Set i} (Base : Category j k Obj) where
    open Category Base
    retract²Category : {i₂ j₂ : Level} {Obj₂ : Set i₂} {Mor₂ : Obj₂ → Obj₂ → Set j₂}
                     → (Id₂ : {A : Obj₂} → Mor₂ A A)
                     → (_⨾₂_ : {A B C : Obj₂} → Mor₂ A B → Mor₂ B C → Mor₂ A C)
                     → (FO : Obj₂ → Obj)
                     → (FM : {A B : Obj₂} → Mor₂ A B → Mor (FO A) (FO B))
                     → (FM-Id₂ : {A : Obj₂} → FM Id₂ ≈ IdOp.Id idOp {FO A})
                     → (FM-⨾₂    : {A B C : Obj₂} {f : Mor₂ A B} {g : Mor₂ B C}
                            → FM (f ⨾₂ g) ≈ FM f ⨾ FM g)
                     → Category j₂ k Obj₂
    retract²Category Id₂ _⨾₂_ FO FM FM-Id₂ FM-⨾₂ = let open IdOp idOp in record
      {semigroupoid = retract²Semigroupoid semigroupoid _⨾₂_ FO FM FM-⨾₂
      ; idOp = retract²IdOp semigroupoid idOp Id₂ _⨾₂_ FO FM FM-Id₂ FM-⨾₂
      }


attachCategory : {i j k : Level} {Obj : Set i} → Category j k Obj → Category (i ⊍ j) k Obj
attachCategory cat = let open Category cat in record
  {semigroupoid = attachSemigroupoid semigroupoid
  ; idOp         = attachIdOp idOp
  }


oppositeCategory : {i j k : Level} {Obj : Set i} → Category j k Obj → Category j k Obj
oppositeCategory cat = let open Category cat in record
  {semigroupoid = oppositeSemigroupoid semigroupoid
  ; idOp         = oppositeIdOp idOp
  }
```

3.15 Categoric.ConvSemigroupoid

A converse operator in a semigroupoid needs to be involutory with respect to composition, self-inverse, and, like every operator, needs to respect the morphism equivalences.

```
record ConvOp {i j k : Level} {Obj : Set i} (SG : Semigroupoid j k Obj)
              : Set (i ⊍ j ⊍ k) where
  open Semigroupoid SG
  field
    _˘          : {A B : Obj} → Mor A B → Mor B A
  field
    ˘-cong      : {A B : Obj}   {R S : Mor A B}             → R ≈ S → R ˘ ≈ S ˘
    ˘˘          : {A B : Obj}   {R : Mor A B}               → (R ˘) ˘ ≈ R
    ˘-involution : {A B C : Obj} {R : Mor A B} {S : Mor B C} → (R ⨾ S) ˘ ≈ (S ˘ ⨾ R ˘)
  ≈-˘-swap : {A B : Obj} {R : Mor A B} {S : Mor B A} → R ≈ S ˘ → R ˘ ≈ S
  ≈-˘-swap R≈S˘ = ˘-cong R≈S˘ ⟨≈≈⟩ ˘˘
  ˘-≈-swap : {A B : Obj} {R : Mor B A} {S : Mor A B} → R ˘ ≈ S → R ≈ S ˘
  ˘-≈-swap R˘≈S = ˘˘ ⟨≈˘≈⟩ ˘-cong R˘≈S
  ˘-≈-˘ : {A B : Obj} {R S : Mor B A} → R ˘ ≈ S ˘ → R ≈ S
  ˘-≈-˘ R˘≈S˘ = ˘˘ ⟨≈˘≈⟩ ≈-˘-swap R˘≈S˘
  un˘-cong : {A B : Obj} {R S : Mor A B} → R ˘ ≈ S ˘ → R ≈ S
  un˘-cong {A} {B} {R} {S} R˘≈S˘ = ≈-begin
          R ≈⟨ ≈-sym ˘˘ ⟩        (R ˘) ˘
            ≈⟨ ˘-cong R˘≈S˘ ⟩ (S ˘) ˘
            ≈⟨ ˘˘ ⟩               S □
  ˘-coinvolution : {A B C : Obj} {R : Mor A B} {S : Mor B C} → (S ˘ ⨾ R ˘) ˘ ≈ (R ⨾ S)
  ˘-coinvolution {A} {B} {C} {R} {S} = ≈-begin
          (S ˘ ⨾ R ˘) ˘ ≈⟨ ˘-involution ⟩ (R ˘) ˘ ⨾ (S ˘) ˘
                        ≈⟨ ⨾-cong ˘˘ ˘˘ ⟩ R ⨾ S □
  ˘-involutionLeftConv : {A B C : Obj} {R : Mor B A} {S : Mor B C} → (S ˘ ⨾ R) ˘ ≈ (R ˘ ⨾ S)
  ˘-involutionLeftConv {A} {B} {C} {R} {S} = ≈-begin
          (S ˘ ⨾ R) ˘   ≈⟨ ˘-involution ⟩ R ˘ ⨾ (S ˘) ˘
                        ≈⟨ ⨾-cong₂ ˘˘ ⟩   R ˘ ⨾ S           □
  ˘-involutionRightConv : {A B C : Obj} {R : Mor A B} {S : Mor C B} → (S ⨾ R ˘) ˘ ≈ (R ⨾ S ˘)
  ˘-involutionRightConv {A} {B} {C} {R} {S} = ≈-begin
          (S ⨾ R ˘) ˘   ≈⟨ ˘-involution ⟩ (R ˘) ˘ ⨾ S ˘
                        ≈⟨ ⨾-cong₁ ˘˘ ⟩   R ⨾ S ˘ □
```

Converse allows us to define symmetry of morphisms:

```
  isSymmetric : {A : Obj} → Mor A A → Set k
  isSymmetric R = R ˘ ≈ R
```

The equality isBiDifunctional R can already be defined here, while the inclusions isDifunctional R and isCodifunctional R will be defined in Sect. 11.4.

```
  isBiDifunctional : {A B : Obj} → Mor A B → Set k
  isBiDifunctional R = R ⨾ R ˘ ⨾ R ≈ R
```

Left- and right-identities are symmetric:

```
  isLeftIdentity-isSymmetric : {A : Obj} {R : Mor A A} → isLeftIdentity R → isSymmetric R
  isLeftIdentity-isSymmetric {A} {R} left = ≈-begin
          R ˘ ≈⟨ ≈-sym left ⟩              R ⨾ R ˘
              ≈⟨ ⨾-cong₁ (≈-sym ˘˘) ⟩ (R ˘) ˘ ⨾ R ˘
              ≈⟨ ≈-sym ˘-involution ⟩  (R ⨾ R ˘) ˘
```

```
            ≈⟨ ˘-cong left ⟩           (R ˘) ˘
            ≈⟨ ˘˘ ⟩                    R
        □
isRightIdentity-isSymmetric : {A : Obj} {R : Mor A A} → isRightIdentity R → isSymmetric R
isRightIdentity-isSymmetric {A} {R} right = ≈-begin
        R ˘ ≈⟨ ≈-sym right ⟩           R ˘ ⨾ R
            ≈⟨ ⨾-cong₂ (≈-sym ˘˘) ⟩ R ˘ ⨾ (R ˘) ˘
            ≈⟨ ≈-sym ˘-involution ⟩  (R ˘ ⨾ R) ˘
            ≈⟨ ˘-cong right ⟩          (R ˘) ˘
            ≈⟨ ˘˘ ⟩                    R
        □
```

Left- and right-identities are therefore identities:

```
isLeftIdentity-isRightIdentity : {A : Obj} {R : Mor A A} → isLeftIdentity R → isRightIdentity R
isLeftIdentity-isRightIdentity {A} {R} left {B} {S} = ≈-begin
      S ⨾ R
    ≈˘⟨ ⨾-cong₂ (isLeftIdentity-isSymmetric left) ⟩
      S ⨾ R ˘
  ≈˘⟨ ˘-involutionRightConv ⟩
      (R ⨾ S ˘) ˘
  ≈⟨ ˘-cong left ⟩
      (S ˘) ˘
  ≈⟨ ˘˘ ⟩
      S
  □
isRightIdentity-isLeftIdentity : {A : Obj} {R : Mor A A} → isRightIdentity R → isLeftIdentity R
isRightIdentity-isLeftIdentity {A} {R} right {B} {S} = ≈-begin
      R ⨾ S
    ≈˘⟨ ⨾-cong₁ (isRightIdentity-isSymmetric right) ⟩
      R ˘ ⨾ S
  ≈˘⟨ ˘-involutionLeftConv ⟩
      (S ˘ ⨾ R) ˘
  ≈⟨ ˘-cong right ⟩
      (S ˘) ˘
  ≈⟨ ˘˘ ⟩
      S
  □
isLeftIdentity-isIdentity : {A : Obj} {R : Mor A A} → isLeftIdentity R → isIdentity R
isLeftIdentity-isIdentity left = left, isLeftIdentity-isRightIdentity left
isRightIdentity-isIdentity : {A : Obj} {R : Mor A A} → isRightIdentity R → isIdentity R
isRightIdentity-isIdentity right = isRightIdentity-isLeftIdentity right, right
```

Symmetric idempotents are one possible presentation of "partial equivalence relations"; for another see Sect. 11.4.

```
record IsSymIdempot {A : Obj} (R : Mor A A) : Set (i ⊍ j ⊍ k) where
    field
      symmetric   : isSymmetric R
      idempotent  : isIdempotent R
    bidifunctional : isBiDifunctional R
    bidifunctional = ≈-begin
        R ⨾ R ˘ ⨾ R
      ≈⟨ ⨾-cong₂₁ symmetric ⟩
        R ⨾ R ⨾ R
      ≈⟨ ⨾-cong₂ idempotent ⟩
        R ⨾ R
      ≈⟨ idempotent ⟩
        R
      □
```

```
record SymIdempot : Set (i ⊍ j ⊍ k) where
  field
    {obj} : Obj
    《_》 : Mor obj obj
  field
      prop : IsSymIdempot 《_》
  open IsSymIdempot prop public


record IsSymSplitting {A B : Obj} (e : Mor A A) (q : Mor A B) : Set (i ⊍ j ⊍ k) where
  field
    factors   : q ⨾ q ˘ ≈ e
    splitId   : isIdentity (q ˘ ⨾ q)
  splitting : Splitting e
  splitting = record {mor₁ = q; mor₂ = q ˘; factors = factors; splitId = splitId}
  idempotent : isIdempotent e
  idempotent = Splitting.idempotent splitting
  symmetric : isSymmetric e
  symmetric = ≈-begin
      e ˘
    ≈˘⟨ ˘-cong factors ⟩
      (q ⨾ q ˘) ˘
    ≈⟨ ˘-involutionRightConv ⟩
      q ⨾ q ˘
    ≈⟨ factors ⟩
      e
    □
  bidifunctional : isBiDifunctional q
  bidifunctional = ≈-begin
      q ⨾ q ˘ ⨾ q
    ≈⟨ proj₂ splitId ⟩
      q
    □
  leftClosed : e ⨾ q ≈ q
  leftClosed = ≈-begin
      e ⨾ q
    ≈⟨ ⨾-cong₁ factors ⟨≈˘≈⟩ ⨾-assoc ⟩
      q ⨾ q ˘ ⨾ q
    ≈⟨ bidifunctional ⟩
      q
    □
record SymSplitting {A : Obj} (e : Mor A A) : Set (i ⊍ j ⊍ k) where
  field
    {obj}     : Obj
    mor       : Mor A obj
    proof     : IsSymSplitting e mor
  open IsSymSplitting proof public
```

For symmetric idempotents, we are interested in symmetric splittings to represent quotients and subobjects:

```
record SymIdempotSplitting (SI : SymIdempot) : Set (i ⊍ j ⊍ k) where
  field
    {obj}     : Obj
    mor       : Mor (SymIdempot.obj SI) obj
    factors   : mor ⨾ mor ˘ ≈ SymIdempot.《_》 SI
    splitId   : isIdentity (mor ˘ ⨾ mor)
  splitting : Splitting (SymIdempot.《_》 SI)
  splitting = record
    {obj = obj
```

```
    ; mor₁  =  mor
    ; mor₂  =  mor ˘
    ; factors  =  factors
    ; splitId  =  splitId
    }
```

```
retractConvOp :  {i₁ i₂ j k : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
              → (F : Obj₂ → Obj₁)
              → {SG : Semigroupoid j k Obj₁} → ConvOp SG → ConvOp (retractSemigroupoid F SG)
retractConvOp F convOp  =  let open ConvOp convOp
  in record {_˘  =  _˘; ˘-cong  =  ˘-cong; ˘˘  =  ˘˘; ˘-involution  =  ˘-involution}
```

A symmetric idempotent on the retraction result is also a symmetric idempotent on the argument:

```
unretractSymIdempot :  {i₁ i₂ j k : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                    → (F : Obj₂ → Obj₁)
                    → {SG : Semigroupoid j k Obj₁} {convOp : ConvOp SG}
                    → ConvOp.SymIdempot (retractConvOp F convOp)
                    → ConvOp.SymIdempot convOp
unretractSymIdempot F {convOp = convOp} SI  =  record
  {obj   =  F obj
  ; ⟪_⟫  =  ⟪_⟫
  ; prop  =  record {symmetric  =  symmetric; idempotent  =  idempotent}
  }
  where open ConvOp.SymIdempot (retractConvOp F convOp) SI
```

```
attachConvOp :  {i j k : Level} {Obj : Set i} {SG : Semigroupoid j k Obj}
             → ConvOp SG → ConvOp (attachSemigroupoid SG)
attachConvOp convOp  =  let open ConvOp convOp in record
  {_˘             =  λ {A} {B} a → ATTACH B A ((edgeᵃ a) ˘)
  ; ˘-cong         =  ˘-cong
  ; ˘˘             =  ˘˘
  ; ˘-involution  =  ˘-involution
  }
```

The definitions of the opposite ConvOp and of semigroupoids with converse are completely straight-forward:

```
oppositeConvOp :  {i j k : Level} {Obj : Set i} {SG : Semigroupoid j k Obj}
               → ConvOp SG → ConvOp (oppositeSemigroupoid SG)
oppositeConvOp convOp  =  let open ConvOp convOp in record
  {_˘             =  _˘
  ; ˘-cong         =  ˘-cong
  ; ˘˘             =  ˘˘
  ; ˘-involution  =  ˘-involution
  }
```

```
record ConvSemigroupoid {i : Level} (j k : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k)) where
  field
    semigroupoid : Semigroupoid j k Obj
    convOp : ConvOp semigroupoid
  open Semigroupoid semigroupoid public
  open ConvOp       convOp       public
```

```
retractConvSemigroupoid :  {i₁ i₂ j k : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂} (F : Obj₂ → Obj₁)
                        → ConvSemigroupoid j k Obj₁ → ConvSemigroupoid j k Obj₂
retractConvSemigroupoid F csg  =  let open ConvSemigroupoid csg in record
```

```
  {semigroupoid = retractSemigroupoid F semigroupoid
  ; convOp       = retractConvOp F convOp
  }

attachConvSemigroupoid : {i j k : Level} {Obj : Set i}
                       → ConvSemigroupoid j k Obj → ConvSemigroupoid (i ⊍ j) k Obj
attachConvSemigroupoid csg = let open ConvSemigroupoid csg in record
  {semigroupoid = attachSemigroupoid semigroupoid
  ; convOp       = attachConvOp convOp
  }

oppositeConvSemigroupoid : {i j k : Level} {Obj : Set i}
                         → ConvSemigroupoid j k Obj → ConvSemigroupoid j k Obj
oppositeConvSemigroupoid csg = let open ConvSemigroupoid csg in record
  {semigroupoid = oppositeSemigroupoid semigroupoid
  ; convOp       = oppositeConvOp convOp
  }
```

3.16 Categoric.ConvCategory

In a category with converse, we can additionally prove that the identities are symmetric:

```
module ConvCategoryProps {i j k : Level} {Obj : Set i}
    (CSG : ConvSemigroupoid j k Obj)
    (idOp : let open ConvSemigroupoid CSG in IdOp Hom _⨾_)
  where
    open ConvSemigroupoid CSG
    open IdOp             idOp
    Id˘ : {A : Obj} → (Id {A}) ˘ ≈ Id {A}
    Id˘ {A} = isLeftIdentity-isSymmetric leftId
```

Since we build the definition of categories with converse directly on that of ConvSemigroupoids instead of on Category, we need to separately make CategoryProps available to achieve a proper theory inclusion.

```
record ConvCategory {i : Level} (j k : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k)) where
  field convSemigroupoid : ConvSemigroupoid j k Obj
  open ConvSemigroupoid convSemigroupoid
  field idOp : IdOp Hom _⨾_
  open IdOp idOp
  category : Category j k Obj
  category = record
    {semigroupoid = semigroupoid
    ; idOp = idOp
    }
  open IdOp                                       idOp public
  open CategoryProps      semigroupoid       idOp public
  open OppositeIsos       semigroupoid       idOp public
  open ConvSemigroupoid   convSemigroupoid        public
  open ConvCategoryProps convSemigroupoid idOp public
```

3.17 Categoric.Diagram

We define diagrams as LES graph homomorphisms into the underlying graph of a CompOp.

The graph LES is the shape of the diagram.

```
module Categoric.Diagram
  {i j k : Level} {Obj : Set i} {Hom : LocalSetoid Obj j k} (compOp : CompOp Hom)
  {i₀ j₀ k₀ : Level} {Node : Set i₀} (LES : LocalSetoid Node j₀ k₀)
  where
open LocalHomSetoid Hom
```

```
record Diagram : Set (i₀ ⊍ j₀ ⊍ k₀ ⊍ i ⊍ j ⊍ k) where
  constructor mkDiagram
  field unDiagram : LESHom LES Hom
  open LESHom unDiagram public renaming
    (mapN to DObj
    ; mapE to DMor
    ; congE to DMor-cong
    )
open Diagram public
```

3.18 Categoric.Diagram.Examples

```
data HouseObj : Set where T TL TR BL BR : HouseObj
data HouseEdge : Set where L R B TL TR G : HouseEdge
open Categoric.Diagram (setoidCompOp ℓ₀ ℓ₀)
house : Diagram graphLES
house = mkDiagram (record {mapN = mapN; mapE = mapE; congE = congE})
  where
    mapN : GraphObjName → Setoid₀₀
    mapN N = ≡-setoid HouseObj
    mapN E = ≡-setoid HouseEdge
    SRC : HouseEdge → HouseObj
    SRC L = TL
    SRC B = BL
    SRC R = BR
    SRC G = TR
    SRC TL = TL
    SRC TR = TR
    TRG : HouseEdge → HouseObj
    TRG L = TL
    TRG R = TR
    TRG B = BR
    TRG G = TL
    TRG TL = T
    TRG TR = T
    mapE : {x y : GraphObjName} → GraphMorName x y → mapN x ⟶ mapN y
    mapE src = →-to-⟶ SRC
    mapE trg = →-to-⟶ TRG
    congE : {X Y : GraphObjName} {e₁ e₂ : GraphMorName X Y}
      → e₁ ≡ e₂ → {x y : Setoid.Carrier (mapN X)} → Setoid._≈_ (mapN X) x y
      → Setoid._≈_ (mapN Y) (mapE e₁ ⟨$⟩ x) (mapE e₂ ⟨$⟩ y)
    congE {n₁} {n₂} {e} {.e} ≡-refl x≈y = cong (mapE e) x≈y
```

3.19 Categoric.Diagram.CompOp

```
module Categoric.Diagram.CompOp
  {i j k : Level} {Obj : Set i} {Hom : LocalSetoid Obj j k} (compOp : CompOp Hom)
    {i₀ j₀ k₀ : Level} {Node : Set i₀} (LES : LocalSetoid Node j₀ k₀)
  where
open Categoric.Diagram compOp LES
open LocalEdgeSetoid LES using (Edge)
open LocalHomSetoid Hom
open LocalSetoidCalc Hom
open CompOp compOp
open CompOpProps₁ compOp
```

Here we are building the pieces of the diagram semigroupoid from the pieces of the target semigroupoid.

```
record DiagMor D₁ D₂ : Set (i₀ ⊍ j₀ ⊍ j ⊍ k) where
  field
    transform : (n : Node) → Mor (DObj D₁ n) (DObj D₂ n)
    commute : {n m : Node} (e : Edge n m) → transform n ⨾ DMor D₂ e ≈ DMor D₁ e ⨾ transform m

DiagMorSetoid : LocalSetoid Diagram (i₀ ⊍ j₀ ⊍ j ⊍ k) (i₀ ⊍ k)
DiagMorSetoid D₁ D₂ = let open DiagMor in record
  {Carrier = DiagMor D₁ D₂
  ; _≈_ = λ Φ Ψ → (n : Node) → transform Φ n ≈ transform Ψ n
  ; isEquivalence = record
    {refl  = λ n → ≈-refl
    ; sym  = λ Φ≈Ψ n → ≈-sym (Φ≈Ψ n)
    ; trans = λ Φ≈Ψ Ψ≈Ξ n → ≈-trans (Φ≈Ψ n) (Ψ≈Ξ n)
    }
  }

DiagMorCompOp : CompOp DiagMorSetoid
DiagMorCompOp = let open Diagram using (DMor); open DiagMor in record
  {_⨾_ = λ {D₁} {D₂} {D₃} Φ Ψ → record
      {transform = λ n → transform Φ n ⨾ transform Ψ n
      ; commute = λ {n} {m} e → ≈-begin
          (transform Φ n ⨾ transform Ψ n) ⨾ DMor D₃ e
        ≈⟨ ⨾-cong₁₂&₂ (commute Ψ e) ⟩
          (transform Φ n ⨾ DMor D₂ e) ⨾ transform Ψ m
        ≈⟨ ≈-trans (⨾-cong₁ (commute Φ e)) ⨾-assoc ⟩
          DMor D₁ e ⨾ transform Φ m ⨾ transform Ψ m
        □
      }
  ; ⨾-cong = λ {D₁} {D₂} {D₃} {Φ₁} {Φ₂} {Ψ₁} {Ψ₂} Φ₁≈Φ₂ Ψ₁≈Ψ₂ n → ≈-begin
          transform Φ₁ n ⨾ transform Ψ₁ n
        ≈⟨ ⨾-cong (Φ₁≈Φ₂ n) (Ψ₁≈Ψ₂ n) ⟩
          transform Φ₂ n ⨾ transform Ψ₂ n
        □
  ; ⨾-assoc = λ n → ⨾-assoc
  }

open CompOpProps₁ DiagMorCompOp using () renaming (isMono to isMono-D)

isMono→isMono-D : {D₁ D₂ : Diagram} {Φ : DiagMor D₁ D₂}
  → ({n : Node} → isMono (DiagMor.transform Φ n)) → isMono-D Φ
isMono→isMono-D {D₁} {D₂} {Φ} Φn-isMono {Z} {R} {S} R⨾Φ≈S⨾Φ n = Φn-isMono (R⨾Φ≈S⨾Φ n)
```

```
open FinColimits using
  (Pushout; module Pushout; CoCone2Univ; module CoCone2Univ
  ; POUniversal; HasPushouts)

diagMorPushout : HasPushouts compOp → HasPushouts DiagMorCompOp
diagMorPushout pushout {A} {B} {C} FF GG = record
        {obj = mkDiagram (record
          {mapN = PN.D₀
          ; mapE = PE.eD
          ; congE = λ {n₁} {n₂} {e₁} {e₂} e₁≈e₂ → let open PE in ≈-sym (eDunique e₁
                      (eDcommute-R e₂ ⟨≈≈˘⟩ ⨾-cong₁ (DMor-cong B e₁≈e₂))
                      (eDcommute-S e₂ ⟨≈≈˘⟩ ⨾-cong₁ (DMor-cong C e₁≈e₂)))
          })
        ; left  = record {transform = PN.R; commute = PE.eDcommute-R}
        ; right = record {transform = PN.S; commute = PE.eDcommute-S}
        ; prf = record
          {commutes = PN.commutes
          ; universal = λ {Z} {P} {Q} F⨾P≈G⨾Q → let
            open PN
            open module TrU (n : Node) = CoCone2Univ compOp (PO-universal n (F⨾P≈G⨾Q n))
              using () renaming
              (univMor to trU
              ; univMor-factors-left to U-left
              ; univMor-factors-right to U-right
              )
            U = record {transform = trU
                       ; commute = λ {n₁} {n₂} e → let
                          P₁ = transform P n₁
                          Q₁ = transform Q n₁
                          P₂ = transform P n₂
                          Q₂ = transform Q n₂
                          open PE e
                          eZ = DMor Z e
                          V = trU n₁ ⨾ eZ
                          R₁⨾V≈P₁⨾eZ : R₁ ⨾ V ≈ P₁ ⨾ eZ
                          R₁⨾V≈P₁⨾eZ = ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (U-left n₁)
                          S₁⨾V≈Q₁⨾eZ : S₁ ⨾ V ≈ Q₁ ⨾ eZ
                          S₁⨾V≈Q₁⨾eZ = ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (U-right n₁)
                          V′ = eD ⨾ trU n₂
                          R₁⨾V′≈P₁⨾eZ : R₁ ⨾ V′ ≈ P₁ ⨾ eZ
                          R₁⨾V′≈P₁⨾eZ = ≈-begin
                              R₁ ⨾ eD ⨾ trU n₂
                            ≈⟨ ⨾-cong₁&₂₁ eDcommute-R ⟩
                              eB ⨾ R₂ ⨾ trU n₂
                            ≈⟨ ⨾-cong₂ (U-left n₂) ⟩
                              eB ⨾ P₂
                            ≈˘⟨ commute P e ⟩
                              P₁ ⨾ eZ
                            □
                          S₁⨾V′≈Q₁⨾eZ : S₁ ⨾ V′ ≈ Q₁ ⨾ eZ
                          S₁⨾V′≈Q₁⨾eZ = ≈-begin
                              S₁ ⨾ eD ⨾ trU n₂
                            ≈⟨ ⨾-cong₁&₂₁ eDcommute-S ⟩
                              eC ⨾ S₂ ⨾ trU n₂
                            ≈⟨ ⨾-cong₂ (U-right n₂) ⟩
                              eC ⨾ Q₂
                            ≈˘⟨ commute Q e ⟩
                              Q₁ ⨾ eZ
                            □
```

```
                          eZ-commutes : F n₁ ⨾ P₁ ⨾ eZ ≈ G n₁ ⨾ Q₁ ⨾ eZ
                          eZ-commutes = ⨾-cong₁&₂₁ (F⨾P≈G⨾Q n₁)
                          open CoCone2Univ compOp (PO.universal n₁ eZ-commutes) using () renaming
                            (univMor-unique to U′-unique)
                          V≈U′ = U′-unique R₁⨾V≈P₁⨾eZ S₁⨾V≈Q₁⨾eZ
                          V′≈U′ = U′-unique R₁⨾V′≈P₁⨾eZ S₁⨾V′≈Q₁⨾eZ
                          in V≈U′ ⟨≈≈˘⟩ V′≈U′
                        }
            in record
              {univMor = U
              ; univMor-factors-left = U-left
              ; univMor-factors-right = U-right
              ; univMor-unique = λ {V} R⨾V≈P S⨾V≈Q n →
                        CoCone2Univ.univMor-unique compOp (PO.universal n (F⨾P≈G⨾Q n))
                        {transform V n} (R⨾V≈P n) (S⨾V≈Q n)
              }
          }
        }
  where
    open Diagram
    open DiagMor
    module PN (n : Node) where
      F = transform FF n
      G = transform GG n
      PO : Pushout compOp F G
      PO = pushout F G
      module PO = Pushout compOp PO
      open PO public using (commutes) renaming
        (obj to D₀; left to R; right to S; universal to PO-universal)
    module PE {n₁ n₂ : Node} (e : Edge n₁ n₂) where
      open PN n₁ public using () renaming
        (F to F₁; G to G₁; R to R₁; S to S₁
        ; PO-universal to PO₁-universal)
      open PN n₂ public using () renaming
        (F to F₂; G to G₂; R to R₂; S to S₂; D₀ to D₂; commutes to commutes₂)
      eB = DMor B e
      eC = DMor C e
      R′ = eB ⨾ R₂
      S′ = eC ⨾ S₂
      F⨾R′≈G⨾S′ : F₁ ⨾ R′ ≈ G₁ ⨾ S′
      F⨾R′≈G⨾S′ = ≈-begin
            F₁ ⨾ DMor B e ⨾ R₂
          ≈⟨ ⨾-cong₁&₂₁ (commute FF e) ⟩
            DMor A e ⨾ F₂ ⨾ R₂
          ≈⟨ ⨾-cong₂ commutes₂ ⟩
            DMor A e ⨾ (G₂ ⨾ S₂)
          ≈˘⟨ ⨾-cong₁&₂₁ (commute GG e) ⟩
            G₁ ⨾ DMor C e ⨾ S₂
          □
      open CoCone2Univ compOp (PO₁-universal {D₂} {R′} {S′} F⨾R′≈G⨾S′) public using () renaming
        (univMor to eD
        ; univMor-factors-left to eDcommute-R     -- : R₁ ⨾ eD ≈ R′
        ; univMor-factors-right to eDcommute-S    -- : S₁ ⨾ eD ≈ S′
        ; univMor-unique to eDunique    -- : ∀ {V} → R₁ ⨾ V ≈ R′ → S₁ ⨾ V ≈ S′ → V ≈ eD
        )
```

Chapter 4

Finite Colimits and Limits

4.1 Categoric.FinColimits.Initial

```
module Categoric.FinColimits.Initial {i j k : Level} {Obj : Set i} {Hom : LocalSetoid Obj j k}
                                     (compOp : CompOp Hom) where
  open SemigroupoidCore compOp


  IsInitial : (I : Obj) → Set (i ⊍ j ⊍ k)
  IsInitial I = {A : Obj} → Σ [U : Mor I A] ((V : Mor I A) → V ≈ U)
  IsInitial≈ : {I : Obj} → IsInitial I → {A : Obj} {F G : Mor I A} → F ≈ G
  IsInitial≈ isInit {A} {F} {G} = ≈-begin
        F
      ≈⟨ proj₂ isInit F ⟩
        proj₁ isInit
      ≈˘⟨ proj₂ isInit G ⟩
        G
      □


  module IsInitial {Ⓘ : Obj} (isInitial : IsInitial Ⓘ) where
    ⓘ : {A : Obj} → Mor Ⓘ A
    ⓘ {A} = proj₁ isInitial
    ≈ⓘ : {A : Obj} {F : Mor Ⓘ A} → F ≈ ⓘ
    ≈ⓘ {A} {F} = proj₂ isInitial F
    Ⓘ≈ : {A : Obj} {F G : Mor Ⓘ A} → F ≈ G
    Ⓘ≈ {A} {F} {G} = IsInitial≈ isInitial
    open MkSpan Hom using (Span; mkSpan)
    Ⓘ-Span : {B C : Obj} → Span Ⓘ B C
    Ⓘ-Span = mkSpan ⓘ ⓘ


  module IsInitial₁ {Ⓘ : Obj} (isInitial : IsInitial Ⓘ) where
    open IsInitial isInitial public renaming (ⓘ to ⓘ₁; ≈ⓘ to ≈ⓘ₁; Ⓘ≈ to Ⓘ₁≈; Ⓘ-Span to Ⓘ₁-Span)
  module IsInitial₂ {Ⓘ : Obj} (isInitial : IsInitial Ⓘ) where
    open IsInitial isInitial public renaming (ⓘ to ⓘ₂; ≈ⓘ to ≈ⓘ₂; Ⓘ≈ to Ⓘ₂≈; Ⓘ-Span to Ⓘ₂-Span)
```

For a full SGIso in the setting of IsInitial-SGIsoL, we would need right-identities on the initial objects, which, in general, need not exist.

```
  open import Categoric.Semigroupoid.SGIso compOp using (SGIsoL; module SGIsoL)
  IsInitial-SGIsoL : {I₁ : Obj} → IsInitial I₁
                   → {I₂ : Obj} → IsInitial I₂
```

```
                        → SGIsoL I₁ I₂
IsInitial-SGIsoL {I₁} isInit₁ {I₂} isInit₂ = record
   {mor = proj₁ (isInit₁ {I₂})
   ; inv = proj₁ (isInit₂ {I₁})
   ; prf = record
      {rightSGInverseL = IsInitial≈ isInit₁
      ; leftSGInverseL  = IsInitial≈ isInit₂
      }
   }

record HasInitialObject : Set (i ⊍ j ⊍ k) where
   field
      ① : Obj
      isInitial : IsInitial ①
   open IsInitial isInitial public

module HasInitialObject₁ (hasInit : HasInitialObject) where
   open HasInitialObject hasInit public using () renaming (① to ①₁; isInitial to isInitial₁)
   open IsInitial₁ isInitial₁ public

module HasInitialObject₂ (hasInit : HasInitialObject) where
   open HasInitialObject hasInit public using () renaming (① to ①₂; isInitial to isInitial₂)
   open IsInitial₂ isInitial₂ public

IsInitial-⨾-SGIsoL : {I₁ : Obj} → IsInitial I₁
                    → {I₂ : Obj} → SGIsoL I₁ I₂
                    → IsInitial I₂
IsInitial-⨾-SGIsoL {I₁} isInit₁ {I₂} F = inv F ⨾ ⓘ, (λ V → ≈-begin
      V
   ≈˘⟨ leftSGInverseL F ⟩
      (inv F ⨾ mor F) ⨾ V
   ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ≈ⓘ ⟩
      inv F ⨾ ⓘ
   □)
   where
      open HasInitialObject (record {① = I₁; isInitial = isInit₁})
      open SGIsoL
```

Freyd and Scedrov (1990, 1.58) introduce "strict coterminators" in categories, with a property that there is equivalent (see Sect. 4.11.1) to the following:

```
IsStrictInitialSG : (I : Obj) → Set (i ⊍ j ⊍ k)
IsStrictInitialSG I = {A : Obj} → Mor A I → IsInitial A
```

4.2 Categoric.FinColimits.CoEqualiser

```
module Categoric.FinColimits.CoEqualiser {i j k : Level} {Obj : Set i} {Hom : LocalSetoid Obj j k}
                                         (compOp : CompOp Hom) where
   open SemigroupoidCore compOp

   record CoEqualiser {A B : Obj} (F G : Mor A B) : Set (i ⊍ j ⊍ k) where
      field {obj} : Obj
            mor : Mor B obj
```

```
    prop : F ⨾ mor ≈ G ⨾ mor
    universal : {Z : Obj} {R : Mor B Z} → F ⨾ R ≈ G ⨾ R
                → ∃! _≈_ (λ U → R ≈ mor ⨾ U)
```

The definition of HasCoEqualisers has been structured with direct implementability in mind: The fields do not contain any references to Categoric datastructures (here CoEqualiser). We kept Σ-types in the fields since the implementations will typically be based on functions producing results of exactly these Σ-types.

```
record HasCoEqualisers : Set (i ⊍ j ⊍ k) where
  field
    coequ : {A B : Obj} → Mor A B → Mor A B → Σ [ C : Obj ] Mor B C
  _⇈º_ : {A B : Obj} → Mor A B → Mor A B → Obj
  _⇈º_ F G = proj₁ (coequ F G)
  _⇈_ : {A B : Obj} → (F G : Mor A B) → Mor B (F ⇈º G)
  _⇈_ F G = proj₂ (coequ F G)
  field
    _⨾⇈_ : {A B : Obj} (F G : Mor A B) → F ⨾ (F ⇈ G) ≈ G ⨾ (F ⇈ G)
    ⇈-factoring : {A B : Obj} (F G : Mor A B) {C : Obj} (H : Mor B C)
                  → F ⨾ H ≈ G ⨾ H
                  → Σ [ U : Mor (F ⇈º G) C ] H ≈ (F ⇈ G) ⨾ U
  ⇈-factor : {A B : Obj} (F G : Mor A B) {C : Obj} (H : Mor B C)
           → F ⨾ H ≈ G ⨾ H → Mor (F ⇈º G) C
  ⇈-factor F G H F⨾H≈G⨾H = proj₁ (⇈-factoring F G H F⨾H≈G⨾H)
  ⇈-factors : {A B : Obj} (F G : Mor A B) {C : Obj} (H : Mor B C)
            → (F⨾H≈G⨾H : F ⨾ H ≈ G ⨾ H) → H ≈ (F ⇈ G) ⨾ ⇈-factor F G H F⨾H≈G⨾H
  ⇈-factors F G H F⨾H≈G⨾H = proj₂ (⇈-factoring F G H F⨾H≈G⨾H)
  field
    ⇈-factor-unique : {A B : Obj} (F G : Mor A B) {C : Obj} (H : Mor B C)
                      → (F⨾H≈G⨾H : F ⨾ H ≈ G ⨾ H)
                      → (V : Mor (F ⇈º G) C) → H ≈ (F ⇈ G) ⨾ V
                      → ⇈-factor F G H F⨾H≈G⨾H ≈ V
  coequaliser : {A B : Obj} → (F G : Mor A B) → CoEqualiser F G
  coequaliser F G = record
    {mor = F ⇈ G
    ;prop = F ⨾⇈ G
    ;universal = λ {C} {H} F⨾H≈G⨾H → ⇈-factor F G H F⨾H≈G⨾H
      ,⇈-factors F G H F⨾H≈G⨾H
      ,λ {V} → ⇈-factor-unique F G H F⨾H≈G⨾H V
    }
```

For a more direct implementation of "has all co-equalisers", we provide the conversion function hasCoEqualisers:

```
hasCoEqualisers : ({A B : Obj} (F G : Mor A B) → CoEqualiser F G) → HasCoEqualisers
hasCoEqualisers CoEq = record
  {coequ = λ {A} {B} F G → _, mor (CoEq F G)
  ; _⨾⇈_   = λ {A} {B} F G → prop (CoEq F G)
  ; ⇈-factoring = factoring
  ; ⇈-factor-unique   = λ {A} {B} F G {C} H F⨾H≈G⨾H V H≈F⇈G⨾V
                      → proj₂ (proj₂ (universal (CoEq F G) {C} {H} F⨾H≈G⨾H)) {V} H≈F⇈G⨾V
  }
  where
    open CoEqualiser
    factoring : {A B : Obj} (F G : Mor A B) {C : Obj} (H : Mor B C)
              → F ⨾ H ≈ G ⨾ H
              → Σ [ U : Mor (obj (CoEq F G)) C ] H ≈ mor (CoEq F G) ⨾ U
    factoring {A} {B} F G {C} H F⨾H≈G⨾H with universal (CoEq F G) {C} {H} F⨾H≈G⨾H
    ... | U, U-eq, U-unique = U, U-eq

CoEqualiser-isEpi : {A B : Obj} {F G : Mor A B}
                    → (e : CoEqualiser F G) → isEpi (CoEqualiser.mor e)
```

```
  CoEqualiser-isEpi {A} {B} {F} {G} e {Z} {R} {S} mor⨾R≈mor⨾S = let
      open CoEqualiser e
      F⨾mor⨾R≈G⨾mor⨾R : F ⨾ mor ⨾ R ≈ G ⨾ mor ⨾ R
      F⨾mor⨾R≈G⨾mor⨾R = ⨾-cong₁&₂₁ prop
      ex1 : ∃! _≈_ (λ U → mor ⨾ R ≈ mor ⨾ U)
      ex1 = universal F⨾mor⨾R≈G⨾mor⨾R
      mor⨾R≈mor⨾U = proj₁ (proj₂ ex1)
      U≈R = proj₂ (proj₂ ex1) ≈-refl
      U≈S = proj₂ (proj₂ ex1) mor⨾R≈mor⨾S
    in U≈R ⟨≈˘≈⟩ U≈S
```

4.3 Categoric.FinColimits.CoCone2

```
module Categoric.FinColimits.CoCone2 {i j k : Level} {Obj : Set i} {Hom : LocalSetoid Obj j k}
                                     (compOp : CompOp Hom) where
  open SemigroupoidCore compOp
  open Categoric.FinColimits.Initial compOp
```

Although it seems to be natural to use the standard library's unique existential quantification ∃! for the universal morphism of coproducts and pushouts, it turns out that in practice, this plain nested tuple is rather unwieldy, and also leads to less readable code. Instead of resorting to ∃!, we therefore introduce a special-purpose record type CoCone2Univ, the elements of which document that a co-cone defined by two morphisms is universal — we then use this not only for coproducts, but also for pushouts. The field names are also chosen to not indicate the direction of the morphisms, so that they are still natural to use in the opposite setting.

$$\begin{tikzcd} & Z & \\ & X \arrow[u, "\mathsf{univMor}"] & \\ A \arrow[ur, "R"] \arrow[uur, "R'"] & & B \arrow[ul, "S"'] \arrow[uul, "S'"'] \end{tikzcd}$$

```
  record CoCone2Univ {A B X : Obj} (R : Mor A X) (S : Mor B X) {Z : Obj} (R′ : Mor A Z) (S′ : Mor B Z)
                     : Set (i ⊍ j ⊍ k) where
    field
      univMor : Mor X Z
      univMor-factors-left  :  R ⨾ univMor ≈ R′
      univMor-factors-right :  S ⨾ univMor ≈ S′
      univMor-unique : {V : Mor X Z} → R ⨾ V ≈ R′ → S ⨾ V ≈ S′ → V ≈ univMor
    univMor-factors : R ⨾ univMor ≈ R′  × S ⨾ univMor ≈ S′
    univMor-factors = univMor-factors-left, univMor-factors-right

    universal-∃! : ∃! _≈_ (λ U → R ⨾ U ≈ R′ × S ⨾ U ≈ S′)
    universal-∃! = univMor, univMor-factors, λ fact → ≈-sym (univMor-unique (proj₁ fact) (proj₂ fact))

    univMor-unique′ : {V₁ V₂ : Mor X Z}
                    → R ⨾ V₁ ≈ R′ → S ⨾ V₁ ≈ S′ → R ⨾ V₂ ≈ R′ → S ⨾ V₂ ≈ S′
                    → V₁ ≈ V₂
    univMor-unique′ {V₁} {V₂} R⨾V₁≈R′ S⨾V₁≈S′ R⨾V₂≈R′ S⨾V₂≈S′ = ≈-begin
        V₁
```

```
  ≈⟨ univMor-unique R⨾V₁≈R′ S⨾V₁≈S′ ⟩
    univMor
  ≈˘⟨ univMor-unique R⨾V₂≈R′ S⨾V₂≈S′ ⟩
    V₂
  □
```

In addition to the conversion universal-∃! above to the standard library's ∃!, we also add the opposite conversion:

```
CoCone2Univ-from-∃! : {A B X : Obj} {R : Mor A X} {S : Mor B X}
                      {Z : Obj} {R′ : Mor A Z} {S′ : Mor B Z}
                    → ∃! _≈_ (λ U → R ⨾ U ≈ R′ × S ⨾ U ≈ S′)
                    → CoCone2Univ R S R′ S′
CoCone2Univ-from-∃! (U, (R⨾V≈R′, S⨾V≈S′), unique) = record
  {univMor               = U
  ; univMor-factors-left  = R⨾V≈R′
  ; univMor-factors-right = S⨾V≈S′
  ; univMor-unique        = λ R⨾V≈R′ S⨾V≈S′ → ≈-sym (unique (R⨾V≈R′, S⨾V≈S′))
  }
```

Horizontally mirroring a CoCone2Univ as depicted by the diagram above preserves the CoCone2Univ properties:

```
CoCone2Univ-sym : {A B X : Obj} {R : Mor A X} {S : Mor B X}
                  {Z : Obj} {R′ : Mor A Z} {S′ : Mor B Z}
                → CoCone2Univ R S R′ S′ → CoCone2Univ S R S′ R′
CoCone2Univ-sym CCU = let open CoCone2Univ CCU in record
  {univMor = univMor
  ; univMor-factors-left = univMor-factors-right
  ; univMor-factors-right = univMor-factors-left
  ; univMor-unique = λ S⨾V≈S′ R⨾V≈R′ → univMor-unique R⨾V≈R′ S⨾V≈S′
  }
```

```
CoCone2Univ-congSrc : {A B X : Obj} {R₁ R₂ : Mor A X} {S₁ S₂ : Mor B X}
                      {Z : Obj} {R′ : Mor A Z} {S′ : Mor B Z}
                →     R₁ ≈ R₂ → S₁ ≈ S₂ → CoCone2Univ R₁ S₁ R′ S′ → CoCone2Univ R₂ S₂ R′ S′
CoCone2Univ-congSrc R₁≈R₂ S₁≈S₂ CCU = let open CoCone2Univ CCU in record
  {univMor = univMor
  ; univMor-factors-left   = ⨾-cong₁ R₁≈R₂ ⟨≈˘≈⟩ univMor-factors-left
  ; univMor-factors-right  = ⨾-cong₁ S₁≈S₂ ⟨≈˘≈⟩  univMor-factors-right
  ; univMor-unique = λ R₂⨾V≈R′ S₂⨾V≈S′ → univMor-unique (⨾-cong₁ R₁≈R₂ ⟨≈≈⟩ R₂⨾V≈R′)
                                                      (⨾-cong₁ S₁≈S₂ ⟨≈≈⟩ S₂⨾V≈S′)
  }
```

A colimit defined by two morphisms R and S is now a function that produces a CoCone2Univ R S for any other cocone:

```
IsColimit2 : {A B X : Obj} (R : Mor A X) (S : Mor B X) → Set (i ⊍ j ⊍ k)
IsColimit2 {A} {B} {X} R S = {Z : Obj} (R′ : Mor A Z) (S′ : Mor B Z) → CoCone2Univ R S {Z} R′ S′
```

```
module IsColimit2 {A B X : Obj} {R : Mor A X} {S : Mor B X} (isColimit2 : IsColimit2 R S) where
  private
    module Univ {Z : Obj} {R′ : Mor A Z} {S′ : Mor B Z} = CoCone2Univ (isColimit2 {Z} R′ S′)
  open Univ public hiding (univMor; universal-∃!)
  univMor : {Z : Obj} (R′ : Mor A Z) (S′ : Mor B Z) → Mor X Z
  univMor R′ S′ = Univ.univMor {_} {R′} {S′}
  universal-∃! : {C : Obj} (F : Mor A C) (G : Mor B C) → ∃! _≈_ (λ U → R ⨾ U ≈ F × S ⨾ U ≈ G)
  universal-∃! F G = Univ.universal-∃! {_} {F} {G}
  univMor-cong : {C : Obj} {F₁ F₂ : Mor A C} → F₁ ≈ F₂
```

```
        →                    {G₁ G₂ : Mor B C} → G₁ ≈ G₂
        →    univMor F₁ G₁ ≈ univMor F₂ G₂
  univMor-cong {F₁ = F₁} {F₂} F₁≈F₂ {G₁} {G₂} G₁≈G₂ = univMor-unique
        (univMor-factors-left ⟨≈≈⟩ F₁≈F₂)
        (univMor-factors-right ⟨≈≈⟩ G₁≈G₂)

  univMor-cong₁  : {C : Obj} {F₁ F₂ : Mor A C} {G : Mor B C}
                 → F₁ ≈ F₂ → univMor F₁ G ≈ univMor F₂ G
  univMor-cong₁ F₁≈F₂ = univMor-cong F₁≈F₂ ≈-refl

  univMor-cong₂  : {C : Obj} {F : Mor A C} {G₁ G₂ : Mor B C}
                 → G₁ ≈ G₂ → univMor F G₁ ≈ univMor F G₂
  univMor-cong₂ G₁≈G₂ = univMor-cong ≈-refl G₁≈G₂

  univMor-⨾ :   {C D : Obj} {F₁ : Mor A C} {F₂ : Mor B C} {G : Mor C D}
     → (univMor F₁ F₂) ⨾ G ≈ univMor (F₁ ⨾ G) (F₂ ⨾ G)
  univMor-⨾ {F₁ = F₁} {F₂} {G} = univMor-unique
        (≈-begin    R ⨾ (univMor F₁ F₂) ⨾ G
                 ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ univMor-factors-left ⟩
                    F₁ ⨾ G                                □)
        (≈-begin    S ⨾ (univMor F₁ F₂) ⨾ G
                 ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ univMor-factors-right ⟩
                    F₂ ⨾ G                                □)

  IdUnivMor : Mor X X
  IdUnivMor = univMor R S

  IdUnivMor-isLeftIdentity : isLeftIdentity IdUnivMor
  IdUnivMor-isLeftIdentity {Z} {H} = ≈-begin
        IdUnivMor ⨾ H
     ≈⟨ univMor-⨾ ⟩
        univMor (R ⨾ H) (S ⨾ H)
     ≈˘⟨ univMor-unique ≈-refl ≈-refl ⟩
        H
     □

  Univ-IsInitial : IsInitial A → IsInitial B → IsInitial X
  Univ-IsInitial isInit-A isInit-B {C} = univMor (proj₁ isInit-A) (proj₁ isInit-B)
     , λ V → univMor-unique (proj₂ isInit-A _) (proj₂ isInit-B _)
```

4.4 Categoric.FinColimits.Coproduct

```
module Categoric.FinColimits.Coproduct    {i j k : Level} {Obj : Set i} {Hom : LocalSetoid Obj j k}
                                          (compOp : CompOp Hom) where
  open SemigroupoidCore compOp
  open Categoric.FinColimits.Initial compOp
  open Categoric.FinColimits.CoCone2 compOp


  IsCoproduct : {A B S : Obj} (ι : Mor A S) (κ : Mor B S) → Set (i ⊍ j ⊍ k)
  IsCoproduct {A} {B} {S} ι κ = {Z : Obj} (F : Mor A Z) (G : Mor B Z) → CoCone2Univ ι κ {Z} F G


  IsCoproduct-subst     : {A B S : Obj} {ι₁ ι₂ : Mor A S} {κ₁ κ₂ : Mor B S}
                        → ι₁ ≈ ι₂ → κ₁ ≈ κ₂ → IsCoproduct ι₁ κ₁ → IsCoproduct ι₂ κ₂
  IsCoproduct-subst ι₁≈ι₂ κ₁≈κ₂ IsCoprod₁ F G = CoCone2Univ-congSrc ι₁≈ι₂ κ₁≈κ₂ (IsCoprod₁ F G)
```

Most of the derived material in **module** IsCoproduct could be obtained by renaming the corresponding material from IsColimit2 in Categoric.FinColimits.CoCone2 (Sect. 4.3), but for the sake of readability is currently duplicated here. (Checked type annotations for “**renaming**” would make re-use more attractive.)

```
module IsCoproduct {A B S : Obj} {ι : Mor A S} {κ : Mor B S} (isCoproduct : IsCoproduct ι κ) where
  private
    module Univ {C : Obj} {F : Mor A C} {G : Mor B C} = CoCone2Univ (isCoproduct {C} F G)
  open Univ public using () renaming
    (univMor-factors-left      to ι⨾⚠        -- : ι ⨾ (F ⚠ G) ≈ F
    ; univMor-factors-right    to κ⨾⚠        -- : κ ⨾ (F ⚠ G) ≈ G
    ; univMor-factors          to ⚠-factors
    ; univMor-unique           to ⚠-unique   -- : {U : Mor S C} → ι ⨾ U ≈ F → κ ⨾ U ≈ G → U ≈ F ⚠ G
    )
  infixr 5 _⚠_
  _⚠_      : {C : Obj} (F : Mor A C) (G : Mor B C) → Mor S C
  F ⚠ G = Univ.univMor {_} {F} {G}
  ⚠-universal : {C : Obj} (F : Mor A C) (G : Mor B C) → CoCone2Univ ι κ F G
  ⚠-universal = isCoproduct
  ⚠-universal-∃! :    {C : Obj} (F : Mor A C) (G : Mor B C) → ∃! _≈_ (λ U → ι ⨾ U ≈ F × κ ⨾ U ≈ G)
  ⚠-universal-∃! F G = Univ.universal-∃! {_} {F} {G}


  ⚠-cong   :    {C : Obj} {F₁ F₂ : Mor A C} → F₁ ≈ F₂
             →                {G₁ G₂ : Mor B C} → G₁ ≈ G₂
             →    F₁ ⚠ G₁ ≈ F₂ ⚠ G₂
  ⚠-cong {F₁ = F₁} {F₂} F₁≈F₂ {G₁} {G₂} G₁≈G₂ = ⚠-unique (ι⨾⚠ ⟨≈≈⟩ F₁≈F₂) (κ⨾⚠ ⟨≈≈⟩ G₁≈G₂)
  ⚠-cong₁    : {C : Obj} {F₁ F₂ : Mor A C} {G : Mor B C} → F₁ ≈ F₂ → F₁ ⚠ G ≈ F₂ ⚠ G
  ⚠-cong₁ F₁≈F₂ = ⚠-cong F₁≈F₂ ≈-refl
  ⚠-cong₂    : {C : Obj} {F : Mor A C} {G₁ G₂ : Mor B C} → G₁ ≈ G₂ → F ⚠ G₁ ≈ F ⚠ G₂
  ⚠-cong₂ G₁≈G₂ = ⚠-cong ≈-refl G₁≈G₂
  ⚠-⨾  :   {C D : Obj} {F₁ : Mor A C} {F₂ : Mor B C} {G : Mor C D}
       →   (F₁ ⚠ F₂) ⨾ G ≈ F₁ ⨾ G ⚠ F₂ ⨾ G
  ⚠-⨾ {F₁ = F₁} {F₂} {G} = ⚠-unique
    (≈-begin          ι ⨾ (F₁ ⚠ F₂) ⨾ G
                    ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ ι⨾⚠ ⟩
                      F₁ ⨾ G □)
    (≈-begin          κ ⨾ (F₁ ⚠ F₂) ⨾ G
                    ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ κ⨾⚠ ⟩
                      F₂ ⨾ G □)
  to-⚠ :      {Z : Obj} {H : Mor S Z}
           → H ≈ ι ⨾ H ⚠ κ ⨾ H
  to-⚠ {Z} {H} = ⚠-unique ≈-refl ≈-refl
  Id⊞ : Mor S S
  Id⊞ = ι ⚠ κ
  Id⊞-isLeftIdentity : isLeftIdentity Id⊞
  Id⊞-isLeftIdentity {X} {R} = ≈-begin
          Id⊞ ⨾ R
        ≈⟨ ⚠-⨾ ⟩
          ι ⨾ R ⚠ κ ⨾ R
        ≈˘⟨ ⚠-unique ≈-refl ≈-refl ⟩
          R
        □
  ⊞-IsInitial : IsInitial A → IsInitial B → IsInitial S
  ⊞-IsInitial isInit-A isInit-B {C} = proj₁ isInit-A ⚠ proj₁ isInit-B
        , λ V → ⚠-unique (proj₂ isInit-A _) (proj₂ isInit-B _)
  ⦶≈⊞ :    ({X : Obj} {F G : Mor A X} → F ≈ G)
       →   ({X : Obj} {F G : Mor B X} → F ≈ G)
       →   ({X : Obj} {F G : Mor S X} → F ≈ G)
  ⦶≈⊞ ⦶≈₁ ⦶≈₂ {X} {F} {G} = ≈-begin
          F
        ≈⟨ Id⊞-isLeftIdentity ⟨≈˘≈⟩ ⚠-⨾ ⟩
```

```
    ι ⨾ F ⊕ κ ⨾ F
  ≈⟨ ⊕-cong ⦶≈₁ ⦶≈₂ ⟩
    ι ⨾ G ⊕ κ ⨾ G
  ≈⟨ ⊕-⨾ ⟨≈˘≈⟩ Id⊞-isLeftIdentity ⟩
    G
  □

IsCoproduct-∃! : {A B S : Obj} (ι : Mor A S) (κ : Mor B S) → Set (i ⊍ j ⊍ k)
IsCoproduct-∃! {A} {B} {S} ι κ     = {C : Obj} (F : Mor A C) (G : Mor B C)
                                      → ∃! _≈_ (λ (U : Mor S C) → ι ⨾ U ≈ F × κ ⨾ U ≈ G)
IsCoproduct-from-∃! : {A B S : Obj} (ι : Mor A S) (κ : Mor B S) → IsCoproduct-∃! ι κ → IsCoproduct ι κ
IsCoproduct-from-∃! ι κ isCoproduct {C} F G = CoCone2Univ-from-∃! (isCoproduct {C} F G)
IsCoproduct-to-∃! : {A B S : Obj} {ι : Mor A S} {κ : Mor B S} → IsCoproduct ι κ → IsCoproduct-∃! ι κ
IsCoproduct-to-∃! = IsCoproduct.⊕-universal-∃!

open import Categoric.Semigroupoid.SGIso compOp using (SGIsoL; module SGIsoL)
IsCoproduct-SGIsoL    : {A B : Obj}
                       → {S₁ : Obj} {ι₁ : Mor A S₁} {κ₁ : Mor B S₁} → IsCoproduct ι₁ κ₁
                       → {S₂ : Obj} {ι₂ : Mor A S₂} {κ₂ : Mor B S₂} → IsCoproduct ι₂ κ₂
                       → SGIsoL S₁ S₂
IsCoproduct-SGIsoL {A} {B} {S₁} {ι₁} {κ₁} isSum₁ {S₂} {ι₂} {κ₂} isSum₂ = record
  {mor = U
  ; inv = V
  ; prf = record {rightSGInverseL = rightInvL; leftSGInverseL = leftInvL}
  }
  where
    open IsCoproduct isSum₁ using () renaming (_⊕_ to _⊕₁_)
    open IsCoproduct isSum₂ using () renaming (_⊕_ to _⊕₂_)
    open IsCoproduct
    U : Mor S₁ S₂
    U = ι₂ ⊕₁ κ₂
    V : Mor S₂ S₁
    V = ι₁ ⊕₂ κ₁
    ι₁⨾U⨾V≈ι₁ : ι₁ ⨾ U ⨾ V ≈ ι₁
    ι₁⨾U⨾V≈ι₁ = ≈-begin
                  ι₁ ⨾ U ⨾ V
                ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (ι⨾⊕ isSum₁) ⟩
                  ι₂ ⨾ V
                ≈⟨ ι⨾⊕ isSum₂ ⟩
                  ι₁
                □
    κ₁⨾U⨾V≈κ₁ : κ₁ ⨾ U ⨾ V ≈ κ₁
    κ₁⨾U⨾V≈κ₁ = ≈-begin
                  κ₁ ⨾ U ⨾ V
                ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (κ⨾⊕ isSum₁) ⟩
                  κ₂ ⨾ V
                ≈⟨ κ⨾⊕ isSum₂ ⟩
                  κ₁
                □
    ι₂⨾V⨾U≈ι₂ : ι₂ ⨾ V ⨾ U ≈ ι₂
    ι₂⨾V⨾U≈ι₂ = ≈-begin
                  ι₂ ⨾ V ⨾ U
                ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (ι⨾⊕ isSum₂) ⟩
                  ι₁ ⨾ U
                ≈⟨ ι⨾⊕ isSum₁ ⟩
                  ι₂
                □
```

```
    κ₂⨾V⨾U≈κ₂ : κ₂ ⨾ V ⨾ U ≈ κ₂
    κ₂⨾V⨾U≈κ₂ = ≈-begin
                    κ₂ ⨾ V ⨾ U
                  ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (κ⨾◬ isSum₂) ⟩
                    κ₁ ⨾ U
                  ≈⟨ κ⨾◬ isSum₁ ⟩
                    κ₂
                  □
    rightInvL : {C : Obj} → {R : Mor S₁ C} → (U ⨾ V) ⨾ R ≈ R
    rightInvL {C} {R} = ≈-begin
          (U ⨾ V) ⨾ R
        ≈⟨ ⨾-cong₁ (◬-unique isSum₁ ι₁⨾U⨾V≈ι₁ κ₁⨾U⨾V≈κ₁) ⟩
          Id⊞ isSum₁ ⨾ R
        ≈⟨ Id⊞-isLeftIdentity isSum₁ ⟩
          R
        □
    leftInvL : {C : Obj} → {R : Mor S₂ C} → (V ⨾ U) ⨾ R ≈ R
    leftInvL {C} {R} = ≈-begin
          (V ⨾ U) ⨾ R
        ≈⟨ ⨾-cong₁ (◬-unique isSum₂ ι₂⨾V⨾U≈ι₂ κ₂⨾V⨾U≈κ₂) ⟩
          Id⊞ isSum₂ ⨾ R
        ≈⟨ Id⊞-isLeftIdentity isSum₂ ⟩
          R
        □

IsCoproduct-⨾-SGIsoL    : {A B : Obj}
                        → {S₁ : Obj} {ι₁ : Mor A S₁} {κ₁ : Mor B S₁} → IsCoproduct ι₁ κ₁
                        → {S₂ : Obj} → (Φ : SGIsoL S₁ S₂)
                        → IsCoproduct (ι₁ ⨾ SGIsoL.mor Φ) (κ₁ ⨾ SGIsoL.mor Φ)
IsCoproduct-⨾-SGIsoL {A} {B} {S₁} {ι₁} {κ₁} isSum₁ {S₂} Φ {C} F G = record
    {univMor = inv Φ ⨾ (F ◬ G)
    ; univMor-factors-left = ≈-begin
            (ι₁ ⨾ mor Φ) ⨾ (inv Φ ⨾ (F ◬ G))
        ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ (⨾-assocL ⟨≈≈⟩ rightSGInverseL Φ) ⟩
            ι₁ ⨾ (F ◬ G)
        ≈⟨ ι⨾◬ ⟩
            F
        □
    ; univMor-factors-right = ≈-begin
            (κ₁ ⨾ mor Φ) ⨾ (inv Φ ⨾ (F ◬ G))
        ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ (⨾-assocL ⟨≈≈⟩ rightSGInverseL Φ) ⟩
            κ₁ ⨾ (F ◬ G)
        ≈⟨ κ⨾◬ ⟩
            G
        □
    ; univMor-unique = λ {{V} ι₂⨾V≈F κ₂⨾V≈F → ≈-begin
            V
          ≈⟨ leftSGInverseL Φ ⟨≈˘≈⟩ ⨾-assoc ⟩
            inv Φ ⨾ mor Φ ⨾ V
          ≈⟨ ⨾-cong₂ (◬-unique (⨾-assocL ⟨≈≈⟩ ι₂⨾V≈F) (⨾-assocL ⟨≈≈⟩ κ₂⨾V≈F)) ⟩
            inv Φ ⨾ (F ◬ G)
          □}
    }
  where
    open SGIsoL
    open IsCoproduct isSum₁
```

For coproducts, we avoid to have Cospan in the types of the fields in order to ease implementability.

```
record HasCoproducts : Set (i ⊍ j ⊍ k) where
  infixr 3 _⊞_
  field
    _⊞_ : Obj → Obj → Obj
    ι : {A B : Obj} → Mor A (A ⊞ B)
    κ : {A B : Obj} → Mor B (A ⊞ B)
    isCoproduct : {A B : Obj} → IsCoproduct {A} {B} ι κ
  module _ {A B : Obj} where
    open IsCoproduct (isCoproduct {A} {B}) public
```

For images of coproducts under functors, we separate the "essentially local" operations and properties, that involve only the the objects of the coproduct diagram(s) under consideration, from those that involve other objects, typically via _⚠_. However, some of the former are defined using (local occurrences of) some of the latter, so we interleave the definition of "local" and "universal" modules depending on each other, ending each of the two strands with a single re-exporting module. **[WK:** *This has been temporarily reverted to a single module HasCoproductsProps, awaiting resolution of Issue 892.* **]**

```
  module HasCoproductsProps where
    -- module HasCoproductsLocalProps1 where

    infixr 5 _⊕_
    _⊕_ :     {A B C D : Obj} (F : Mor A C) (G : Mor B D) → Mor (A ⊞ B) (C ⊞ D)
    F ⊕ G = F ⨾ ι ⚠ G ⨾ κ
    ι⨾⊕ : {A B C D : Obj} {F : Mor A C} {G : Mor B D} → ι ⨾ (F ⊕ G) ≈ F ⨾ ι
    ι⨾⊕ = ι⨾⚠
    κ⨾⊕ : {A B C D : Obj} {F : Mor A C} {G : Mor B D} → κ ⨾ (F ⊕ G) ≈ G ⨾ κ
    κ⨾⊕ = κ⨾⚠
    module _ {A₁ A₂ A₃ B₁ B₂ B₃ : Obj} {F : Mor A₁ B₁} {G : Mor A₂ B₂} {H : Mor A₃ B₃} where
      ι⨾ι⨾⊕⊕ : ι ⨾ ι ⨾ ((F ⊕ G) ⊕ H) ≈ F ⨾ ι ⨾ ι
      ι⨾ι⨾⊕⊕ = ⨾-cong₂ ι⨾⊕ ⟨≈≈⟩ ⨾-cong₁&₂₁ ι⨾⊕
      κ⨾ι⨾⊕⊕ : κ ⨾ ι ⨾ ((F ⊕ G) ⊕ H) ≈ G ⨾ κ ⨾ ι
      κ⨾ι⨾⊕⊕ = ⨾-cong₂ ι⨾⊕ ⟨≈≈⟩ ⨾-cong₁&₂₁ κ⨾⊕
      ι⨾κ⨾⊕⊕ : ι ⨾ κ ⨾ (F ⊕ (G ⊕ H)) ≈ G ⨾ ι ⨾ κ
      ι⨾κ⨾⊕⊕ = ⨾-cong₂ κ⨾⊕ ⟨≈≈⟩ ⨾-cong₁&₂₁ ι⨾⊕
      κ⨾κ⨾⊕⊕ : κ ⨾ κ ⨾ (F ⊕ (G ⊕ H)) ≈ H ⨾ κ ⨾ κ
      κ⨾κ⨾⊕⊕ = ⨾-cong₂ κ⨾⊕ ⟨≈≈⟩ ⨾-cong₁&₂₁ κ⨾⊕
      ιι⨾⊕⊕ : (ι ⨾ ι) ⨾ ((F ⊕ G) ⊕ H) ≈ F ⨾ ι ⨾ ι
      ιι⨾⊕⊕ = ⨾-assoc ⟨≈≈⟩ ι⨾ι⨾⊕⊕
      κι⨾⊕⊕ : (κ ⨾ ι) ⨾ ((F ⊕ G) ⊕ H) ≈ G ⨾ κ ⨾ ι
      κι⨾⊕⊕ = ⨾-assoc ⟨≈≈⟩ κ⨾ι⨾⊕⊕
      ικ⨾⊕⊕ : (ι ⨾ κ) ⨾ (F ⊕ (G ⊕ H)) ≈ G ⨾ ι ⨾ κ
      ικ⨾⊕⊕ = ⨾-assoc ⟨≈≈⟩ ι⨾κ⨾⊕⊕
      κκ⨾⊕⊕ : (κ ⨾ κ) ⨾ (F ⊕ (G ⊕ H)) ≈ H ⨾ κ ⨾ κ
      κκ⨾⊕⊕ = ⨾-assoc ⟨≈≈⟩ κ⨾κ⨾⊕⊕
    ⊕-cong      : {A₁ B₁ : Obj} {F₁ F₂ : Mor A₁ B₁} → F₁ ≈ F₂
                → {A₂ B₂ : Obj} {G₁ G₂ : Mor A₂ B₂} → G₁ ≈ G₂
                → F₁ ⊕ G₁ ≈ F₂ ⊕ G₂
    ⊕-cong F₁≈F₂ G₁≈G₂ = ⚠-cong (⨾-cong₁ F₁≈F₂) (⨾-cong₁ G₁≈G₂)
    ⊕-cong₁   :   {A₁ B₁ : Obj} {F₁ F₂ : Mor A₁ B₁}
              →   {A₂ B₂ : Obj} {G : Mor A₂ B₂} → F₁ ≈ F₂
              →   F₁ ⊕ G ≈ F₂ ⊕ G
    ⊕-cong₁ F₁≈F₂ = ⊕-cong F₁≈F₂ ≈-refl
    ⊕-cong₂   :   {A₁ B₁ : Obj} {F : Mor A₁ B₁}
              →   {A₂ B₂ : Obj} {G₁ G₂ : Mor A₂ B₂} → G₁ ≈ G₂
```

```
              →      F ⊕ G₁ ≈ F ⊕ G₂
⊕-cong₂ G₁≈G₂  =  ⊕-cong ≈-refl G₁≈G₂
⊕-PreservesMonos       : Set (i ⊍ j ⊍ k)
⊕-PreservesMonos        = {A₁ B₁ A₂ B₂ : Obj} {F : Mor A₁ A₂} {G : Mor B₁ B₂}
                        → isMono F → isMono G → isMono (F ⊕ G)
```

Frequently, but not always, the type of the "identity" in F ⊕Id and Id⊕ G needs to be specified. We use a choice here that, although not symmetric, should be fairly unintrusive most of the time, namely having to write (F ⊕Id) {B} and Id⊕ {A} G to achieve this.

```
infix 10 _⊕Id
_⊕Id : {A₁ A₂ : Obj} (F : Mor A₁ A₂) {B : Obj} → Mor (A₁ ⊞ B) (A₂ ⊞ B)
_⊕Id F =      F ⨾ ι ⚠ κ
Id⊕ :         {A B₁ B₂ : Obj} (G : Mor B₁ B₂) → Mor (A ⊞ B₁) (A ⊞ B₂)
Id⊕ G = ι ⚠ G ⨾ κ
ι⨾⊕Id : {B A₁ A₂ : Obj} {F : Mor A₁ A₂} → ι {A₁} {B} ⨾ (F ⊕Id) ≈ F ⨾ ι
ι⨾⊕Id = ι⨾⚠
κ⨾⊕Id : {B A₁ A₂ : Obj} {F : Mor A₁ A₂} → κ {A₁} {B} ⨾ (F ⊕Id) ≈ κ
κ⨾⊕Id = κ⨾⚠
ι⨾Id⊕ : {A B₁ B₂ : Obj} {G : Mor B₁ B₂} → ι {A} {B₁} ⨾ (Id⊕ G) ≈ ι
ι⨾Id⊕ = ι⨾⚠
κ⨾Id⊕ : {A B₁ B₂ : Obj} {G : Mor B₁ B₂} → κ {A} {B₁} ⨾ (Id⊕ G) ≈ G ⨾ κ
κ⨾Id⊕ = κ⨾⚠
⊕Id-cong       :  {B A₁ A₂ : Obj} {F₁ F₂ : Mor A₁ A₂} → F₁ ≈ F₂
                  → (F₁ ⊕Id) {B} ≈ (F₂ ⊕Id) {B}
⊕Id-cong F₁≈F₂ =  ⚠-cong₁ (⨾-cong₁ F₁≈F₂)
Id⊕-cong       :  {A B₁ B₂ : Obj} {G₁ G₂ : Mor B₁ B₂} → G₁ ≈ G₂
                  → Id⊕ {A} G₁ ≈ Id⊕ {A} G₂
Id⊕-cong G₁≈G₂ =  ⚠-cong₂ (⨾-cong₁ G₁≈G₂)

⊞-swap         :  {A B : Obj} → Mor (A ⊞ B) (B ⊞ A)
⊞-swap         =  κ ⚠ ι

⊞-assoc        :  {A B C : Obj} → Mor ((A ⊞ B) ⊞ C) (A ⊞ (B ⊞ C))
⊞-assoc        =  (Id⊕ ι) ⚠ κ ⨾ κ
κ-⊞-assoc :       {A B C : Obj} → κ ⨾ ⊞-assoc {A} {B} {C} ≈ κ ⨾ κ
κ-⊞-assoc = κ⨾⚠
⊞-assocL       :  {A B C : Obj} → Mor (A ⊞ (B ⊞ C)) ((A ⊞ B) ⊞ C)
⊞-assocL       =  ι ⨾ ι ⚠ (κ ⊕Id)
ι-⊞-assocL :      {A B C : Obj} → ι ⨾ ⊞-assocL {A} {B} {C} ≈ ι ⨾ ι
ι-⊞-assocL = ι⨾⚠
```

The name ⊞-transpose₂ has been chosen because the type can be seen as transposing a two-by-two matrix $\begin{pmatrix} A & B \\ C & D \end{pmatrix}$.

[WK: *Is there an established name for this kind of rearrangement?* **]**

```
⊞-transpose₂ : {A B C D : Obj} → Mor ((A ⊞ B) ⊞ (C ⊞ D)) ((A ⊞ C) ⊞ (B ⊞ D))
⊞-transpose₂ = (ι ⊕ ι) ⚠ (κ ⊕ κ)

-- module HasCoproductsUniversalProps where
  -- open HasCoproductsLocalProps1

module _ {A B C Z : Obj} {F : Mor A Z} {G : Mor B Z} {H : Mor C Z} where
  ι⨾ι⨾⚠⚠ : ι ⨾ ι ⨾ ((F ⚠ G) ⚠ H) ≈ F
```

```
    ι⨾ι⨾⨹⨹  = ⨾-cong₂ ι⨾⨹ ⟨≈≈⟩ ι⨾⨹
    κ⨾ι⨾⨹⨹  : κ ⨾ ι ⨾ ((F ⨹ G) ⨹ H) ≈ G
    κ⨾ι⨾⨹⨹  = ⨾-cong₂ ι⨾⨹ ⟨≈≈⟩ κ⨾⨹
    ι⨾κ⨾⨹⨹  : ι ⨾ κ ⨾ (F ⨹ (G ⨹ H)) ≈ G
    ι⨾κ⨾⨹⨹  = ⨾-cong₂ κ⨾⨹ ⟨≈≈⟩ ι⨾⨹
    κ⨾κ⨾⨹⨹  : κ ⨾ κ ⨾ (F ⨹ (G ⨹ H)) ≈ H
    κ⨾κ⨾⨹⨹  = ⨾-cong₂ κ⨾⨹ ⟨≈≈⟩ κ⨾⨹
    ιι⨾⨹⨹  : (ι ⨾ ι) ⨾ ((F ⨹ G) ⨹ H) ≈ F
    ιι⨾⨹⨹  = ⨾-assoc ⟨≈≈⟩ ι⨾ι⨾⨹⨹
    κι⨾⨹⨹  : (κ ⨾ ι) ⨾ ((F ⨹ G) ⨹ H) ≈ G
    κι⨾⨹⨹  = ⨾-assoc ⟨≈≈⟩ κ⨾ι⨾⨹⨹
    ικ⨾⨹⨹  : (ι ⨾ κ) ⨾ (F ⨹ (G ⨹ H)) ≈ G
    ικ⨾⨹⨹  = ⨾-assoc ⟨≈≈⟩ ι⨾κ⨾⨹⨹
    κκ⨾⨹⨹  : (κ ⨾ κ) ⨾ (F ⨹ (G ⨹ H)) ≈ H
    κκ⨾⨹⨹  = ⨾-assoc ⟨≈≈⟩ κ⨾κ⨾⨹⨹

⊕-⨾-⨹       : {A₁ B₁ A₂ B₂ C : Obj} {F₁ : Mor A₁ B₁} {G₁ : Mor B₁ C}
              → {F₂ : Mor A₂ B₂} {G₂ : Mor B₂ C}
              → (F₁ ⊕ F₂) ⨾ (G₁ ⨹ G₂) ≈ F₁ ⨾ G₁ ⨹ F₂ ⨾ G₂
⊕-⨾-⨹ {F₁ = F₁} {G₁} {F₂} {G₂} = ⨹-unique
      (≈-begin
          ι ⨾ ((F₁ ⊕ F₂) ⨾ (G₁ ⨹ G₂))
        ≈⟨ ⨾-cong₁&₂₁ ι⨾⊕ ⟩
          F₁ ⨾ ι ⨾ (G₁ ⨹ G₂)
        ≈⟨ ⨾-cong₂ ι⨾⨹ ⟩
          F₁ ⨾ G₁
        □)
      (≈-begin
          κ ⨾ ((F₁ ⊕ F₂) ⨾ (G₁ ⨹ G₂))
        ≈⟨ ⨾-cong₁&₂₁ κ⨾⊕ ⟩
          F₂ ⨾ κ ⨾ (G₁ ⨹ G₂)
        ≈⟨ ⨾-cong₂ κ⨾⨹ ⟩
          F₂ ⨾ G₂
        □)

⊕Id-⨾-⨹   :    {A₁ B₁ B₂ C : Obj} {F : Mor A₁ B₁} {G₁ : Mor B₁ C} {G₂ : Mor B₂ C}
           →    (F ⊕Id) ⨾ (G₁ ⨹ G₂) ≈ F ⨾ G₁ ⨹ G₂
⊕Id-⨾-⨹ {F = F} {G₁} {G₂} = ⨹-unique
      (≈-begin
          ι ⨾ ((F ⊕Id) ⨾ (G₁ ⨹ G₂))
        ≈⟨ ⨾-cong₁&₂₁ ι⨾⊕Id ⟩
          F ⨾ ι ⨾ (G₁ ⨹ G₂)
        ≈⟨ ⨾-cong₂ ι⨾⨹ ⟩
          F ⨾ G₁
        □)
      (≈-begin
          κ ⨾ ((F ⊕Id) ⨾ (G₁ ⨹ G₂))
        ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ κ⨾⊕Id ⟩
          κ ⨾ (G₁ ⨹ G₂)
        ≈⟨ κ⨾⨹ ⟩
          G₂
        □)
Id⊕-⨾-⨹   :    {A₂ B₁ B₂ C : Obj} {F : Mor A₂ B₂} {G₁ : Mor B₁ C} {G₂ : Mor B₂ C}
           →    (Id⊕ F) ⨾ (G₁ ⨹ G₂) ≈ G₁ ⨹ F ⨾ G₂
Id⊕-⨾-⨹ {F = F} {G₁} {G₂} = ⨹-unique
      (≈-begin
          ι ⨾ ((Id⊕ F) ⨾ (G₁ ⨹ G₂))
        ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ ι⨾Id⊕ ⟩
```

```
          ι ⨾ (G₁ ⨹ G₂)
        ≈⟨ ι⨾⨹ ⟩
          G₁
        □)
      (≈-begin
          κ ⨾ ((Id⊕ F) ⨾ (G₁ ⨹ G₂))
        ≈⟨ ⨾-cong₁&₂₁ κ⨾Id⊕ ⟩
          F ⨾ κ ⨾ (G₁ ⨹ G₂)
        ≈⟨ ⨾-cong₂ κ⨾⨹ ⟩
          F ⨾ G₂
        □)


⊞-swap-⨾-⨹       :  {A B D : Obj} {F : Mor A D} {G : Mor B D}
                  →  ⊞-swap ⨾ (F ⨹ G) ≈ G ⨹ F
⊞-swap-⨾-⨹ {F = F} {G} = ≈-begin
        ⊞-swap ⨾ (F ⨹ G)
    ≈⟨ ⨹-⨾ ⟩
        κ ⨾ (F ⨹ G) ⨹ ι ⨾ (F ⨹ G)
    ≈⟨ ⨹-cong κ⨾⨹ ι⨾⨹ ⟩
        G ⨹ F
    □


⊞-assoc-⨹⨹      :  {A B C D : Obj} {F : Mor A D} {G : Mor B D} {H : Mor C D}
                  →  ⊞-assoc ⨾ (F ⨹ (G ⨹ H)) ≈ (F ⨹ G) ⨹ H
⊞-assoc-⨹⨹     {F = F} {G} {H} = ≈-begin
          ⊞-assoc ⨾ (F ⨹ (G ⨹ H))
    ≈⟨ ⨹-⨾ ⟩
          (ι ⨹ ι ⨾ κ) ⨾ (F ⨹ (G ⨹ H)) ⨹ (κ ⨾ κ) ⨾ (F ⨹ (G ⨹ H))
    ≈⟨ ⨹-cong (⨹-⨾ ⟨≈≈⟩ ⨹-cong ι⨾⨹ ικ⨾⨹⨹) κκ⨾⨹⨹ ⟩
          (F ⨹ G) ⨹ H
    □

⊞-assocL-⨹⨹     :  {A B C D : Obj} {F : Mor A D} {G : Mor B D} {H : Mor C D}
                  → ⊞-assocL ⨾ ((F ⨹ G) ⨹ H) ≈ F ⨹ (G ⨹ H)
⊞-assocL-⨹⨹    {F = F} {G} {H} = ≈-begin
          ((ι ⨾ ι) ⨹ (κ ⊕Id)) ⨾ ((F ⨹ G) ⨹ H)
    ≈⟨ ⨹-⨾ ⟨≈≈⟩ ⨹-cong ιι⨾⨹⨹ (⊕Id-⨾-⨹ ⟨≈≈⟩ ⨹-cong₁ κ⨾⨹) ⟩
          F ⨹ (G ⨹ H)
    □


⊞-assoc-⨾-⊕⨹-⨾-⨹     :  {A B C D E : Obj}
                          {F : Mor A D} {G : Mor B D} {H : Mor C D} {J : Mor D E}
                        → ⊞-assoc ⨾ (F ⊕ (G ⨹ H)) ⨾ (J ⨹ J) ≈ ((F ⨹ G) ⊕ H) ⨾ (J ⨹ J)
⊞-assoc-⨾-⊕⨹-⨾-⨹ {F = F} {G} {H} {J} = ≈-begin
        ⊞-assoc ⨾ (F ⊕ (G ⨹ H)) ⨾ (J ⨹ J)
    ≈⟨ ⨾-cong₂ (⊕-⨾-⨹ ⟨≈≈⟩ ⨹-cong₂ ⨹-⨾) ⟩
        ⊞-assoc ⨾ (F ⨾ J ⨹ (G ⨾ J ⨹ H ⨾ J))
    ≈⟨ ⊞-assoc-⨹⨹ ⟩
        (F ⨾ J ⨹ G ⨾ J) ⨹ H ⨾ J
    ≈˘⟨ ⊕-⨾-⨹ ⟨≈≈⟩ ⨹-cong₁ ⨹-⨾ ⟩
        ((F ⨹ G) ⊕ H) ⨾ (J ⨹ J)
    □


⊞-transpose₂-⨾       :  {A B C D Z : Obj}
                          {F : Mor A Z} {G : Mor B Z} {H : Mor C Z} {K : Mor D Z}
                      →  ⊞-transpose₂ ⨾ ((F ⨹ G) ⨹ (H ⨹ K)) ≈ (F ⨹ H) ⨹ (G ⨹ K)
⊞-transpose₂-⨾      {F = F} {G} {H} {K} = ≈-begin
```

```
          ⊞-transpose₂ ⨾ ((F ⨹ G) ⨹ (H ⨹ K))
      ≈⟨ ⨹-⨾ ⟩
          (ι ⊕ ι) ⨾ ((F ⨹ G) ⨹ (H ⨹ K)) ⨹ (κ ⊕ κ) ⨾ ((F ⨹ G) ⨹ (H ⨹ K))
      ≈⟨ ⨹-cong (⊕-⨾-⨹ ⟨≈≈⟩ ⨹-cong ι⨾⨹ ι⨾⨹) (⊕-⨾-⨹ ⟨≈≈⟩ ⨹-cong κ⨾⨹ κ⨾⨹) ⟩
          (F ⨹ H) ⨹ (G ⨹ K)
      □


-- module HasCoproductsLocalProps2 where
   -- open HasCoproductsLocalProps1
   -- open HasCoproductsUniversalProps


⊕-⨾-Id⊞    :    {A₁ B₁ A₂ B₂ : Obj} {F₁ : Mor A₁ B₁} {F₂ : Mor A₂ B₂}
            →    (F₁ ⊕ F₂) ⨾ Id⊞ ≈ (F₁ ⊕ F₂)
⊕-⨾-Id⊞ = ⊕-⨾-⨹
⨾-⊕-⨾        : {A₁ A₂ A₃ : Obj} {F₁ : Mor A₁ A₂} {G₁ : Mor A₂ A₃}
             → {B₁ B₂ B₃ : Obj} {F₂ : Mor B₁ B₂} {G₂ : Mor B₂ B₃}
             → (F₁ ⨾ G₁) ⊕ (F₂ ⨾ G₂) ≈ (F₁ ⊕ F₂) ⨾ (G₁ ⊕ G₂)
⨾-⊕-⨾ {F₁ = F₁} {G₁} {F₂ = F₂} {G₂} = ≈-begin
             (F₁ ⨾ G₁) ⊕ (F₂ ⨾ G₂)
           ≈⟨ ⨹-cong ⨾-assoc   ⨾-assoc ⟩
             F₁ ⨾ G₁ ⨾ ι ⨹ F₂ ⨾ G₂ ⨾ κ
           ≈˘⟨ ⊕-⨾-⨹ ⟩
             (F₁ ⊕ F₂) ⨾ (G₁ ⊕ G₂)
           □


⊕Id-⨾-⊕    :    {A₁ B₁ B₂ C₁ C₂ : Obj} {F : Mor A₁ B₁} {G₁ : Mor B₁ C₁} {G₂ : Mor B₂ C₂}
            →    (F ⊕Id) ⨾ (G₁ ⊕ G₂) ≈ F ⨾ G₁ ⊕ G₂
⊕Id-⨾-⊕ {F = F} {G₁} {G₂} = ≈-begin
          (F ⊕Id) ⨾ (G₁ ⊕ G₂)
      ≈⟨ ⊕Id-⨾-⨹ ⟩
          F ⨾ G₁ ⨾ ι ⨹ G₂ ⨾ κ
      ≈⟨ ⨹-cong₁ ⨾-assocL ⟩
          F ⨾ G₁ ⊕ G₂
      □
⊕-⨾-⊕Id    :    {B₁ B₂ C₁ C₂ D₁ : Obj} {G₁ : Mor B₁ C₁} {G₂ : Mor B₂ C₂} {H : Mor C₁ D₁}
            →    (G₁ ⊕ G₂) ⨾ (H ⊕Id) ≈ G₁ ⨾ H ⊕ G₂
⊕-⨾-⊕Id {G₁ = G₁} {G₂} {H} = ≈-begin
          (G₁ ⊕ G₂) ⨾ (H ⊕Id)
      ≈⟨ ⊕-⨾-⨹ ⟩
          G₁ ⨾ H ⨾ ι ⨹ G₂ ⨾ κ
      ≈⟨ ⨹-cong₁ ⨾-assocL ⟩
          G₁ ⨾ H ⊕ G₂
      □
Id⊕-⨾-⊕    :    {A₂ B₁ B₂ C₁ C₂ : Obj} {F : Mor A₂ B₂} {G₁ : Mor B₁ C₁} {G₂ : Mor B₂ C₂}
            →    (Id⊕ F) ⨾ (G₁ ⊕ G₂) ≈ G₁ ⊕ F ⨾ G₂
Id⊕-⨾-⊕ {F = F} {G₁} {G₂} = ≈-begin
          (Id⊕ F) ⨾ (G₁ ⊕ G₂)
      ≈⟨ Id⊕-⨾-⨹ ⟩
          G₁ ⨾ ι ⨹ F ⨾ G₂ ⨾ κ
      ≈⟨ ⨹-cong₂ ⨾-assocL ⟩
          G₁ ⊕ F ⨾ G₂
      □
⊕-⨾-Id⊕    :    {B₁ B₂ C₁ C₂ D₂ : Obj} {G₁ : Mor B₁ C₁} {G₂ : Mor B₂ C₂} {H : Mor C₂ D₂}
            →    (G₁ ⊕ G₂) ⨾ (Id⊕ H) ≈ G₁ ⊕ G₂ ⨾ H
⊕-⨾-Id⊕ {G₁ = G₁} {G₂} {H} = ≈-begin
          (G₁ ⊕ G₂) ⨾ (Id⊕ H)
```

```
    ≈⟨ ⊕-⨾-⨹ ⟩
        G₁ ⨾ ι ⨹ G₂ ⨾ H ⨾ κ
    ≈⟨ ⨹-cong₂ ⨾-assocL ⟩
        G₁ ⊕ G₂ ⨾ H
    □

⊕Id-⨾-Id⊕       :  {B₁ B₂ C₁ C₂ : Obj} {G₁ : Mor B₁ C₁} {G₂ : Mor B₂ C₂}
                → (G₁ ⊕Id) ⨾ (Id⊕ G₂) ≈ G₁ ⊕ G₂
⊕Id-⨾-Id⊕ = ⊕Id-⨾-⨹

Id⊕-⨾-⊕Id       :  {B₁ B₂ C₁ C₂ : Obj} {G₁ : Mor B₁ C₁} {G₂ : Mor B₂ C₂}
                → (Id⊕ G₂) ⨾ (G₁ ⊕Id) ≈ G₁ ⊕ G₂
Id⊕-⨾-⊕Id = Id⊕-⨾-⨹

⨾-⊕Id        : {B A₁ A₂ A₃ : Obj} {F : Mor A₁ A₂} {G : Mor A₂ A₃}
             → ((F ⨾ G) ⊕Id) {B} ≈ (F ⊕Id) ⨾ (G ⊕Id)
⨾-⊕Id {F = F} {G} = ≈-begin
        (F ⨾ G) ⊕Id
    ≈⟨ ⨹-cong₁ ⨾-assoc ⟩
        F ⨾ G ⨾ ι ⨹ κ
    ≈˘⟨ ⊕Id-⨾-⨹ ⟩
        (F ⊕Id) ⨾ (G ⊕Id)
    □

Id⊕-⨾        : {A B₁ B₂ B₃ : Obj} {F : Mor B₁ B₂} {G : Mor B₂ B₃}
             → Id⊕ {A} (F ⨾ G) ≈ (Id⊕ F) ⨾ (Id⊕ G)
Id⊕-⨾ {F = F} {G} = ≈-begin
        Id⊕ (F ⨾ G)
    ≈⟨ ⨹-cong₂ ⨾-assoc ⟩
        ι ⨹ F ⨾ G ⨾ κ
    ≈˘⟨ Id⊕-⨾-⨹ ⟩
        (Id⊕ F) ⨾ (Id⊕ G)
    □

Id⊞-⊕Id :    {A B C : Obj} → ((Id⊞ {A} {B}) ⊕Id) {C} ≈ Id⊞
Id⊞-⊕Id = ⨹-cong₁ Id⊞-isLeftIdentity

Id⊕-Id⊞ :    {A B C : Obj} → Id⊕ {A} (Id⊞ {B} {C}) ≈ Id⊞
Id⊕-Id⊞ = ⨹-cong₂ Id⊞-isLeftIdentity


⊕-⨾-⊞-swap      : {A₁ B₁ A₂ B₂ : Obj} {F : Mor A₁ A₂} {G : Mor B₁ B₂}
                → (F ⊕ G) ⨾ ⊞-swap ≈ ⊞-swap ⨾ (G ⊕ F)
⊕-⨾-⊞-swap {F = F} {G} = ≈-begin
        (F ⊕ G) ⨾ ⊞-swap
    ≈⟨ ⊕-⨾-⨹ ⟩
        F ⨾ κ ⨹ G ⨾ ι
    ≈˘⟨ ⊞-swap-⨾-⨹ ⟩
        ⊞-swap ⨾ (G ⊕ F)
    □

⊞-swap-⨾-⊕      : {A₁ B₁ A₂ B₂ : Obj} {F : Mor A₁ A₂} {G : Mor B₁ B₂}
                → ⊞-swap ⨾ (F ⊕ G) ≈ (G ⊕ F) ⨾ ⊞-swap
⊞-swap-⨾-⊕ {F = F} {G} = ≈-sym ⊕-⨾-⊞-swap

⊕Id-⨾-⊞-swap       : {B A₁ A₂ : Obj} {F : Mor A₁ A₂}
                   → (F ⊕Id) {B} ⨾ ⊞-swap ≈ ⊞-swap ⨾ (Id⊕ F)
⊕Id-⨾-⊞-swap {F = F} = ≈-begin
        (F ⊕Id) ⨾ ⊞-swap
    ≈⟨ ⊕Id-⨾-⨹ ⟩
        F ⨾ κ ⨹ ι
    ≈˘⟨ ⊞-swap-⨾-⨹ ⟩
        ⊞-swap ⨾ (Id⊕ F)
    □

Id⊕-⨾-⊞-swap       : {A B₁ B₂ : Obj} {G : Mor B₁ B₂}
                   → (Id⊕ {A} G) ⨾ ⊞-swap ≈ ⊞-swap ⨾ (G ⊕Id)
```

```
Id⊕-⨾-⊞-swap {G = G} = ≈-begin
        (Id⊕ G) ⨾ ⊞-swap
    ≈⟨ Id⊕-⨾-⨹ ⟩
        κ ⨹ G ⨾ ι
    ≈˘⟨ ⊞-swap-⨾-⨹ ⟩
        ⊞-swap ⨾ (G ⊕Id)
    □


κ-ι-⊞-assoc :        {A B C : Obj} → κ ⨾ ι ⨾ ⊞-assoc {A} {B} {C} ≈ ι ⨾ κ
κ-ι-⊞-assoc = κ⨾ι⨾⨹ ⨹
ι-ι-⊞-assoc :        {A B C : Obj} → ι ⨾ ι ⨾ ⊞-assoc {A} {B} {C} ≈ ι
ι-ι-⊞-assoc = ι⨾ι⨾⨹ ⨹
κι-⊞-assoc :       {A B C : Obj} → (κ ⨾ ι) ⨾ ⊞-assoc {A} {B} {C} ≈ ι ⨾ κ
κι-⊞-assoc = κι⨾⨹ ⨹
ιι-⊞-assoc :       {A B C : Obj} → (ι ⨾ ι) ⨾ ⊞-assoc {A} {B} {C} ≈ ι
ιι-⊞-assoc = ιι⨾⨹ ⨹
ι-κ-⊞-assocL :       {A B C : Obj} → ι ⨾ κ ⨾ ⊞-assocL {A} {B} {C} ≈ κ ⨾ ι
ι-κ-⊞-assocL = ι⨾κ⨾⨹ ⨹
κ-κ-⊞-assocL :       {A B C : Obj} → κ ⨾ κ ⨾ ⊞-assocL {A} {B} {C} ≈ κ
κ-κ-⊞-assocL = κ⨾κ⨾⨹ ⨹
ικ-⊞-assocL :        {A B C : Obj} → (ι ⨾ κ) ⨾ ⊞-assocL {A} {B} {C} ≈ κ ⨾ ι
ικ-⊞-assocL = ικ⨾⨹ ⨹
κκ-⊞-assocL :        {A B C : Obj} → (κ ⨾ κ) ⨾ ⊞-assocL {A} {B} {C} ≈ κ
κκ-⊞-assocL = κκ⨾⨹ ⨹


⊞-assoc-assocL : {A B C : Obj} → ⊞-assoc {A} {B} {C} ⨾ ⊞-assocL ≈ Id⊞
⊞-assoc-assocL {A} {B} {C} = ≈-begin
        ⊞-assoc ⨾ ⊞-assocL
    ≈⟨ ⨹-⨾ ⟩
        (Id⊕ ι) ⨾ ⊞-assocL ⨹ (κ ⨾ κ) ⨾ ⊞-assocL
    ≈⟨ ⨹-cong Id⊕-⨾-⨹ κκ-⊞-assocL ⟩
        (ι ⨾ ι ⨹ ι ⨾ (κ ⊕Id)) ⨹ κ
    ≈⟨ ⨹-cong₁ (⨹-cong₂ ι⨾⊕Id) ⟩
        (ι ⨾ ι ⨹ κ ⨾ ι) ⨹ κ
    ≈˘⟨ ⨹-cong₁ ⨹-⨾ ⟩
        Id⊞ ⨾ ι ⨹ κ
    ≈⟨ ⨹-cong₁ Id⊞-isLeftIdentity ⟩
        Id⊞
    □
⊞-assocL-assoc : {A B C : Obj} → ⊞-assocL {A} {B} {C} ⨾ ⊞-assoc ≈ Id⊞
⊞-assocL-assoc {A} {B} {C} = ≈-begin
        ⊞-assocL ⨾ ⊞-assoc
    ≈⟨ ⨹-⨾ ⟩
        (ι ⨾ ι) ⨾ ⊞-assoc ⨹ (κ ⊕Id) ⨾ ⊞-assoc
    ≈⟨ ⨹-cong ιι-⊞-assoc ⊕Id-⨾-⨹ ⟩
        ι ⨹ ((κ ⨾ (Id⊕ ι)) ⨹ (κ ⨾ κ))
    ≈⟨ ⨹-cong₂ (⨹-cong₁ κ⨾Id⊕) ⟩
        ι ⨹ (ι ⨾ κ ⨹ κ ⨾ κ)
    ≈˘⟨ ⨹-cong₂ ⨹-⨾ ⟩
        ι ⨹ Id⊞ ⨾ κ
    ≈⟨ ⨹-cong₂ Id⊞-isLeftIdentity ⟩
        Id⊞
    □


⊞-assoc-⨾       : {A₁ B₁ C₁ A₂ B₂ C₂ : Obj} {F : Mor A₁ A₂} {G : Mor B₁ B₂} {H : Mor C₁ C₂}
                → ⊞-assoc ⨾ (F ⊕ (G ⊕ H)) ≈ ((F ⊕ G) ⊕ H) ⨾ ⊞-assoc
⊞-assoc-⨾ {F = F} {G} {H} = ≈-begin
```

```
          ⊞-assoc ⨾ (F ⊕ (G ⊕ H))
    ≈⟨ ⨾-cong₂ (⨹-cong₂ (⨹-⨾ ⟨≈≈⟩ ⨹-cong ⨾-assoc ⨾-assoc)) ⟩
          ⊞-assoc ⨾ (F ⨾ ι ⨹ (G ⨾ ι ⨾ κ ⨹   H ⨾ κ ⨾ κ))
    ≈⟨ ⊞-assoc-⨹⨹ ⟩
       (F ⨾ ι ⨹ G ⨾ ι ⨾ κ) ⨹ H ⨾ κ ⨾ κ
    ≈⟨ ⨹-cong₁ (⨹-cong₂ ⨾-assocL ⟨≈≈˘⟩ ⊕-⨾-Id⊕) ⟩
       (F ⊕ G) ⨾ (Id⊕ ι)   ⨹ H ⨾ κ ⨾ κ
    ≈˘⟨ ⊕-⨾-⨹ ⟩
       ((F ⊕ G) ⊕ H) ⨾ ⊞-assoc
    □

⨾-⊞-assoc       : {A₁ B₁ C₁ A₂ B₂ C₂ : Obj} {F : Mor A₁ A₂} {G : Mor B₁ B₂} {H : Mor C₁ C₂}
                → ((F ⊕ G) ⊕ H) ⨾ ⊞-assoc ≈ ⊞-assoc ⨾ (F ⊕ (G ⊕ H))
⨾-⊞-assoc = ≈-sym ⊞-assoc-⨾


⊞-assocL-⨾        : {A₁ B₁ C₁ A₂ B₂ C₂ : Obj} {F : Mor A₁ A₂} {G : Mor B₁ B₂} {H : Mor C₁ C₂}
                  → ⊞-assocL ⨾ ((F ⊕ G) ⊕ H) ≈ (F ⊕ (G ⊕ H)) ⨾ ⊞-assocL
⊞-assocL-⨾ {F = F} {G} {H} =    ≈-begin
          ⊞-assocL ⨾ ((F ⊕ G) ⊕ H)
    ≈⟨ ⨾-cong₂ (⨹-cong₁ (⨹-⨾ ⟨≈≈⟩ ⨹-cong ⨾-assoc ⨾-assoc)) ⟩
          ⊞-assocL ⨾ ((F ⨾ ι ⨾ ι ⨹ G ⨾ κ ⨾ ι) ⨹ H ⨾ κ)
    ≈⟨ ⊞-assocL-⨹⨹ ⟩
       F ⨾ ι ⨾ ι ⨹ (G ⨾ κ ⨾ ι ⨹       H ⨾ κ)
    ≈⟨ ⨹-cong₂ (⨹-cong₁ ⨾-assocL ⟨≈≈˘⟩ ⊕-⨾-⊕Id) ⟩
       F ⨾ ι ⨾ ι ⨹ (G ⊕ H) ⨾ (κ ⊕Id)
    ≈˘⟨ ⊕-⨾-⨹ ⟩
       (F ⊕ (G ⊕ H)) ⨾ ⊞-assocL
    □

⨾-⊞-assocL        : {A₁ B₁ C₁ A₂ B₂ C₂ : Obj} {F : Mor A₁ A₂} {G : Mor B₁ B₂} {H : Mor C₁ C₂}
                  → (F ⊕ (G ⊕ H)) ⨾ ⊞-assocL ≈ ⊞-assocL ⨾ ((F ⊕ G) ⊕ H)
⨾-⊞-assocL {F = F} {G} {H} = ≈-sym ⊞-assocL-⨾


Id⊕-⨾-⊞-assoc       : {A B C₁ C₂ : Obj} {H : Mor C₁ C₂}
                    → (Id⊕ {A ⊞ B} H) ⨾ ⊞-assoc ≈ ⊞-assoc ⨾ (Id⊕ (Id⊕ H))
Id⊕-⨾-⊞-assoc {H = H} = ≈-begin
             (Id⊕ H) ⨾ ⊞-assoc
        ≈⟨ Id⊕-⨾-⨹ ⟩
             (Id⊕ ι) ⨹ H ⨾ κ ⨾ κ
        ≈˘⟨ ⊞-assoc-⨹⨹ ⟩
             ⊞-assoc ⨾ (ι ⨹ (ι ⨾ κ ⨹ H ⨾ κ ⨾ κ))
        ≈˘⟨ ⨾-cong₂ (⨹-cong₂ (⨹-⨾ ⟨≈≈⟩ ⨹-cong₂ ⨾-assoc)) ⟩
             ⊞-assoc ⨾ (Id⊕ (Id⊕ H))
        □

⊕Id-⨾-⊞-assocL      : {A₁ A₂ B C : Obj} {F : Mor A₁ A₂}
                  → (F ⊕Id) {B ⊞ C} ⨾ ⊞-assocL ≈ ⊞-assocL ⨾ ((F ⊕Id) ⊕Id)
⊕Id-⨾-⊞-assocL {F = F} = ≈-begin
             (F ⊕Id) ⨾ ⊞-assocL
        ≈⟨ ⊕Id-⨾-⨹ ⟩
             F ⨾ ι ⨾ ι ⨹ (κ ⊕Id)
        ≈˘⟨ ⊞-assocL-⨹⨹ ⟩
             ⊞-assocL ⨾ ((F ⨾ ι ⨾ ι ⨹ κ ⨾ ι) ⨹ κ)
        ≈˘⟨ ⨾-cong₂ (⨹-cong₁ (⨹-⨾ ⟨≈≈⟩ ⨹-cong₁ ⨾-assoc)) ⟩
             ⊞-assocL ⨾ ((F ⊕Id) ⊕Id)
        □


⊞-₂₂assoc₁₂₁        : {A B C D : Obj} → Mor ((A ⊞ B) ⊞ (C ⊞ D)) (A ⊞ (B ⊞ C) ⊞ D)
⊞-₂₂assoc₁₂₁        = (ι ⨹ ι ⨾ ι ⨾ κ) ⨹ ((κ ⨾ ι ⨹ κ) ⨾ κ)
⊞-₂₂assoc₁₂₁-split : {A B C D : Obj}
```

```
    → ⊞-₂₂assoc₁₂₁ {A} {B} {C} {D} ≈ ⊞-assoc {A} {B} {C ⊞ D} ⨾ (Id⊕ ⊞-assocL)
⊞-₂₂assoc₁₂₁-split {A} {B} {C} {D} = ≈-sym (≈-begin
        ⊞-assoc {A} {B} {C ⊞ D} ⨾ (Id⊕ ⊞-assocL)
    ≈⟨ ⨹-⨾ ⟨≈≈⟩ ⨹-cong ⨹-⨾ (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ κ⨾Id⊕) ⟩
        (ι ⨾ (Id⊕ ⊞-assocL) ⨹ (ι ⨾ κ) ⨾ (Id⊕ ⊞-assocL)) ⨹ κ ⨾ ⊞-assocL ⨾ κ
    ≈⟨ ⨹-cong₁ (⨹-cong ι⨾Id⊕ (⨾-cong₁₂&₂ κ⨾Id⊕ ⟨≈≈⟩ ⨾-cong₁ ι-⊞-assocL ⟨≈≈⟩ ⨾-assoc)) ⟩
        (ι ⨹ ι ⨾ ι ⨾ κ) ⨹ κ ⨾ ⊞-assocL ⨾ κ
    ≈⟨ ⨹-cong₂ (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ κ⨾⨹) ⟩
        ⊞-₂₂assoc₁₂₁ {A} {B} {C} {D}
    □)

⊞-₁₂₁assoc₂₂       : {A B C D : Obj} → Mor (A ⊞ (B ⊞ C) ⊞ D) ((A ⊞ B) ⊞ (C ⊞ D))
⊞-₁₂₁assoc₂₂       = ι ⨾ ι    ⨹ ((κ ⨾ ι ⨹ ι ⨾ κ) ⨹ (κ ⨾ κ))

⊞-₁₂₁assoc₂₂-split : {A B C D : Obj}
    → ⊞-₁₂₁assoc₂₂ {A} {B} {C} {D} ≈ (Id⊕ ⊞-assoc) ⨾ ⊞-assocL {A} {B} {C ⊞ D}
⊞-₁₂₁assoc₂₂-split {A} {B} {C} {D} = ≈-sym (≈-begin
        (Id⊕ ⊞-assoc) ⨾ ⊞-assocL {A} {B} {C ⊞ D}
    ≈⟨ Id⊕-⨾-⨹ ⟩
        ι ⨾ ι     ⨹ (⊞-assoc ⨾ (κ ⨾ ι ⨹ κ))
    ≈⟨ ⨹-cong₂ (⨹-⨾ ⟨≈≈⟩ ⨹-cong Id⊕-⨾-⨹ (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ κ⨾⨹)) ⟩
        ⊞-₁₂₁assoc₂₂ {A} {B} {C} {D}
    □)

⊞-₂₂assoc₁₂₁-₁₂₁assoc₂₂ : {A B C D : Obj} → ⊞-₂₂assoc₁₂₁ {A} {B} {C} {D} ⨾ ⊞-₁₂₁assoc₂₂ ≈ Id⊞
⊞-₂₂assoc₁₂₁-₁₂₁assoc₂₂ {A} {B} {C} {D} = ≈-begin
        ⊞-₂₂assoc₁₂₁ {A} {B} {C} {D} ⨾ ⊞-₁₂₁assoc₂₂
    ≈⟨ ⨾-cong ⊞-₂₂assoc₁₂₁-split ⊞-₁₂₁assoc₂₂-split ⟩
        (⊞-assoc {A} {B} {C ⊞ D} ⨾ (Id⊕ ⊞-assocL)) ⨾
        (Id⊕ ⊞-assoc) ⨾ ⊞-assocL {A} {B} {C ⊞ D}
    ≈⟨ ⨾-₂₂assoc₁₂₁ ⟨≈≈⟩ ⨾-cong₂₁ (Id⊕-⨾ ⟨≈˘≈⟩ Id⊕-cong ⊞-assocL-assoc ⟨≈≈⟩ Id⊕-Id⊞) ⟩
        ⊞-assoc {A} {B} {C ⊞ D} ⨾ Id⊞ ⨾ ⊞-assocL {A} {B} {C ⊞ D}
    ≈⟨ ⨾-cong₂ Id⊞-isLeftIdentity ⟨≈≈⟩ ⊞-assoc-assocL ⟩
        Id⊞
    □

⊞-₁₂₁assoc₂₂-₂₂assoc₁₂₁ : {A B C D : Obj} → ⊞-₁₂₁assoc₂₂ {A} {B} {C} {D} ⨾ ⊞-₂₂assoc₁₂₁ ≈ Id⊞
⊞-₁₂₁assoc₂₂-₂₂assoc₁₂₁ {A} {B} {C} {D} = ≈-begin
        ⊞-₁₂₁assoc₂₂ ⨾ ⊞-₂₂assoc₁₂₁
    ≈⟨ ⨾-cong ⊞-₁₂₁assoc₂₂-split ⊞-₂₂assoc₁₂₁-split ⟩
        ((Id⊕ ⊞-assoc) ⨾ ⊞-assocL {A} {B} {C ⊞ D}) ⨾
        ⊞-assoc {A} {B} {C ⊞ D} ⨾ (Id⊕ ⊞-assocL)
    ≈⟨ ⨾-₂₂assoc₁₂₁ ⟨≈≈⟩ ⨾-cong₂₁ ⊞-assocL-assoc ⟩
        (Id⊕ ⊞-assoc) ⨾ Id⊞ ⨾ (Id⊕ ⊞-assocL)
    ≈⟨ ⨾-cong₂ Id⊞-isLeftIdentity ⟨≈≈⟩ (Id⊕-⨾ ⟨≈˘≈⟩ Id⊕-cong ⊞-assoc-assocL ⟨≈≈⟩ Id⊕-Id⊞) ⟩
        Id⊞
    □


⊞-assoc-pentagon₀ : {A B C D : Obj}
    →   ⊞-assoc {A ⊞ B} {C} {D} ⨾ ⊞-assoc {A} {B} {C ⊞ D}
    ≈   (⊞-assoc {A} {B} {C} ⊕Id) ⨾ ⊞-assoc {A} {B ⊞ C} {D} ⨾ (Id⊕ (⊞-assoc {B} {C} {D}))
⊞-assoc-pentagon₀ = ≈-begin
        ⊞-assoc ⨾ ⊞-assoc
    ≈⟨ ⨹-⨾ ⟩
        (Id⊕ ι) ⨾ ⊞-assoc ⨹ (κ ⨾ κ) ⨾ ⊞-assoc
    ≈⟨ ⨹-cong Id⊕-⨾-⊞-assoc (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ κ-⊞-assoc) ⟩
        (⊞-assoc ⨾ Id⊕ (Id⊕ ι)) ⨹ κ ⨾ κ ⨾ κ
    ≈˘⟨ ⨹-⨾ ⟨≈≈⟩ ⨹-cong (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ (Id⊕-⨾ ⟨≈˘≈⟩ Id⊕-cong ι⨾⨹))
                            (⨾-cong₁₂&₂ κ⨾Id⊕ ⟨≈≈⟩ ⨾-cong₁ κ-⊞-assoc ⟨≈≈⟩ ⨾-assoc) ⟩
        ((⊞-assoc ⨾ Id⊕ ι) ⨹ κ ⨾ κ) ⨾ (Id⊕ ⊞-assoc)
    ≈˘⟨ ⨾-assocL     ⟨≈≈⟩ ⨾-cong₁ ⊕Id-⨾-⨹ ⟩
        (⊞-assoc ⊕Id) ⨾ ⊞-assoc ⨾ (Id⊕ ⊞-assoc)
```

```
    □


  ⊞-assocL-pentagon₀ : {A B C D : Obj}
      → ⊞-assocL {A} {B} {C ⊞ D} ⨾ ⊞-assocL {A ⊞ B} {C} {D}
      ≈ (Id⊕ ⊞-assocL ⨾ ⊞-assocL) ⨾ (⊞-assocL ⊕Id) {D}
  ⊞-assocL-pentagon₀ {A} {B} {C} {D} = ≈-begin
              ⊞-assocL ⨾ ⊞-assocL
      ≈⟨ ⨹-⨾ ⟩
              (ι ⨾ ι) ⨾ ⊞-assocL ⨹ (κ ⊕Id) ⨾ ⊞-assocL
      ≈⟨ ⨹-cong (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ι-⊞-assocL) ⊕Id-⨾-⊞-assocL ⟩
              ι ⨾ ι ⨾ ι ⨹ ⊞-assocL ⨾ ((κ ⊕Id) ⊕Id)
      ≈˘⟨      ⨹-⨾ ⟨≈≈⟩ ⨹-cong (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ι⨾⊕Id ⟨≈≈⟩ ⨾-cong₁&₂₁ ι-⊞-assocL)
                                (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ (⨾-⊕Id ⟨≈˘≈⟩ ⊕Id-cong κ⨾⨹)) ⟩
              (ι ⨾ ι ⨹ ⊞-assocL ⨾ (κ ⊕Id)) ⨾ ⊞-assocL ⊕Id
      ≈˘⟨ ⨾-cong₁ Id⊕-⨾-⨹ ⟩
              (Id⊕ ⊞-assocL ⨾ ⊞-assocL) ⨾ ⊞-assocL ⊕Id
      □


  ⊞-swap-monoidal₀ : {A B C : Obj}
      →     Id⊕ {A} (⊞-swap {B} {C}) ⨾ ⊞-assocL {A} {C} {B} ⨾ ((⊞-swap {A} {C}) ⊕Id) {B}
      ≈     ⊞-assocL {A} {B} {C} ⨾ ⊞-swap {A ⊞ B} {C} ⨾ ⊞-assocL {C} {A} {B}
  ⊞-swap-monoidal₀ = ≈-begin
            (Id⊕ ⊞-swap) ⨾ ⊞-assocL ⨾ (⊞-swap ⊕Id)
      ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ Id⊕-⨾-⨹ ⟩
            (ι ⨾ ι ⨹ ⊞-swap ⨾ (κ ⊕Id)) ⨾ (⊞-swap ⊕Id)
      ≈⟨ ⨹-⨾ ⟨≈≈⟩ ⨹-cong (⨾-cong₁₂&₂ ι⨾⊕Id ⟨≈≈⟩ ⨾-cong₁ ι⨾⨹)
                            (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ (⨾-⊕Id ⟨≈˘≈⟩ ⊕Id-cong κ⨾⨹) ⟨≈≈⟩ ⊞-swap-⨾-⨹) ⟩
         (κ ⨾ ι) ⨹ (κ ⨹ ι ⨾ ι)
      ≈˘⟨ ⨹-⨾ ⟨≈≈⟩ ⨹-cong (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ι⨾⨹ ⟨≈≈⟩ ι⨾⊕Id) (⊕Id-⨾-⨹ ⟨≈≈⟩ ⨹-cong₁ κ⨾⊕Id) ⟩
            ⊞-assocL ⨾ (κ ⊕Id ⨹ ι ⨾ ι)
      ≈˘⟨ ⨾-cong₂ ⊞-swap-⨾-⨹ ⟩
            ⊞-assocL ⨾ ⊞-swap ⨾ ⊞-assocL
      □

  ⊞-swap-monoidal˘₀ : {A B C : Obj}
      →     (((⊞-swap {A} {B}) ⊕Id) {C} ⨾ ⊞-assoc {B} {A} {C}) ⨾ Id⊕ {B} (⊞-swap {A} {C})
      ≈     (⊞-assoc {A} {B} {C} ⨾ ⊞-swap {A} {B ⊞ C}) ⨾ ⊞-assoc {B} {C} {A}
  ⊞-swap-monoidal˘₀ = ≈-begin
            (⊞-swap ⊕Id ⨾ ⊞-assoc) ⨾ Id⊕ ⊞-swap
      ≈⟨ ⨾-cong₁ ⊕Id-⨾-⨹ ⟩
            (⊞-swap ⨾ Id⊕ ι ⨹ κ ⨾ κ) ⨾ Id⊕ ⊞-swap
      ≈⟨ ⨹-⨾ ⟨≈≈⟩ ⨹-cong (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ (Id⊕-⨾ ⟨≈˘≈⟩ Id⊕-cong ι⨾⨹) ⟨≈≈⟩ ⊞-swap-⨾-⨹)
                            (⨾-cong₁₂&₂ κ⨾Id⊕ ⟨≈≈⟩ ⨾-cong₁ κ⨾⨹) ⟩
            (κ ⨾ κ ⨹ ι) ⨹ ι ⨾ κ
      ≈˘⟨ ⊞-assoc-⨹⨹ ⟩
            ⊞-assoc ⨾ (κ ⨾ κ ⨹ Id⊕ ι)
      ≈˘⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⊞-swap-⨾-⨹ ⟩
            (⊞-assoc ⨾ ⊞-swap) ⨾ ⊞-assoc
      □


  -- module HasCoproductsProps where
    -- open HasCoproductsLocalProps1 public
    -- open HasCoproductsLocalProps2 public
    -- open HasCoproductsUniversalProps public


open HasCoproductsProps public
```

Renamings

```
module IsCoproduct₁ {A B S : Obj} {ι : Mor A S} {κ : Mor B S} (isCoproduct : IsCoproduct ι κ)
  where
  open IsCoproduct isCoproduct public renaming
    ( _⨹_ to   _⨹₁_; ι⨾⨹ to ι⨾⨹₁; κ⨾⨹ to κ⨾⨹₁
    ; ⨹-unique          to ⨹₁-unique
    ; ⨹-universal       to ⨹₁-universal
    ; ⨹-factors         to ⨹₁-factors
    ; ⨹-universal-∃!    to ⨹₁-universal-∃!
    ; ⨹-cong to ⨹₁-cong; ⨹-cong₁ to ⨹₁-cong₁; ⨹-cong₂ to ⨹₁-cong₂
    ; ⨹-⨾ to ⨹₁-⨾; to-⨹ to to-⨹₁
    ; Id⊞ to Id⊞₁; Id⊞-isLeftIdentity to Id⊞₁-isLeftIdentity
    ; ⊞-IsInitial to ⊞₁-IsInitial; ⦶≈⊞ to ⦶≈⊞₁)
```

```
module IsCoproduct₂ {A B S : Obj} {ι : Mor A S} {κ : Mor B S} (isCoproduct : IsCoproduct ι κ)
  where
  open IsCoproduct isCoproduct public renaming
    ( _⨹_ to   _⨹₂_; ι⨾⨹ to ι⨾⨹₂; κ⨾⨹ to κ⨾⨹₂
    ; ⨹-unique          to ⨹₂-unique
    ; ⨹-universal       to ⨹₂-universal
    ; ⨹-factors         to ⨹₂-factors
    ; ⨹-universal-∃!    to ⨹₂-universal-∃!
    ; ⨹-cong to ⨹₂-cong; ⨹-cong₁ to ⨹₂-cong₁; ⨹-cong₂ to ⨹₂-cong₂
    ; ⨹-⨾ to ⨹₂-⨾; to-⨹ to to-⨹₂
    ; Id⊞ to Id⊞₂; Id⊞-isLeftIdentity to Id⊞₂-isLeftIdentity
    ; ⊞-IsInitial to ⊞₂-IsInitial; ⦶≈⊞ to ⦶≈⊞₂)
```

```
module IsCoproduct₃ {A B S : Obj} {ι : Mor A S} {κ : Mor B S} (isCoproduct : IsCoproduct ι κ)
  where
  open IsCoproduct isCoproduct public renaming
    ( _⨹_ to   _⨹₃_; ι⨾⨹ to ι⨾⨹₃; κ⨾⨹ to κ⨾⨹₃
    ; ⨹-unique          to ⨹₃-unique
    ; ⨹-universal       to ⨹₃-universal
    ; ⨹-factors         to ⨹₃-factors
    ; ⨹-universal-∃!    to ⨹₃-universal-∃!
    ; ⨹-cong to ⨹₃-cong; ⨹-cong₁ to ⨹₃-cong₁; ⨹-cong₂ to ⨹₃-cong₂
    ; ⨹-⨾ to ⨹₃-⨾; to-⨹ to to-⨹₃
    ; Id⊞ to Id⊞₃; Id⊞-isLeftIdentity to Id⊞₃-isLeftIdentity
    ; ⊞-IsInitial to ⊞₃-IsInitial; ⦶≈⊞ to ⦶≈⊞₃)
```

For renaming HasCoproductsLocalProps, we attach the suffix to the first "stand-alone" occurrence of ⊕ or ⊞ in the name, if any, otherwise to the first occurrence.

```
module HasCoproductsLocalProps₂ (hasCoprod : HasCoproducts) where
    -- open HasCoproducts.HasCoproductsLocalProps1 hasCoprod public renaming
  open HasCoproducts.HasCoproductsProps hasCoprod public using () renaming
    (_⊕_                      to _⊕₂_
    ; ι⨾⊕                     to ι⨾⊕₂
    ; κ⨾⊕                     to κ⨾⊕₂
    ; ι⨾ι⨾⊕⊕                  to ι⨾ι⨾⊕₂⊕
    ; κ⨾ι⨾⊕⊕                  to κ⨾ι⨾⊕₂⊕
    ; ι⨾κ⨾⊕⊕                  to ι⨾κ⨾⊕₂⊕
    ; κ⨾κ⨾⊕⊕                  to κ⨾κ⨾⊕₂⊕
    ; ιι⨾⊕⊕                   to ιι⨾⊕₂⊕
    ; κι⨾⊕⊕                   to κι⨾⊕₂⊕
    ; ικ⨾⊕⊕                   to ικ⨾⊕₂⊕
```

```
; κκ⨾⊕⊕                    to κκ⨾⊕₂⊕
; ⊕-cong                   to ⊕₂-cong
; ⊕-cong₁                  to ⊕₂-cong₁
; ⊕-cong₂                  to ⊕₂-cong₂
; ⊕-PreservesMonos         to ⊕₂-PreservesMonos
; _⊕Id                     to _⊕₂Id
; Id⊕                      to Id⊕₂
; ι⨾⊕Id                    to ι⨾⊕₂Id
; κ⨾⊕Id                    to κ⨾⊕₂Id
; ι⨾Id⊕                    to ι⨾Id⊕₂
; κ⨾Id⊕                    to κ⨾Id⊕₂
; ⊕Id-cong                 to ⊕₂Id-cong
; Id⊕-cong                 to Id⊕₂-cong
; ⊞-swap                   to ⊞₂-swap
; ⊞-assoc                  to ⊞₂-assoc
; κ-⊞-assoc                to κ-⊞₂-assoc
; ⊞-assocL                 to ⊞₂-assocL
; ι-⊞-assocL               to ι-⊞₂-assocL
; ⊞-transpose₂             to ⊞₂-transpose₂
  -- )
-- open HasCoproducts.HasCoproductsLocalProps2 hasCoprod public renaming
  -- (
; ⊕-⨾-Id⊞                  to ⊕₂-⨾-Id⊞
; ⨾-⊕-⨾                    to ⨾-⊕₂-⨾
; ⊕Id-⨾-⊕                  to ⊕Id-⨾-⊕₂
; ⊕-⨾-⊕Id                  to ⊕₂-⨾-⊕Id
; Id⊕-⨾-⊕                  to Id⊕-⨾-⊕₂
; ⊕-⨾-Id⊕                  to ⊕₂-⨾-Id⊕
; ⊕Id-⨾-Id⊕                to ⊕₂Id-⨾-Id⊕
; Id⊕-⨾-⊕Id                to Id⊕₂-⨾-⊕Id
; ⨾-⊕Id                    to ⨾-⊕₂Id
; Id⊕-⨾                    to Id⊕₂-⨾
; Id⊞-⊕Id                  to Id⊞₂-⊕Id
; Id⊕-Id⊞                  to Id⊕₂-Id⊞
; ⊕-⨾-⊞-swap               to ⊕₂-⨾-⊞-swap
; ⊞-swap-⨾-⊕               to ⊞₂-swap-⨾-⊕
; ⊕Id-⨾-⊞-swap             to ⊕₂Id-⨾-⊞-swap
; Id⊕-⨾-⊞-swap             to Id⊕₂-⨾-⊞-swap
; κ-ι-⊞-assoc              to κ-ι-⊞₂-assoc
; ι-ι-⊞-assoc              to ι-ι-⊞₂-assoc
; κι-⊞-assoc               to κι-⊞₂-assoc
; ιι-⊞-assoc               to ιι-⊞₂-assoc
; ι-κ-⊞-assocL             to ι-κ-⊞₂-assocL
; κ-κ-⊞-assocL             to κ-κ-⊞₂-assocL
; ικ-⊞-assocL              to ικ-⊞₂-assocL
; κκ-⊞-assocL              to κκ-⊞₂-assocL
; ⊞-assoc-assocL           to ⊞₂-assoc-assocL
; ⊞-assocL-assoc           to ⊞₂-assocL-assoc
; ⨾-⊞-assoc                to ⨾-⊞₂-assoc
; ⨾-⊞-assocL               to ⨾-⊞₂-assocL
; ⊞-assocL-⨾               to ⊞₂-assocL-⨾
; Id⊕-⨾-⊞-assoc            to Id⊕-⨾-⊞₂-assoc
; ⊕Id-⨾-⊞-assocL           to ⊕Id-⨾-⊞₂-assocL
; ⊞-₂₂assoc₁₂₁             to ⊞₂-₂₂assoc₁₂₁
; ⊞-₂₂assoc₁₂₁-split       to ⊞₂-₂₂assoc₁₂₁-split
; ⊞-₁₂₁assoc₂₂             to ⊞₂-₁₂₁assoc₂₂
; ⊞-₁₂₁assoc₂₂-split       to ⊞₂-₁₂₁assoc₂₂-split
; ⊞-₂₂assoc₁₂₁-₁₂₁assoc₂₂  to ⊞₂-₂₂assoc₁₂₁-₁₂₁assoc₂₂
; ⊞-₁₂₁assoc₂₂-₂₂assoc₁₂₁  to ⊞₂-₁₂₁assoc₂₂-₂₂assoc₁₂₁
```

```
    ; ⊞-assoc-pentagon₀            to ⊞₂-assoc-pentagon₀
    ; ⊞-assocL-pentagon₀           to ⊞₂-assocL-pentagon₀
    ; ⊞-swap-monoidal₀             to ⊞₂-swap-monoidal₀
    ; ⊞-swap-monoidal˘₀            to ⊞₂-swap-monoidal˘₀
    )

module HasCoproducts₁ (hasCoprod : HasCoproducts) where
  open HasCoproducts hasCoprod
  module _ {A B : Obj} where
    isCoproduct₁ = isCoproduct {A} {B}
    open IsCoproduct₁ isCoproduct₁ public
  infixr 3 _⊞₁_
  _⊞₁_ : Obj → Obj → Obj
  _⊞₁_ = _⊞_
  ι₁ : {A B : Obj} → Mor A (A ⊞ B)
  ι₁ = ι
  κ₁ : {A B : Obj} → Mor B (A ⊞ B)
  κ₁ = κ
  module 𝒫₁ = HasCoproductsProps
  infixr 5 _⊕₁_
  _⊕₁_ : {A B C D : Obj} (F : Mor A C) (G : Mor B D) → Mor (A ⊞ B) (C ⊞ D)
  _⊕₁_ = _⊕_
  infix 10 _⊕₁Id
  _⊕₁Id : {A₁ A₂ : Obj} (F : Mor A₁ A₂) {B : Obj} → Mor (A₁ ⊞ B) (A₂ ⊞ B)
  _⊕₁Id = _⊕Id

module HasCoproducts₂ (hasCoprod : HasCoproducts) where
  open HasCoproducts hasCoprod
  module _ {A B : Obj} where
    isCoproduct₂ = isCoproduct {A} {B}
    open IsCoproduct₂ isCoproduct₂ public
  infixr 3 _⊞₂_
  _⊞₂_ : Obj → Obj → Obj
  _⊞₂_ = _⊞_
  ι₂ : {A B : Obj} → Mor A (A ⊞ B)
  ι₂ = ι
  κ₂ : {A B : Obj} → Mor B (A ⊞ B)
  κ₂ = κ
  module 𝒫₂ = HasCoproductsProps
  infixr 5 _⊕₂_
  _⊕₂_ : {A B C D : Obj} (F : Mor A C) (G : Mor B D) → Mor (A ⊞ B) (C ⊞ D)
  _⊕₂_ = _⊕_
  infix 10 _⊕₂Id
  _⊕₂Id : {A₁ A₂ : Obj} (F : Mor A₁ A₂) {B : Obj} → Mor (A₁ ⊞ B) (A₂ ⊞ B)
  _⊕₂Id = _⊕Id

module HasCoproducts₃ (hasCoprod : HasCoproducts) where
  open HasCoproducts hasCoprod
  module _ {A B : Obj} where
    isCoproduct₃ = isCoproduct {A} {B}
    open IsCoproduct₃ isCoproduct₃ public
  infixr 3 _⊞₃_
  _⊞₃_ : Obj → Obj → Obj
  _⊞₃_ = _⊞_
  ι₃ : {A B : Obj} → Mor A (A ⊞ B)
  ι₃ = ι
  κ₃ : {A B : Obj} → Mor B (A ⊞ B)
```

```
  κ₃ = κ
  module 𝒫₃ = HasCoproductsProps
  infixr 5 _⊕₃_
  _⊕₃_ : {A B C D : Obj} (F : Mor A C) (G : Mor B D) → Mor (A ⊞ B) (C ⊞ D)
  _⊕₃_ = _⊕_
  infix 10 _⊕₃Id
  _⊕₃Id : {A₁ A₂ : Obj} (F : Mor A₁ A₂) {B : Obj} → Mor (A₁ ⊞ B) (A₂ ⊞ B)
  _⊕₃Id = _⊕Id
```

4.5 Categoric.FinColimits.Pushout

```
module Categoric.FinColimits.Pushout {i j k : Level} {Obj : Set i} {Hom : LocalSetoid Obj j k}
                                    (compOp : CompOp Hom) where
 open SemigroupoidCore compOp
 open  Categoric.FinColimits.CoCone2 compOp
```

Pushouts can easily be formulated as, given a span, unique co-spans that let the resulting square commute and satisfy a universal property. However, using Span and Cospan in the Agda formulation leads to friction, since Cospans in the oppositeCompOp are not directly recognised as Spans in the original CompOp. Therefore, we use a formulation that does not refer to Spans and Cospans.

```
 POUniversal :  {A B C D : Obj}
                (F : Mor A B) (G : Mor A C) (R : Mor B D) (S : Mor C D) → Set (i ⊍ j ⊍ k)
 POUniversal {A} {B} {C} {D} F G R S = {Z : Obj} {R′ : Mor B Z} {S′ : Mor C Z}
                                        (F⨾R′≈G⨾S′ : F ⨾ R′ ≈ G ⨾ S′)
                                      → CoCone2Univ R S R′ S′
 record IsPushout  {A B C D : Obj} (F : Mor A B) (G : Mor A C) (R : Mor B D) (S : Mor C D)
                   : Set (i ⊍ j ⊍ k) where
  field    commutes :  F ⨾ R ≈ G ⨾ S
           universal  :  POUniversal F G R S
  module _ {Z : Obj} {R′ : Mor B Z} {S′ : Mor C Z} (F⨾R′≈G⨾S′ : F ⨾ R′ ≈ G ⨾ S′) where
   open CoCone2Univ (universal F⨾R′≈G⨾S′) public
  module _ {Y : Obj} {R′ : Mor B Y} {S′ : Mor C Y} {F⨾R′≈G⨾S′ : F ⨾ R′ ≈ G ⨾ S′}
            {Z : Obj} {W : Mor Y Z} {V : Mor D Z} where
   private
    module Y-U = CoCone2Univ (universal F⨾R′≈G⨾S′)
    module Z-U = CoCone2Univ (universal (≈-begin
        F ⨾ (R′ ⨾ W)
      ≈⟨ ⨾-cong₁&₂₁ F⨾R′≈G⨾S′ ⟩
        G ⨾ (S′ ⨾ W)
      □))
   univMor⨾-unique  :  R ⨾ V ≈ R′ ⨾ W → S ⨾ V ≈ S′ ⨾ W → Y-U.univMor ⨾ W ≈ V
   univMor⨾-unique R⨾V≈R′⨾W S⨾V≈S′⨾W = ≈-sym (Z-U.univMor-unique′ R⨾V≈R′⨾W S⨾V≈S′⨾W
    (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ Y-U.univMor-factors-left)
    (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ Y-U.univMor-factors-right))


 record IsPushout-∃! {A B C D : Obj} (F : Mor A B) (G : Mor A C) (R : Mor B D) (S : Mor C D)
                    : Set (i ⊍ j ⊍ k) where
  field
   commutes   : F ⨾ R ≈ G ⨾ S
   universal    : {Z : Obj} {R′ : Mor B Z} {S′ : Mor C Z} → F ⨾ R′ ≈ G ⨾ S′
                → ∃! _≈_ (λ U → R ⨾ U ≈ R′ × S ⨾ U ≈ S′)
  universal-U   :  {Z : Obj} {R′ : Mor B Z} {S′ : Mor C Z} (F⨾R′≈G⨾S′ : F ⨾ R′ ≈ G ⨾ S′)
                → CoCone2Univ R S R′ S′
  universal-U F⨾R′≈G⨾S′ = CoCone2Univ-from-∃! (universal F⨾R′≈G⨾S′)
```

```
IsPushout-sym   : {A B C D : Obj} {F : Mor A B} {G : Mor A C} {R : Mor B D} {S : Mor C D}
                → IsPushout F G R S → IsPushout G F S R
IsPushout-sym   {F = F} {G} {R} {S} IsPO = let open IsPushout IsPO in record
  {commutes = ≈-sym commutes
  ;universal = λ G⨾R′≈F⨾S′ → CoCone2Univ-sym (universal (≈-sym G⨾R′≈F⨾S′))
  }
```

```
IsPushout-PreservesMono₁ IsPushout-PreservesMono₂
  IsPushout-PreservesEpi₁ IsPushout-PreservesEpi₂
    :   {A B C D : Obj} {F : Mor A B} {G : Mor A C} {R : Mor B D} {S : Mor C D}
    →   IsPushout F G R S → Set (i ⊍ j ⊍ k)
IsPushout-PreservesMono₁ {F = F} {G} {R} {S} IsPO = isMono F → isMono S
IsPushout-PreservesMono₂ {F = F} {G} {R} {S} IsPO = isMono G → isMono R
IsPushout-PreservesEpi₁  {F = F} {G} {R} {S} IsPO = isEpi  F → isEpi S
IsPushout-PreservesEpi₂  {F = F} {G} {R} {S} IsPO = isEpi  G → isEpi R
```

```
IsPushout-preservesEpi₁ : {A B C D : Obj} {F : Mor A B} {G : Mor A C} {R : Mor B D} {S : Mor C D}
                        → IsPushout F G R S → isEpi F → isEpi S
IsPushout-preservesEpi₁ {A} {B} {C} {D} {F} {G} {R} {S} isPO Fepi {Z} {T₁} {T₂} S⨾T₁≈S⨾T₂
   = univMor-unique′ {Z = Z} {R ⨾ T₂} {S ⨾ T₂} (⨾-cong₁&₂₁ commutes)
                     {V₁ = T₁} {T₂} (Fepi aux) S⨾T₁≈S⨾T₂ ≈-refl ≈-refl
  where
    open IsPushout isPO
    aux : F ⨾ R ⨾ T₁ ≈ F ⨾ R ⨾ T₂
    aux = ≈-begin
              F ⨾ R ⨾ T₁
          ≈⟨ ⨾-cong₁&₂₁ commutes ⟩
              G ⨾ S ⨾ T₁
          ≈⟨ ⨾-cong₂ S⨾T₁≈S⨾T₂ ⟩
              G ⨾ S ⨾ T₂
          ≈˘⟨ ⨾-cong₁&₂₁ commutes ⟩
              F ⨾ R ⨾ T₂
          □
```

```
IsPushout-preservesEpi₂ : {A B C D : Obj} {F : Mor A B} {G : Mor A C} {R : Mor B D} {S : Mor C D}
                        → IsPushout F G R S → isEpi G → isEpi R
IsPushout-preservesEpi₂ {A} {B} {C} {D} {F} {G} {R} {S} isPO Gepi {Z} {T₁} {T₂} R⨾T₁≈R⨾T₂
   = univMor-unique′ {Z = Z} {R ⨾ T₂} {S ⨾ T₂} (⨾-cong₁&₂₁ commutes)
                     {V₁ = T₁} {T₂} R⨾T₁≈R⨾T₂ (Gepi aux) ≈-refl ≈-refl
  where
    open IsPushout isPO
    aux : G ⨾ S ⨾ T₁ ≈ G ⨾ S ⨾ T₂
    aux = ≈-begin
              G ⨾ S ⨾ T₁
          ≈˘⟨ ⨾-cong₁&₂₁ commutes ⟩
              F ⨾ R ⨾ T₁
          ≈⟨ ⨾-cong₂ R⨾T₁≈R⨾T₂ ⟩
              F ⨾ R ⨾ T₂
          ≈⟨ ⨾-cong₁&₂₁ commutes ⟩
              G ⨾ S ⨾ T₂
          □
```

The lemmas IsPushout-compose and IsPushout-decompose both should be understood against the following diagram:

$$\begin{tikzcd}
A \arrow[r, "F_1"] \arrow[d, "G"'] \arrow[dr, phantom, "(1)"] & B \arrow[r, "F_2"] \arrow[d, "R_1"'] \arrow[dr, phantom, "(2)"] & E \arrow[d, "R_2"'] \\
C \arrow[r, "S_1"] & D \arrow[r, "S_2"] & F
\end{tikzcd}$$

```
module _ {A B C D E F : Obj}
         {F₁ : Mor A B} {F₂ : Mor B E} {G : Mor A C}
         {R₁ : Mor B D} {R₂ : Mor E F} {S₁ : Mor C D} {S₂ : Mor D F} where
```

Pushout composition: If (1) and (2) are pushouts, then (1)+(2) is also a pushout:

```
  IsPushout-compose : IsPushout F₁ G R₁ S₁ → IsPushout F₂ R₁ R₂ S₂
                    → IsPushout (F₁ ⨾ F₂) G R₂ (S₁ ⨾ S₂)
  IsPushout-compose isPO₁ isPO₂ = record
    {commutes = ≈-begin
        (F₁ ⨾ F₂) ⨾ R₂
      ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ isPO₂.commutes ⟩
        F₁ ⨾ R₁ ⨾ S₂
      ≈⟨ ⨾-cong₁&₂₁ isPO₁.commutes ⟩
        G ⨾ S₁ ⨾ S₂
      □
    ; universal = λ {Z} {R′} {S′} F₁⨾F₂⨾R′≈G⨾S′ → let
          open CoCone2Univ (isPO₁.universal {Z} {F₂ ⨾ R′} {S′} (⨾-assocL ⟨≈≈⟩ F₁⨾F₂⨾R′≈G⨾S′))
            using () renaming
              (univMor to S″
              ; univMor-factors-left to R₁⨾S″≈F₂⨾R′
              ; univMor-factors-right to S₁⨾S″≈S′
              ; univMor-unique to S″-unique
              )
          open CoCone2Univ (isPO₂.universal {Z} {R′} {S″} (≈-sym R₁⨾S″≈F₂⨾R′))
            using () renaming
              (univMor to U
              ; univMor-factors-left to R₂⨾U≈R′
              ; univMor-factors-right to S₂⨾U≈S″
              ; univMor-unique to U-unique
              )
       in record
       {univMor = U
       ; univMor-factors-left = R₂⨾U≈R′
       ; univMor-factors-right = ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ S₂⨾U≈S″ ⟨≈≈⟩ S₁⨾S″≈S′
       ; univMor-unique = λ {V} R₂⨾V≈R′ S₁⨾S₂⨾V≈S′
                        → U-unique R₂⨾V≈R′
                          (S″-unique (⨾-cong₁&₂₁ isPO₂.commutes ⟨≈˘≈⟩ ⨾-cong₂ R₂⨾V≈R′)
                                     (⨾-assocL ⟨≈≈⟩ S₁⨾S₂⨾V≈S′))
       }
    }
    where
      module isPO₁ = IsPushout isPO₁
      module isPO₂ = IsPushout isPO₂
```

Pushout decomposition: If (1) and (1)+(2) are pushouts and the diagram commutes, then (2) is also a pushout:

```
  IsPushout-decompose : IsPushout F₁ G R₁ S₁ → IsPushout (F₁ ⨾ F₂) G R₂ (S₁ ⨾ S₂)
                      → F₂ ⨾ R₂ ≈ R₁ ⨾ S₂ → IsPushout F₂ R₁ R₂ S₂
  IsPushout-decompose isPO₁ isPO₁₂ F₂⨾R₂≈R₁⨾S₂ = record
```

```
      {commutes = F₂⨾R₂≈R₁⨾S₂
      ; universal = λ {Z} {R′} {S′} F₂⨾R′≈R₁⨾S′ → let
           F₁⨾F₂⨾R′≈G⨾S₁⨾S′ : F₁ ⨾ F₂ ⨾ R′ ≈ G ⨾ S₁ ⨾ S′
           F₁⨾F₂⨾R′≈G⨾S₁⨾S′ = ⨾-cong₂ F₂⨾R′≈R₁⨾S′ ⟨≈≈⟩ ⨾-cong₁&₂₁ isPO₁.commutes
           open CoCone2Univ (isPO₁.universal {Z} {F₂ ⨾ R′} {S₁ ⨾ S′} F₁⨾F₂⨾R′≈G⨾S₁⨾S′)
             using () renaming (univMor-unique′ to S″-unique′)
           open CoCone2Univ (isPO₁₂.universal {Z} {R′} {S₁ ⨾ S′} (⨾-assoc ⟨≈≈⟩ F₁⨾F₂⨾R′≈G⨾S₁⨾S′))
             using () renaming (univMor                to U
                               ; univMor-factors-left  to R₂⨾U≈R′
                               ; univMor-factors-right to S₁⨾S₂⨾U≈S₁⨾S′
                               ; univMor-unique        to U-unique)
        in record
        {univMor = U
        ; univMor-factors-left = R₂⨾U≈R′
        ; univMor-factors-right = S″-unique′  (⨾-cong₁&₂₁ F₂⨾R₂≈R₁⨾S₂ ⟨≈˘≈⟩ ⨾-cong₂ R₂⨾U≈R′)
                                              (⨾-assocL ⟨≈≈⟩ S₁⨾S₂⨾U≈S₁⨾S′)
                                              (≈-sym F₂⨾R′≈R₁⨾S′) ≈-refl
        ; univMor-unique = λ {V} R₂⨾V≈R′ S₂⨾V≈S′ → U-unique R₂⨾V≈R′ (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ S₂⨾V≈S′)
        }
      }
    where
      module isPO₁ = IsPushout isPO₁
      module isPO₁₂ = IsPushout isPO₁₂

record Pushout {A B C : Obj} (F : Mor A B) (G : Mor A C) : Set (i ⊍ j ⊍ k) where
  field  {obj} : Obj
         left : Mor B obj
         right : Mor C obj
         prf : IsPushout F G left right
  open IsPushout prf public

Pushout-sym  :  {A B C : Obj} {F : Mor A B} {G : Mor A C}
                → Pushout F G → Pushout G F
Pushout-sym {F = F} {G} PO = let open Pushout PO in record
  {left = right
  ; right = left
  ; prf = IsPushout-sym prf
  }

Pushout-PreservesMono₁ Pushout-PreservesMono₂ Pushout-PreservesMonos
  Pushout-PreservesEpi₁ Pushout-PreservesEpi₂ Pushout-PreservesEpis
    : {A B C : Obj} {F : Mor A B} {G : Mor A C} → Pushout F G → Set (i ⊍ j ⊍ k)
Pushout-PreservesMono₁ PO = IsPushout-PreservesMono₁ (Pushout.prf PO)
Pushout-PreservesMono₂ PO = IsPushout-PreservesMono₂ (Pushout.prf PO)
Pushout-PreservesMonos PO = Pushout-PreservesMono₁ PO × Pushout-PreservesMono₂ PO
Pushout-PreservesEpi₁    PO = IsPushout-PreservesEpi₁    (Pushout.prf PO)
Pushout-PreservesEpi₂    PO = IsPushout-PreservesEpi₂    (Pushout.prf PO)
Pushout-PreservesEpis    PO = Pushout-PreservesEpi₁ PO × Pushout-PreservesEpi₂ PO

Pushouts-Preserve-Monos₁ Pushouts-Preserve-Monos₂ Pushouts-Preserve-Monos
  Pushouts-Preserve-Epis₁ Pushouts-Preserve-Epis₂ Pushouts-Preserve-Epis : Set (i ⊍ j ⊍ k)
Pushouts-Preserve-Monos₁ = {A B C : Obj} {F : Mor A B} {G : Mor A C}
                         → (PO : Pushout F G) → Pushout-PreservesMono₁ PO
Pushouts-Preserve-Monos₂ = {A B C : Obj} {F : Mor A B} {G : Mor A C}
                         → (PO : Pushout F G) → Pushout-PreservesMono₂ PO
Pushouts-Preserve-Monos  = {A B C : Obj} {F : Mor A B} {G : Mor A C}
                         → (PO : Pushout F G) → Pushout-PreservesMonos PO
```

```
Pushouts-Preserve-Epis₁    = {A B C : Obj} {F : Mor A B} {G : Mor A C}
                           → (PO : Pushout F G) → Pushout-PreservesEpi₁ PO
Pushouts-Preserve-Epis₂    = {A B C : Obj} {F : Mor A B} {G : Mor A C}
                           → (PO : Pushout F G) → Pushout-PreservesEpi₂ PO
Pushouts-Preserve-Epis     = {A B C : Obj} {F : Mor A B} {G : Mor A C}
                           → (PO : Pushout F G) → Pushout-PreservesEpis PO

pushouts-preserve-epis₁ : Pushouts-Preserve-Epis₁
pushouts-preserve-epis₁ PO = IsPushout-preservesEpi₁ (Pushout.prf PO)
pushouts-preserve-epis₂ : Pushouts-Preserve-Epis₂
pushouts-preserve-epis₂ PO = IsPushout-preservesEpi₂ (Pushout.prf PO)
pushouts-preserve-epis : Pushouts-Preserve-Epis
pushouts-preserve-epis PO = pushouts-preserve-epis₁ PO, pushouts-preserve-epis₂ PO

pushouts-preserve-monos-from-₁ : Pushouts-Preserve-Monos₁ → Pushouts-Preserve-Monos
pushouts-preserve-monos-from-₁ pm PO = pm PO, pm (Pushout-sym PO)
pushouts-preserve-monos-from-₂ : Pushouts-Preserve-Monos₂ → Pushouts-Preserve-Monos
pushouts-preserve-monos-from-₂ pm PO = pm (Pushout-sym PO), pm PO
pushouts-preserve-epis-from-₁ : Pushouts-Preserve-Epis₁ → Pushouts-Preserve-Epis
pushouts-preserve-epis-from-₁ pm PO = pm PO, pm (Pushout-sym PO)
pushouts-preserve-epis-from-₂ : Pushouts-Preserve-Epis₂ → Pushouts-Preserve-Epis
pushouts-preserve-epis-from-₂ pm PO = pm (Pushout-sym PO), pm PO

HasPushouts : Set (i ⊍ j ⊍ k)
HasPushouts = {A B C : Obj} (F : Mor A B) (G : Mor A C) → Pushout F G
```

Renamings

```
module IsPushout₁ {A B C D : Obj} {F : Mor A B} {G : Mor A C} {R : Mor B D} {S : Mor C D}
                  (IsPO : IsPushout F G R S) where
  open IsPushout IsPO public using () renaming
    (commutes to commutes₁; universal to universal₁
    ; univMor to univMor₁; univMor-factors to univMor₁-factors
    ; univMor-factors-left to univMor₁-factors-left
    ; univMor-factors-right to univMor₁-factors-right
    ; univMor-unique to univMor₁-unique
    ; universal-∃! to universal₁-∃!
    ; univMor⨾-unique to univMor₁⨾-unique
    )

module IsPushout₂ {A B C D : Obj} {F : Mor A B} {G : Mor A C} {R : Mor B D} {S : Mor C D}
                  (IsPO : IsPushout F G R S) where
  open IsPushout IsPO public using () renaming
    (commutes to commutes₂; universal to universal₂
    ; univMor to univMor₂; univMor-factors to univMor₂-factors
    ; univMor-factors-left to univMor₂-factors-left
    ; univMor-factors-right to univMor₂-factors-right
    ; univMor-unique to univMor₂-unique
    ; universal-∃! to universal₂-∃!
    ; univMor⨾-unique to univMor₂⨾-unique
    )

module Pushout₁ {A B C : Obj} {F : Mor A B} {G : Mor A C} (PO : Pushout F G) where
  open Pushout PO public using () renaming (obj to obj₁; left to left₁; right to right₁; prf to prf₁)
  open IsPushout₁ prf₁ public
```

```
  module Pushout₂ {A B C : Obj} {F : Mor A B} {G : Mor A C} (PO : Pushout F G) where
    open Pushout PO public using () renaming (obj to obj₂; left to left₂; right to right₂; prf to prf₂)
    open IsPushout₂ prf₂ public
```

4.6 Categoric.FinColimits.Pushout-Coproduct

Pushouts can be composed via _⊕_:

```
module Pushout-⊞ {i j k : Level} {Obj : Set i}
                 {Hom : LocalSetoid Obj j k} (compOp : CompOp Hom) where
  open SemigroupoidCore compOp
  open Categoric.FinColimits.CoCone2   compOp
  open Categoric.FinColimits.Coproduct compOp
  open Categoric.FinColimits.Pushout   compOp


  module _ (HasCoprod : HasCoproducts) where
    open HasCoproducts HasCoprod


    open IsPushout
    IsPushout-⊞ :  {A₁ A₂ B₁ B₂ C₁ C₂ D₁ D₂ : Obj}
                   {F₁ : Mor A₁ B₁} {G₁ : Mor A₁ C₁} {R₁ : Mor B₁ D₁} {S₁ : Mor C₁ D₁}
                   {F₂ : Mor A₂ B₂} {G₂ : Mor A₂ C₂} {R₂ : Mor B₂ D₂} {S₂ : Mor C₂ D₂}
                →  IsPushout F₁ G₁ R₁ S₁
                →  IsPushout F₂ G₂ R₂ S₂
                →  IsPushout (F₁ ⊕ F₂) (G₁ ⊕ G₂) (R₁ ⊕ R₂) (S₁ ⊕ S₂)
    IsPushout-⊞ {A₁} {A₂} {B₁} {B₂} {C₁} {C₂} {D₁} {D₂} {F₁} {G₁} {R₁} {S₁} {F₂} {G₂} {R₂} {S₂}
                isPO₁ isPO₂ =
      record
        {commutes = ≈-begin
                         (F₁ ⊕ F₂) ⨾ (R₁ ⊕ R₂)
                       ≈˘⟨ ⨾-⊕-⨾ ⟩
                         (F₁ ⨾ R₁) ⊕ (F₂ ⨾ R₂)
                       ≈⟨ ⊕-cong (commutes isPO₁) (commutes isPO₂) ⟩
                         (G₁ ⨾ S₁) ⊕ (G₂ ⨾ S₂)
                       ≈⟨ ⨾-⊕-⨾ ⟩
                         (G₁ ⊕ G₂) ⨾ (S₁ ⊕ S₂)
                       □
        ; universal = λ {Z} {R′} {S′} F⨾R′≈G⨾S′ →
          let comm-univMor₁ : F₁ ⨾ (ι ⨾ R′) ≈ G₁ ⨾ (ι ⨾ S′)
              comm-univMor₁ = ≈-begin
                  F₁ ⨾ (ι ⨾ R′)
                ≈⟨ ⨾-assocL ⟩
                  (F₁ ⨾ ι) ⨾ R′
                ≈˘⟨ ⨾-cong₁ ι⨾⊕ ⟩
                  (ι ⨾ (F₁ ⊕ F₂)) ⨾ R′
                ≈⟨ ⨾-cong₁₂&₂ F⨾R′≈G⨾S′ ⟩
                  (ι ⨾ (G₁ ⊕ G₂)) ⨾ S′
                ≈⟨ ⨾-cong₁ ι⨾⊕ ⟩
                  (G₁ ⨾ ι) ⨾ S′
                ≈⟨ ⨾-assoc ⟩
                  G₁ ⨾ (ι ⨾ S′)
                □
              comm-univMor₂ : F₂ ⨾ (κ ⨾ R′) ≈ G₂ ⨾ (κ ⨾ S′)
              comm-univMor₂ = ⨾-assocL ⟨≈≈˘⟩ ⨾-cong₁ κ⨾⊕ ⟨≈≈⟩
                ⨾-cong₁₂&₂ F⨾R′≈G⨾S′ ⟨≈≈⟩ ⨾-cong₁ κ⨾⊕ ⟨≈≈⟩ ⨾-assoc
```

```
      univMor-⊕ = univMor isPO₁ comm-univMor₁ ⨹ univMor isPO₂ comm-univMor₂
    in record
      {univMor = univMor-⊕
      ; univMor-factors-left =
          ≈-begin
            (R₁ ⊕ R₂) ⨾ univMor-⊕
          ≈⟨ ⊕-⨾-⨹ ⟩
            R₁ ⨾ (univMor isPO₁ comm-univMor₁)
            ⨹ R₂ ⨾ (univMor isPO₂ comm-univMor₂)
          ≈⟨ ⨹-cong (univMor-factors-left isPO₁ comm-univMor₁)
            (univMor-factors-left isPO₂ comm-univMor₂) ⟩
            ι ⨾ R′ ⨹ κ ⨾ R′
          ≈˘⟨ ⨹-unique ≈-refl ≈-refl ⟩
            R′
          □
      ; univMor-factors-right =
          ⊕-⨾-⨹ ⟨≈≈⟩
          ⨹-cong (univMor-factors-right isPO₁ comm-univMor₁)
            (univMor-factors-right isPO₂ comm-univMor₂) ⟨≈≈˘⟩
          ⨹-unique ≈-refl ≈-refl
      ; univMor-unique = λ {V} R⨾V≈R′ S⨾V≈S′ →
        let R₁⨾ι⨾V≈ι⨾R′ : R₁ ⨾ ι ⨾ V ≈ ι ⨾ R′
            R₁⨾ι⨾V≈ι⨾R′ = ⨾-assocL ⟨≈≈˘⟩ ⨾-cong₁ ι⨾⊕ ⟨≈≈⟩ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ R⨾V≈R′
            S₁⨾ι⨾V≈ι⨾S′ : S₁ ⨾ ι ⨾ V ≈ ι ⨾ S′
            S₁⨾ι⨾V≈ι⨾S′ = ⨾-assocL ⟨≈≈˘⟩ ⨾-cong₁ ι⨾⊕ ⟨≈≈⟩ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ S⨾V≈S′
            R₂⨾κ⨾V≈κ⨾R′ : R₂ ⨾ κ ⨾ V ≈ κ ⨾ R′
            R₂⨾κ⨾V≈κ⨾R′ = ⨾-assocL ⟨≈≈˘⟩ ⨾-cong₁ κ⨾⊕ ⟨≈≈⟩ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ R⨾V≈R′
            S₂⨾κ⨾V≈κ⨾S′ : S₂ ⨾ κ ⨾ V ≈ κ ⨾ S′
            S₂⨾κ⨾V≈κ⨾S′ = ⨾-assocL ⟨≈≈˘⟩ ⨾-cong₁ κ⨾⊕ ⟨≈≈⟩ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ S⨾V≈S′
        in
          ≈-begin
            V
          ≈⟨ ⨹-unique ≈-refl ≈-refl ⟩
            ι ⨾ V ⨹ κ ⨾ V
          ≈⟨ ⨹-cong (univMor-unique isPO₁ comm-univMor₁ R₁⨾ι⨾V≈ι⨾R′ S₁⨾ι⨾V≈ι⨾S′)
            (univMor-unique isPO₂ comm-univMor₂ R₂⨾κ⨾V≈κ⨾R′ S₂⨾κ⨾V≈κ⨾S′) ⟩
            univMor isPO₁ comm-univMor₁ ⨹ univMor isPO₂ comm-univMor₂
          ≈⟨ ≈-refl ⟩
            univMor-⊕
          □
        }
  }
```

4.7 Categoric.FinColimits

We now collect the modules for the individual kinds of finite colimits together into a single re-export, and add material that uses more than one kind of finite colimits.

```
module FinColimits {i j k : Level} {Obj : Set i}
                   {Hom : LocalSetoid Obj j k} (compOp : CompOp Hom) where
  open SemigroupoidCore compOp
  open Categoric.FinColimits.Initial       compOp public
  open Categoric.FinColimits.CoEqualiser   compOp public
  open Categoric.FinColimits.CoCone2       compOp public
  open Categoric.FinColimits.Coproduct     compOp public
  open Categoric.FinColimits.Pushout       compOp public
```

4.7.1 Pushout Construction From Coequalisers and Coproducts

If coequalisers are available, then, given a span $\mathsf{B} \xleftarrow{\mathsf{F}} \mathsf{A} \xrightarrow{\mathsf{G}} \mathsf{C}$, only a coproduct for B and C is required to construct a pushout:

```
constructPushout₁ : {A B C : Obj} (F : Mor A B) (G : Mor A C)
                  → {S : Obj} {ι : Mor B S} {κ : Mor C S} → IsCoproduct ι κ
                  → HasCoEqualisers
                  → Pushout F G
constructPushout₁ F G {S} {ι} {κ} isCoproduct HasCoEqu = let
    ce = HasCoEqualisers.coequaliser HasCoEqu (F ⨾ ι) (G ⨾ κ)
    open CoEqualiser ce using (obj; mor)
  in record
    {obj = obj
    ; left = ι ⨾ mor
    ; right = κ ⨾ mor
    ; prf = record
      {commutes = on-⨾-assocL (CoEqualiser.prop ce)
      ; universal = λ {Z} {P} {Q} F⨾P≈G⨾Q → let
          open CoCone2Univ (isCoproduct P Q) using () renaming
            (univMor                to V
            ; univMor-factors-left  to ι⨾V≈P
            ; univMor-factors-right to κ⨾V≈Q
            ; univMor-factors       to ι⨾V≈P, κ⨾V≈Q
            ; univMor-unique        to V-unique
            )
          ceu = CoEqualiser.universal ce {_} {V} (≈-begin
                (F ⨾ ι) ⨾ V
              ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ι⨾V≈P ⟩
                F ⨾ P
              ≈⟨ F⨾P≈G⨾Q ⟩
                G ⨾ Q
              ≈⟨ ⨾-cong₂ κ⨾V≈Q ⟨≈˘≈⟩ ⨾-assocL ⟩
                (G ⨾ κ) ⨾ V
              □)
          U = proj₁ ceu
          V≈m⨾U : V ≈ mor ⨾ U
          V≈m⨾U = proj₁ (proj₂ ceu)
          ι⨾m⨾U≈P = ⨾-assoc ⟨≈≈⟩ (⨾-cong₂ V≈m⨾U ⟨≈˘≈⟩ ι⨾V≈P)
          κ⨾m⨾U≈Q = ⨾-assoc ⟨≈≈⟩ (⨾-cong₂ V≈m⨾U ⟨≈˘≈⟩ κ⨾V≈Q)
        in record
          {univMor = U
          ; univMor-factors-left = ι⨾m⨾U≈P
          ; univMor-factors-right = κ⨾m⨾U≈Q
          ; univMor-unique = λ {U′} ι⨾m⨾U′≈P κ⨾m⨾U′≈Q → let
            V≈m⨾U′ : V ≈ mor ⨾ U′
            V≈m⨾U′ = ≈-sym (V-unique (⨾-assocL ⟨≈≈⟩ ι⨾m⨾U′≈P) (⨾-assocL ⟨≈≈⟩ κ⨾m⨾U′≈Q))
            in ≈-sym (proj₂ (proj₂ ceu) V≈m⨾U′)
          }
      }
    }
```

Therefore, if we have both coequalisers and coproducts, we have pushouts:

```
constructPushout : HasCoproducts → HasCoEqualisers → HasPushouts
constructPushout hasCoprod hasCoEqu {A} {B} {C} F G = let open HasCoproducts hasCoprod
  in constructPushout₁ F G {B ⊞ C} {ι} {κ} isCoproduct hasCoEqu
```

4.7.2 Coproducts are Pushouts of Initial Spans

```
①-Pushout-IsCoproduct : {① A B S : Obj} {ι : Mor A S} {κ : Mor B S}
                        → (isInitial : IsInitial ①) → let open IsInitial isInitial in
                          IsPushout ⓘ ⓘ ι κ → IsCoproduct ι κ
①-Pushout-IsCoproduct isInitial isPO {C} F G = IsPushout.universal isPO {C} {F} {G} ①≈
  where open IsInitial isInitial using (①≈)


Coproduct-is-①-Pushout : {① A B S : Obj} {ι : Mor A S} {κ : Mor B S}
                        → (isInitial : IsInitial ①) → let open IsInitial isInitial in
                          IsCoproduct ι κ → IsPushout ⓘ ⓘ ι κ
Coproduct-is-①-Pushout isInitial isCP = record
  {commutes = IsInitial.①≈ isInitial
  ; universal = λ {Z} {R} {S} _ → IsCoproduct.⚠-universal isCP R S
  }
```

4.7.3 Retractions

```
module FinColimitRetractions {i₁ j k : Level} {Obj₁ : Set i₁} {Hom₁ : LocalSetoid Obj₁ j k}
                             (compOp₁ : CompOp Hom₁) where
  open SemigroupoidCore compOp₁ using (Mor)
  open FinColimits
  module _ {i₂ : Level} {Obj₂ : Set i₂} (F : Obj₂ → Obj₁) where
    private
      compOp₂ = retractCompOp F compOp₁


retractIsInitial : {I : Obj₂} → IsInitial compOp₁ (F I) → IsInitial compOp₂ I
retractIsInitial FI-isInit {A} = FI-isInit {F A}


retractIsCoproduct : {A B S : Obj₂} {ι : Mor (F A) (F S)} {κ : Mor (F B) (F S)}
                   → IsCoproduct compOp₁ ι κ → IsCoproduct compOp₂ ι κ
retractIsCoproduct isCoproduct {Z} R S = let
    open CoCone2Univ compOp₁ (isCoproduct {F Z} R S)
  in record
    {univMor = univMor
    ; univMor-factors-left = univMor-factors-left
    ; univMor-factors-right = univMor-factors-right
    ; univMor-unique = univMor-unique
    }
```

4.8 Categoric.FinLimits

We obtain our material about finite limits via dualisation, that is, we use the material for finite colimits set in the opposite CompOp amd rename it appropriately. Some of the renamed modules, as for example Cone2Univ, directly use the inner names of the original modules (here CoCone2Univ) because those name have been chosen to be “direction-independent”. For other modules, as for example IsInitial, the inner names are tightly tied to nomenclature for initial objects, so we need to create replacement modules via appropriately renaming those inner names.

With large modules, especially if they also have local renamings, like $\mathsf{HasCoproducts}_1$, this approach becomes rather cumbersome; we are now experimenting with more use of qualified names (see the “$\mathcal{P}_1$ = HasCoproductsProps” there), while still trying to avoid qualified names for infix operators and some especially frequently-used identifiers.

We intentionally avoid "**using** ()" because in this way, when additional material is added to any FinColimits module, failing to add a renaming here will trigger a "duplicate definition" error message in modules that use both FinColimits and FinLimits.

```
module FinLimits {i j k : Level} {Obj : Set i} {Hom : LocalSetoid Obj j k} (compOp : CompOp Hom)
  where
  open LocalHomSetoid Hom
  open FinColimits (oppositeCompOp compOp) public
    hiding
      (module IsCoproduct
      ; module IsCoproduct₁
      ; module IsCoproduct₂
      ; module IsCoproduct₃
      ; HasCoproducts
      ; module HasCoproducts
      ; module HasCoproducts₁
      ; module HasCoproducts₂
      ; module HasCoproducts₃
      ; module HasCoproductsLocalProps₂
      ; module IsInitial
      ; module IsInitial₁
      ; module IsInitial₂
      ; module HasInitialObject
      ; module HasInitialObject₁
      ; module HasInitialObject₂
      ; module HasCoEqualisers
      ; constructPushout
      )
    renaming
      (CoEqualiser              to Equaliser
      ; module CoEqualiser      to Equaliser
      ; HasCoEqualisers         to HasEqualisers
      ; hasCoEqualisers         to hasEqualisers
      ; CoEqualiser-isEpi       to Equaliser-isMono
      ; IsInitial               to IsTerminal
      ; IsInitial≈              to IsTerminal≈
      ; IsInitial-SGIsoL        to IsTerminal-SGIsoL
      ; IsInitial-⨾-SGIsoL      to IsTerminal-⨾-SGIsoL
      ; HasInitialObject        to HasTerminalObject
      ; IsStrictInitialSG       to IsStrictTerminalSG   -- unlikely to be ever used
      ; CoCone2Univ             to Cone2Univ
      ; module CoCone2Univ      to Cone2Univ
      ; CoCone2Univ-from-∃!     to Cone2Univ-from-∃!
      ; CoCone2Univ-sym         to Cone2Univ-sym
      ; CoCone2Univ-congSrc     to Cone2Univ-congTrg
      ; IsColimit2              to IsLimit2
      ; module IsColimit2       to IsLimit2
      ; IsCoproduct             to IsProduct
      ; IsCoproduct-subst       to IsProduct-subst
      ; IsCoproduct-∃!          to IsProduct-∃!
      ; IsCoproduct-from-∃!     to IsProduct-from-∃!
      ; IsCoproduct-to-∃!       to IsProduct-to-∃!
      ; IsCoproduct-SGIsoL      to IsProduct-SGIsoL
      ; IsCoproduct-⨾-SGIsoL    to IsProduct-⨾-SGIsoL
      ; POUniversal             to PBUniversal
      ; IsPushout               to IsPullback
      ; module IsPushout        to IsPullback
      ; module IsPushout₁       to IsPullback₁
      ; module IsPushout₂       to IsPullback₂
```

```
    ; IsPushout-∃!                to IsPullback-∃!
    ; IsPushout-sym               to IsPullback-sym
    ; IsPushout-PreservesMono₁    to IsPullback-PreservesEpi₁
    ; IsPushout-PreservesMono₂    to IsPullback-PreservesEpi₂
    ; IsPushout-PreservesEpi₁     to IsPullback-PreservesMono₁
    ; IsPushout-PreservesEpi₂     to IsPullback-PreservesMono₂
    ; IsPushout-preservesEpi₁     to IsPullback-preservesMono₁
    ; IsPushout-preservesEpi₂     to IsPullback-preservesMono₂
    ; module IsPushout-∃!         to IsPullback-∃!
    ; IsPushout-compose           to IsPullback-compose
    ; IsPushout-decompose         to IsPullback-decompose
    ; Pushout                     to Pullback
    ; Pushout-sym                 to Pullback-sym
    ; module Pushout              to Pullback
    ; module Pushout₁             to Pullback₁
    ; module Pushout₂             to Pullback₂
    ; Pushout-PreservesMono₁      to Pullback-PreservesEpi₁
    ; Pushout-PreservesMono₂      to Pullback-PreservesEpi₂
    ; Pushout-PreservesMonos      to Pullback-PreservesEpis
    ; Pushout-PreservesEpi₁       to Pullback-PreservesMono₁
    ; Pushout-PreservesEpi₂       to Pullback-PreservesMono₂
    ; Pushout-PreservesEpis       to Pullback-PreservesMonos
    ; Pushouts-Preserve-Monos₁    to Pullbacks-Preserve-Epis₁
    ; Pushouts-Preserve-Monos₂    to Pullbacks-Preserve-Epis₂
    ; Pushouts-Preserve-Monos     to Pullbacks-Preserve-Epis
    ; Pushouts-Preserve-Epis₁     to Pullbacks-Preserve-Monos₁
    ; Pushouts-Preserve-Epis₂     to Pullbacks-Preserve-Monos₂
    ; Pushouts-Preserve-Epis      to Pullbacks-Preserve-Monos
    ; pushouts-preserve-epis₁     to pullbacks-preserve-monos₁
    ; pushouts-preserve-epis₂     to pullbacks-preserve-monos₂
    ; pushouts-preserve-epis      to pullbacks-preserve-monos
    ; pushouts-preserve-monos-from-₂ to pullbacks-preserve-epis-from-₂
    ; pushouts-preserve-monos-from-₁ to pullbacks-preserve-epis-from-₁
    ; pushouts-preserve-epis-from-₂ to pullbacks-preserve-monos-from-₂
    ; pushouts-preserve-epis-from-₁ to pullbacks-preserve-monos-from-₁
    ; constructPushout₁           to constructPullback₁
    ; HasPushouts                 to HasPullbacks
    ; Coproduct-is-Ⓘ-Pushout      to Product-is-Ⓣ-Pullback
    ; Ⓘ-Pushout-IsCoproduct       to Ⓣ-Pullback-IsProduct
    )
open FinColimitRetractions (oppositeCompOp compOp) public renaming
    (retractIsInitial     to retractIsTerminal
    ; retractIsCoproduct to retractIsProduct
    )
```

4.8.1 Equalisers

```
module HasEqualisers (H : HasEqualisers) where
  open FinColimits.HasCoEqualisers (oppositeCompOp compOp) H public renaming
    (coequ to equ
    ; _⇈º_ to _⇊º_
    ; _⇈_ to _⇊_
    ; _⨾⇈_ to _⇊⨾_
    ; ⇈-factoring to ⇊-factoring
    ; ⇈-factor to ⇊-factor
    ; ⇈-factors to ⇊-factors
    ; ⇈-factor-unique to ⇊-factor-unique
```

```
    ; coequaliser to equaliser
    )
```

4.8.2 Terminal Objects

```
module IsTerminal {Ⓣ : Obj} (isTerminal : IsTerminal Ⓣ) where
  open FinColimits.IsInitial (oppositeCompOp compOp) isTerminal public renaming
    (ⓘ to ⓣ
    ; ≈ⓘ to ≈ⓣ
    ; Ⓘ≈ to Ⓣ≈
    ; Ⓘ-Span to Ⓣ-Cospan
    )
module IsTerminal₁ {Ⓣ : Obj} (isTerminal : IsTerminal Ⓣ) where
  open IsTerminal isTerminal public
    renaming (ⓣ to ⓣ₁; ≈ⓣ to ≈ⓣ₁; Ⓣ≈ to Ⓣ₁≈; Ⓣ-Cospan to Ⓣ₁-Cospan)
module IsTerminal₂ {Ⓣ : Obj} (isTerminal : IsTerminal Ⓣ) where
  open IsTerminal isTerminal public
    renaming (ⓣ to ⓣ₂; ≈ⓣ to ≈ⓣ₂; Ⓣ≈ to Ⓣ₂≈; Ⓣ-Cospan to Ⓣ₂-Cospan)


module HasTerminalObject (H : HasTerminalObject) where
  open FinColimits.HasInitialObject (oppositeCompOp compOp) H public using () renaming
    (Ⓘ to Ⓣ; isInitial to isTerminal)
  open IsTerminal isTerminal public
module HasTerminalObject₁ (H : HasTerminalObject) where
  open HasTerminalObject H public using () renaming (Ⓣ to Ⓣ₁; isTerminal to isTerminal₁)
  open IsTerminal₁ isTerminal₁ public
module HasTerminalObject₂ (H : HasTerminalObject) where
  open HasTerminalObject H public using () renaming (Ⓣ to Ⓣ₂; isTerminal to isTerminal₂)
  open IsTerminal₂ isTerminal₂ public
```

4.8.3 Products

```
module IsProduct {A B P : Obj} {π : Mor P A} {ρ : Mor P B} (isProduct : IsProduct π ρ) where
  open FinColimits.IsCoproduct (oppositeCompOp compOp) isProduct public renaming
    (_⨹_                  to _∇_
    ; ι⨾⨹                 to ∇⨾π
    ; κ⨾⨹                 to ∇⨾ρ
    ; ⨹-unique            to ∇-unique
    ; ⨹-factors           to ∇-factors
    ; ⨹-universal         to ∇-universal
    ; ⨹-universal-∃!      to ∇-universal-∃!
    ; ⨹-cong              to ∇-cong
    ; ⨹-cong₁             to ∇-cong₁
    ; ⨹-cong₂             to ∇-cong₂
    ; ⨹-⨾                 to ⨾-∇
    ; to-⨹                to to-∇
    ; Id⊞                 to Id⊠
    ; Id⊞-isLeftIdentity  to Id⊠-isRightIdentity
    ; ⊞-IsInitial         to ⊠-isTerminal
    ; Ⓘ≈⊞                 to Ⓣ≈⊠
    )
module IsProduct₁ {A B P : Obj} {π : Mor P A} {ρ : Mor P B} (isProduct : IsProduct π ρ) where
  open IsProduct isProduct public renaming
    (_∇_ to _∇₁_; ∇⨾π to ∇₁⨾π; ∇⨾ρ to ∇₁⨾ρ
    ; ∇-unique to ∇₁-unique; ∇-factors to ∇₁-factors
```

```
      ; ∇-universal to ∇₁-universal; ∇-universal-∃! to ∇₁-universal-∃!
      ; ∇-cong to ∇₁-cong; ∇-cong₁ to ∇₁-cong₁; ∇-cong₂ to ∇₁-cong₂
      ; ⨾-∇ to ⨾-∇₁; to-∇ to to-∇₁
      ; Id⊠ to Id⊠₁; Id⊠-isRightIdentity to Id⊠₁-isRightIdentity
      ; ⊠-isTerminal to ⊠₁-isTerminal
      )
  module IsProduct₂ {A B P : Obj} {π : Mor P A} {ρ : Mor P B} (isProduct : IsProduct π ρ) where
    open IsProduct isProduct public renaming
      (_∇_ to _∇₂_; ∇⨾π to ∇₂⨾π₂; ∇⨾ρ to ∇₂⨾ρ₂
      ; ∇-unique to ∇₂-unique; ∇-factors to ∇₂-factors
      ; ∇-universal to ∇₂-universal; ∇-universal-∃! to ∇₂-universal-∃!
      ; ∇-cong to ∇₂-cong; ∇-cong₁ to ∇₂-cong₁; ∇-cong₂ to ∇₂-cong₂
      ; ⨾-∇ to ⨾-∇₂; to-∇ to to-∇₂
      ; Id⊠ to Id⊠₂; Id⊠-isRightIdentity to Id⊠₂-isRightIdentity
      ; ⊠-isTerminal to ⊠₂-isTerminal
      )
  module IsProduct₃ {A B P : Obj} {π : Mor P A} {ρ : Mor P B} (isProduct : IsProduct π ρ) where
    open IsProduct isProduct public renaming
      (_∇_ to _∇₃_; ∇⨾π to ∇₃⨾π₃; ∇⨾ρ to ∇₃⨾ρ₃
      ; ∇-unique to ∇₃-unique; ∇-factors to ∇₃-factors
      ; ∇-universal to ∇₃-universal; ∇-universal-∃! to ∇₃-universal-∃!
      ; ∇-cong to ∇₃-cong; ∇-cong₁ to ∇₃-cong₁; ∇-cong₂ to ∇₃-cong₂
      ; ⨾-∇ to ⨾-∇₃; to-∇ to to-∇₃
      ; Id⊠ to Id⊠₃; Id⊠-isRightIdentity to Id⊠₃-isRightIdentity
      ; ⊠-isTerminal to ⊠₃-isTerminal
      )
```

If we were to rename the record HasCoproducts to HasProducts, we would be inheriting the HasCoproducts field names, which makes explicit definitions of HasProducts records and related inferred types essentially unreadable. Therefore we define a separate **record** HasProducts, and provide the conversion opHasCoproducts for use in dualised contexts.

```
  record HasProducts : Set (i ⊍ j ⊍ k) where
    infixr 3 _⊠_
    field
      _⊠_ : Obj → Obj → Obj
      π : {A B : Obj} → Mor (A ⊠ B) A
      ρ : {A B : Obj} → Mor (A ⊠ B) B
      isProduct : {A B : Obj} → IsProduct {A} {B} π ρ
    module _ {A B : Obj} where
      open IsProduct (isProduct {A} {B}) public
    opHasCoproducts : FinColimits.HasCoproducts (oppositeCompOp compOp)
    opHasCoproducts = record {_⊞_ = _⊠_; ι = π; κ = ρ; isCoproduct = isProduct}


    module HasProductsProps where
      open FinColimits.HasCoproducts.HasCoproductsProps (oppositeCompOp compOp) opHasCoproducts
        public renaming
        (_⊕_            to _⊗_
        ; ι⨾⊕           to ⊗⨾π
        ; κ⨾⊕           to ⊗⨾ρ
        ; ι⨾ι⨾⊕⊕        to ⊗⊗⨾π⨾π
        ; κ⨾ι⨾⊕⊕        to ⊗⊗⨾π⨾ρ
        ; ι⨾κ⨾⊕⊕        to ⊗⊗⨾ρ⨾π
        ; κ⨾κ⨾⊕⊕        to ⊗⊗⨾ρ⨾ρ
        ; ιι⨾⊕⊕         to ⊗⊗⨾ππ
        ; κι⨾⊕⊕         to ⊗⊗⨾πρ
        ; ικ⨾⊕⊕         to ⊗⊗⨾ρπ
        ; κκ⨾⊕⊕         to ⊗⊗⨾ρρ
```

```
; ⊕-cong                to ⊗-cong
; ⊕-cong₁               to ⊗-cong₁
; ⊕-cong₂               to ⊗-cong₂
; ι⨾ι⨾⨹ ⨹               to ▽▽⨾π⨾π
; κ⨾ι⨾⨹ ⨹               to ▽▽⨾π⨾ρ
; ι⨾κ⨾⨹ ⨹               to ▽▽⨾ρ⨾π
; κ⨾κ⨾⨹ ⨹               to ▽▽⨾ρ⨾ρ
; ιι⨾⨹ ⨹                to ▽▽⨾ππ
; κι⨾⨹ ⨹                to ▽▽⨾πρ
; ικ⨾⨹ ⨹                to ▽▽⨾ρπ
; κκ⨾⨹ ⨹                to ▽▽⨾ρρ
; ⊕-⨾-⨹                 to ▽-⨾-⊗
; ⊕-⨾-Id⊞               to Id⊠-⨾-⊗
; ⨾-⊕-⨾                 to ⨾-⊗-⨾
; ⊕-PreservesMonos      to ⊗-PreservesEpis
; Id⊕                   to Id⊗
; _⊕Id                  to _⊗Id
; ι⨾⊕Id                 to ⊗Id⨾π
; κ⨾⊕Id                 to ⊗Id⨾ρ
; ι⨾Id⊕                 to Id⊗⨾π
; κ⨾Id⊕                 to Id⊗⨾ρ
; ⊕Id-cong              to ⊗Id-cong
; Id⊕-cong              to Id⊗-cong
; ⊕Id-⨾-⨹               to ▽-⨾-⊗Id
; Id⊕-⨾-⨹               to ▽-⨾-Id⊗
; ⊕Id-⨾-⊕               to ⊗-⨾-⊗Id
; ⊕-⨾-⊕Id               to ⊗Id-⨾-⊗
; Id⊕-⨾-⊕               to ⊗-⨾-Id⊗
; ⊕-⨾-Id⊕               to Id⊗-⨾-⊗
; ⊕Id-⨾-Id⊕             to Id⊗-⨾-⊗Id
; Id⊕-⨾-⊕Id             to ⊗Id-⨾-Id⊗
; ⨾-⊕Id                 to ⨾-⊗Id
; Id⊕-⨾                 to Id⊗-⨾
; Id⊞-⊕Id               to Id⊠-⊗Id
; Id⊕-Id⊞               to Id⊗-Id⊠
; ⊞-swap                to ⊠-swap
; ⊕-⨾-⊞-swap            to ⊠-swap-⨾-⊗
; ⊞-swap-⨾-⊕            to ⊗-⨾-⊠-swap
; ⊕Id-⨾-⊞-swap          to ⊠-swap-⨾-⊗Id
; Id⊕-⨾-⊞-swap          to ⊠-swap-⨾-Id⊗
; ⊞-swap-⨾-⨹            to ▽-⨾-⊠-swap
; ⊞-assoc               to ⊠-assocL
; ⊞-assocL              to ⊠-assoc
; κ-⊞-assoc             to ⊠-assocL-ρ
; κ-ι-⊞-assoc           to ⊠-assocL-π-ρ
; ι-ι-⊞-assoc           to ⊠-assocL-π-π
; κι-⊞-assoc            to ⊠-assocL-πρ
; ιι-⊞-assoc            to ⊠-assocL-ππ
; ι-⊞-assocL            to ⊠-assoc-π
; ι-κ-⊞-assocL          to ⊠-assoc-ρ-π
; κ-κ-⊞-assocL          to ⊠-assoc-ρ-ρ
; ικ-⊞-assocL           to ⊠-assoc-ρπ
; κκ-⊞-assocL           to ⊠-assoc-ρρ
; ⨾-⊞-assoc             to ⊠-assocL-⨾
; ⨾-⊞-assocL            to ⊠-assoc-⨾
; Id⊕-⨾-⊞-assoc         to ⊠-assocL-⨾-Id⊗
; ⊕Id-⨾-⊞-assocL        to ⊠-assoc-⨾-⊗Id
; ⊞-₂₂assoc₁₂₁          to ⊠-₁₂₁assoc₂₂
; ⊞-₁₂₁assoc₂₂          to ⊠-₂₂assoc₁₂₁
```

```
      ; ⊞-₂₂assoc₁₂₁-split          to ⊠-₁₂₁assoc₂₂-split
      ; ⊞-₁₂₁assoc₂₂-split          to ⊠-₂₂assoc₁₂₁-split
      ; ⊞-₂₂assoc₁₂₁-₁₂₁assoc₂₂ to ⊠-₂₂assoc₁₂₁-₁₂₁assoc₂₂
      ; ⊞-₁₂₁assoc₂₂-₂₂assoc₁₂₁ to ⊠-₁₂₁assoc₂₂-₂₂assoc₁₂₁
      ; ⊞-assoc-pentagon₀           to ⊠-assocL-pentagon₀
      ; ⊞-assocL-pentagon₀          to ⊠-assoc-pentagon₀
      ; ⊞-swap-monoidal₀            to ⊠-swap-monoidal₀
      ; ⊞-swap-monoidal˘₀           to Id⊗swap-assocL-swap⊗Id
      ; ⊞-assoc-assocL              to ⊠-assoc-assocL
      ; ⊞-assocL-assoc              to ⊠-assocL-assoc
      ; ⊞-assoc-⨹⨹                 to ∇∇-⊠-assocL
      ; ⊞-assocL-⨹⨹                to ∇∇-⊠-assoc
      ; ⊞-transpose₂                to ⊠-transpose₂
      ; ⊞-transpose₂-⨾              to ⨾-⊠-transpose₂
      ; ⊞-assoc-⨾                   to ⨾-⊠-assocL
      ; ⊞-assocL-⨾                  to ⨾-⊠-assoc
      ; ⊞-assoc-⨾-⊕⨹-⨾-⨹          to ∇-⨾-⊗∇-⨾-⊠-assocL
      )
  open HasProductsProps public
constructPullback : HasProducts → HasEqualisers → HasPullbacks
constructPullback hasProd hasEqu {A} {B} {C} F G = let open HasProducts hasProd
  in constructPullback₁ F G {B ⊠ C} {π} {ρ} isProduct hasEqu

module HasProducts₁ (hasProd : HasProducts) where
  open HasProducts hasProd
  module _ {A B : Obj} where
    open IsProduct₁ (isProduct {A} {B}) public
  infixr 3 _⊠₁_
  _⊠₁_ : Obj → Obj → Obj
  _⊠₁_ = _⊠_
  π₁ : {A B : Obj} → Mor (A ⊠ B) A
  π₁ = π
  ρ₁ : {A B : Obj} → Mor (A ⊠ B) B
  ρ₁ = ρ
  module 𝒫₁ = HasProductsProps
  infixr 5 _⊗₁_
  _⊗₁_ : {A B C D : Obj} (F : Mor A C) (G : Mor B D) → Mor (A ⊠ B) (C ⊠ D)
  _⊗₁_ = _⊗_
  infix 10 _⊗₁Id
  _⊗₁Id : {A₁ A₂ : Obj} (F : Mor A₁ A₂) {B : Obj} → Mor (A₁ ⊠ B) (A₂ ⊠ B)
  _⊗₁Id = _⊗Id
module HasProducts₂ (hasProd : HasProducts) where
  open HasProducts hasProd
  module _ {A B : Obj} where
    open IsProduct₂ (isProduct {A} {B}) public
  infixr 3 _⊠₂_
  _⊠₂_ : Obj → Obj → Obj
  _⊠₂_ = _⊠_
  π₂ : {A B : Obj} → Mor (A ⊠ B) A
  π₂ = π
  ρ₂ : {A B : Obj} → Mor (A ⊠ B) B
  ρ₂ = ρ
  module 𝒫₂ = HasProductsProps
  infixr 5 _⊗₂_
  _⊗₂_ : {A B C D : Obj} (F : Mor A C) (G : Mor B D) → Mor (A ⊠ B) (C ⊠ D)
  _⊗₂_ = _⊗_
  infix 10 _⊗₂Id
  _⊗₂Id : {A₁ A₂ : Obj} (F : Mor A₁ A₂) {B : Obj} → Mor (A₁ ⊠ B) (A₂ ⊠ B)
  _⊗₂Id = _⊗Id
```

```
  module HasProducts₃ (hasProd : HasProducts) where
    open HasProducts hasProd
    module _ {A B : Obj} where
      open IsProduct₃ (isProduct {A} {B}) public
    infixr 3 _⊠₃_
    _⊠₃_ : Obj → Obj → Obj
    _⊠₃_ = _⊠_
    π₃ : {A B : Obj} → Mor (A ⊠ B) A
    π₃ = π
    ρ₃ : {A B : Obj} → Mor (A ⊠ B) B
    ρ₃ = ρ
    module 𝒫₃ = HasProductsProps
    infixr 5 _⊗₃_
    _⊗₃_ : {A B C D : Obj} (F : Mor A C) (G : Mor B D) → Mor (A ⊠ B) (C ⊠ D)
    _⊗₃_ = _⊗_
    infix 10 _⊗₃Id
    _⊗₃Id : {A₁ A₂ : Obj} (F : Mor A₁ A₂) {B : Obj} → Mor (A₁ ⊠ B) (A₂ ⊠ B)
    _⊗₃Id = _⊗Id
```

4.9 Categoric.Semigroupoid.FinColimits

```
module Categoric.Semigroupoid.FinColimits {i j k : Level} {Obj : Set i}
                                          (SG : Semigroupoid j k Obj) where
  open Semigroupoid SG using (compOp)
  open FinColimits           compOp public
  open FinColimitRetractions compOp public
```

4.10 Categoric.Semigroupoid.FinLimits

```
module Categoric.Semigroupoid.FinLimits {i j k : Level} {Obj : Set i}
                                        (SG : Semigroupoid j k Obj) where
  open Semigroupoid SG using (compOp)
  open FinLimits compOp public
```

4.11 Categoric.Category.FinColimits

Here we collect additional properties of finite colimits in categories, in sub-modules that re-export their semigroupoid counterpart from Categoric.FinColimits, and are in turn provided by the collecting re-export CatFinColimits below in place of their semigroupoid counterparts.

```
module CatFinColimits₀ {i j k : Level} {Obj : Set i} (C : Category j k Obj) where
  open Category   C
  private module FinColimits = Categoric.Semigroupoid.FinColimits semigroupoid
  open FinColimits hiding
    (module HasCoproducts
    ; module HasCoproducts₁
    ; module HasCoproducts₂
    ; module HasCoproducts₃
    )
```

4.11.1 Initial Objects and Strict Initial Objects

Initial objects and direct sums are unique up to isomorphism, and are preserved by isomorphisms — this is not in Categoric.FinColimits since we rely on the convenient category-based definition of isomorphism here.

```
IsInitial-Iso        : {I₁ : Obj} → IsInitial I₁
                     → {I₂ : Obj} → IsInitial I₂
                     → Iso I₁ I₂
IsInitial-Iso {I₁} isInit₁ {I₂} isInit₂ = record
  {isoMor = proj₁ (isInit₁ {I₂})
  ; isIso = record
    {_⁻¹ = proj₁ (isInit₂ {I₁})
    ; rightInverse = IsInitial≈ isInit₁
    ; leftInverse = IsInitial≈ isInit₂
    }
  }
```

```
IsInitial-⨾-Iso        : {I₁ : Obj} → IsInitial I₁
                       → {I₂ : Obj} → Iso I₁ I₂
                       → IsInitial I₂
IsInitial-⨾-Iso {I₁} isInit₁ {I₂} F = F ⁻¹ ⨾ ⓘ, (λ V → ≈-begin
      V
  ≈˘⟨ ≈Id-isLeftIdentity (leftInverse F) ⟩
      (F ⁻¹ ⨾ isoMor F) ⨾ V
  ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ≈ⓘ ⟩
      F ⁻¹ ⨾ ⓘ
  □)
  where
    open HasInitialObject (record {ⓘ = I₁; isInitial = isInit₁})
```

Freyd and Scedrov (1990, 1.58) introduce a “strict coterminator” in a category as a “coterminator” (initial object) with the “extra property” that it is “an object 0 such that all morphisms targeted at 0 are isomorphisms”. (They introduce these in cartesian categories, where this “extra property” implies initiality.)

```
IsStrictInitial : (I : Obj) → Set (i ⊍ j ⊍ k)
IsStrictInitial I = {A : Obj} (f : Mor A I) → IsIso f
```

In general categories, IsStrictInitialSG from Categoric.FinColimits.Initial (Sect. 4.1) is equivalent to having both initiality and this “extra property”:

```
strictInitialSG-IsInitial : {I : Obj} → IsStrictInitialSG I → IsInitial I
strictInitialSG-IsInitial {I} I-isStrictInitSG = I-isStrictInitSG Id
strictInitialSG-IsStrictInitial : {I : Obj} → IsStrictInitialSG I → IsStrictInitial I
strictInitialSG-IsStrictInitial {I} I-isStrictInitSG {A} f = record
    {_⁻¹ = proj₁ (I-isStrictInitSG Id)
    ; rightInverse     = IsInitial≈ (I-isStrictInitSG f)
    ; leftInverse      = IsInitial≈ (I-isStrictInitSG Id)
    }
strictInitial-IsStrictInitialSG : {I : Obj} → IsInitial I → IsStrictInitial I → IsStrictInitialSG I
strictInitial-IsStrictInitialSG {I} I-isInit I-isStrictInit {A} f {B} = f ⨾ ⓘ, (λ V → ≈-begin
      V
  ≈˘⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ f⨾f⁻¹ ⟨≈≈⟩ leftId ⟩
      f ⨾ f⁻¹ ⨾ V
  ≈⟨ ⨾-cong₂ ≈ⓘ ⟩
      f ⨾ ⓘ
  □)
  where
```

```
    open IsIso (I-isStrictInit f) renaming (_⁻¹ to f⁻¹; rightInverse to f⨾f⁻¹)
    open IsInitial I-isInit
```

In general categories, IsStrictInitialSG from Categoric.FinColimits.Initial (Sect. 4.1) is equivalent to having both initiality and this "extra property":

```
module StrictInitial {I : Obj} (I-isInit : IsInitial I) (I-isStrictInit : IsStrictInitial I) where
  open IsInitial I-isInit using (Ⓘ; Ⓘ≈)
  Ⓘ≈strict : {A : Obj} {f g : Mor A I} → f ≈ g
  Ⓘ≈strict {A} {f} {g} = ≈-begin
      f
    ≈˘⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ f⨾f⁻¹ ⟨≈≈⟩ leftId ⟩
      f ⨾ f⁻¹ ⨾ f
    ≈⟨ ⨾-cong₂ Ⓘ≈ ⟩
      f ⨾ f⁻¹ ⨾ g
    ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ f⨾f⁻¹ ⟨≈≈⟩ leftId ⟩
      g
    □
    where
      open IsIso (I-isStrictInit f) using () renaming (_⁻¹ to f⁻¹; rightInverse to f⨾f⁻¹)
  toInitial-strict-≈ : {J : Obj} → IsInitial J → {A : Obj} {f g : Mor A J} → f ≈ g
  toInitial-strict-≈ {J} J-isInit {A} {f} {g} = ≈-begin
      f
    ≈˘⟨ ⨾-cong₂ J→I→J ⟨≈≈⟩ rightId ⟩
      f ⨾ J→I ⨾ I→J
    ≈⟨ ⨾-cong₁&₂₁ Ⓘ≈strict ⟩
      g ⨾ J→I ⨾ I→J
    ≈⟨ ⨾-cong₂ J→I→J ⟨≈≈⟩ rightId ⟩
      g
    □
    where
      open Iso (IsInitial-Iso I-isInit J-isInit) renaming
        (isoMor to I→J; _⁻¹ to J→I; leftInverse to J→I→J)
  Ⓘ→-isMono : {A : Obj} {f : Mor I A} → isMono f
  Ⓘ→-isMono {A} {Z} {g} {h} g⨾f≈h⨾f = Ⓘ≈strict
  fromInitial-isMono : {J A : Obj} {f : Mor J A} → IsInitial J → isMono f
  fromInitial-isMono J-isInit _ = toInitial-strict-≈ J-isInit
module StrictInitial′ (hasInit : HasInitialObject) (I-isStrictInit : IsStrictInitial (HasInitialObject.Ⓘ hasInit))
            = StrictInitial (HasInitialObject.isInitial hasInit) I-isStrictInit


record HasStrictInitialObject : Set (i ⊍ j ⊍ k) where
  field hasInit : HasInitialObject
        strictInit : IsStrictInitial (HasInitialObject.Ⓘ hasInit)
module HasStrictInitial (hasStrictInit : HasStrictInitialObject) where
  open HasStrictInitialObject hasStrictInit    public
  open HasInitialObject        hasInit          public
  open StrictInitial′          hasInit strictInit public
module HasStrictInitial₁ (hasStrictInit : HasStrictInitialObject) where
  open HasStrictInitialObject hasStrictInit        public renaming
    (hasInit to hasInit₁; strictInit to strictInit₁)
  open HasInitialObject₁       hasInit₁             public
  open StrictInitial′          hasInit₁ strictInit₁ public renaming
    (Ⓘ≈strict            to Ⓘ₁≈strict
    ; toInitial-strict-≈  to toInitial₁-strict-≈
    ; Ⓘ→-isMono          to Ⓘ₁→-isMono
    ; fromInitial-isMono to fromInitial₁-isMono
```

```
    )
module HasStrictInitial₂ (hasStrictInit : HasStrictInitialObject) where
  open HasStrictInitialObject hasStrictInit          public renaming
    (hasInit to hasInit₂; strictInit to strictInit₂)
  open HasInitialObject₂      hasInit₂                public
  open StrictInitial′         hasInit₂ strictInit₂    public renaming
    (⦷≈strict              to ⦷₂≈strict
    ; toInitial-strict-≈   to toInitial₂-strict-≈
    ; ⦷→-isMono            to ⦷₂→-isMono
    ; fromInitial-isMono   to fromInitial₂-isMono
    )
```

4.11.2 Binary Coproducts

```
  IsCoproduct-Iso   : {A B : Obj}
                    → {S₁ : Obj} {ι₁ : Mor A S₁} {κ₁ : Mor B S₁} → IsCoproduct ι₁ κ₁
                    → {S₂ : Obj} {ι₂ : Mor A S₂} {κ₂ : Mor B S₂} → IsCoproduct ι₂ κ₂
                    → Σ [ Φ : Iso S₁ S₂ ] ((ι₁ ⨾ isoMor Φ ≈ ι₂ × κ₁ ⨾ isoMor Φ ≈ κ₂)
                      × (ι₂ ⨾ Φ ⁻¹ ≈ ι₁ × κ₂ ⨾ Φ ⁻¹ ≈ κ₁))
  IsCoproduct-Iso {A} {B} {S₁} {ι₁} {κ₁} isCoproduct₁ {S₂} {ι₂} {κ₂} isCoproduct₂ = record
    {isoMor = U
    ; isIso = record
        {_⁻¹ = V
        ; rightInverse = U⨾V≈Id
        ; leftInverse = V⨾U≈Id
        }
    }, ⟁₁-factors, ⟁₂-factors
    where
      open IsCoproduct₁ isCoproduct₁
      open IsCoproduct₂ isCoproduct₂
      U : Mor S₁ S₂
      U = ι₂ ⟁₁ κ₂
      V : Mor S₂ S₁
      V = ι₁ ⟁₂ κ₁
      U⨾V≈Id : U ⨾ V ≈ Id
      U⨾V≈Id = ≈-begin
          U ⨾ V
        ≈⟨ ⟁₁-unique
            (≈-begin
                ι₁ ⨾ U ⨾ V
              ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ ι⨾⟁₁ ⟩
                ι₂ ⨾ V
              ≈⟨ ι⨾⟁₂ ⟩
                ι₁
              □)
            (≈-begin
                κ₁ ⨾ U ⨾ V
              ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ κ⨾⟁₁ ⟩
                κ₂ ⨾ V
              ≈⟨ κ⨾⟁₂ ⟩
                κ₁
              □)
          ⟩
          ι₁ ⟁₁ κ₁
        ≈˘⟨ ⟁₁-unique rightId rightId ⟩
          Id
        □
```

```
  V⨾U≈Id : V ⨾ U ≈ Id
  V⨾U≈Id = ≈-begin
      V ⨾ U
    ≈⟨ ⊕₂-unique
        (≈-begin
              ι₂ ⨾ V ⨾ U
            ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ ι⨾⊕₂ ⟩
              ι₁ ⨾ U
            ≈⟨ ι⨾⊕₁ ⟩
              ι₂
            □)
        (≈-begin
              κ₂ ⨾ V ⨾ U
            ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ κ⨾⊕₂ ⟩
              κ₁ ⨾ U
            ≈⟨ κ⨾⊕₁ ⟩
              κ₂
            □)
      ⟩
      ι₂ ⊕₂ κ₂
    ≈˘⟨ ⊕₂-unique rightId rightId ⟩
      Id
    □


IsCoproduct-⨾-Iso   : {A B : Obj}
                      → {S₁ : Obj} {ι₁ : Mor A S₁} {κ₁ : Mor B S₁} → IsCoproduct ι₁ κ₁
                      → {S₂ : Obj} → (Φ : Iso S₁ S₂)
                      → IsCoproduct (ι₁ ⨾ isoMor Φ) (κ₁ ⨾ isoMor Φ)
IsCoproduct-⨾-Iso {A} {B} {S₁} {ι₁} {κ₁} isCoproduct₁ {S₂} Φ {C} F G = record
    {univMor = Φ ⁻¹ ⨾ U₁
    ; univMor-factors-left = ≈-begin
          (ι₁ ⨾ isoMor Φ) ⨾ (Φ ⁻¹ ⨾ U₁)
        ≈⟨ ⨾-cong₁₂&₂₁ (rightInverse Φ) ⟩
          ι₁ ⨾ Id ⨾ U₁
        ≈⟨ ⨾-cong₂ leftId ⟨≈≈⟩ ι₁⨾U₁≈F ⟩
          F
        □
    ; univMor-factors-right = ≈-begin
          (κ₁ ⨾ isoMor Φ) ⨾ (Φ ⁻¹ ⨾ U₁)
        ≈⟨ ⨾-cong₁₂&₂₁ (rightInverse Φ) ⟩
          κ₁ ⨾ Id ⨾ U₁
        ≈⟨ ⨾-cong₂ leftId ⟨≈≈⟩ κ₁⨾U₁≈G ⟩
          G
        □
    ; univMor-unique = λ {V} ι₂⨾V≈F κ₂⨾V≈F → ≈-begin
              V
            ≈˘⟨ ⨾-assocL ⟨≈≈⟩ ≈Id-isLeftIdentity (leftInverse Φ) ⟩
              Φ ⁻¹ ⨾ isoMor Φ ⨾ V
            ≈⟨ ⨾-cong₂ (U₁-unique (⨾-assocL ⟨≈≈⟩ ι₂⨾V≈F) (⨾-assocL ⟨≈≈⟩ κ₂⨾V≈F)) ⟩
              Φ ⁻¹ ⨾ U₁
            □
    }
  where
    ι₂ : Mor A S₂
    ι₂ = ι₁ ⨾ isoMor Φ
    κ₂ : Mor B S₂
    κ₂ = κ₁ ⨾ isoMor Φ
    open CoCone2Univ (isCoproduct₁ F G) using () renaming
```

```
      (univMor to U₁
      ; univMor-factors-left to ι₁⨾U₁≈F
      ; univMor-factors-right to κ₁⨾U₁≈G
      ; univMor-unique to U₁-unique
      )


module HasCoproducts (HDS : HasCoproducts) where
  open FinColimits.HasCoproducts HDS public
  module HasCoproductsCatProps where


    Id⊞≈Id : {A B : Obj} → Id⊞ {A} {B} ≈ Id {A ⊞ B}
    Id⊞≈Id = ≈-begin
          Id⊞
      ≈˘⟨ rightId ⟩
          Id⊞ ⨾ Id
      ≈⟨ Id⊞-isLeftIdentity ⟩
          Id
      □

    ⊕Id-Id : {B A₁ A₂ : Obj} {F : Mor A₁ A₂} → (F ⊕Id) {B} ≈ F ⊕ Id
    ⊕Id-Id {F = F} = ≈-begin
          F ⊕Id
      ≈˘⟨ ⨹-cong₂ leftId ⟩
          F ⊕ Id
      □

    Id⊕-Id : {A B₁ B₂ : Obj} {G : Mor B₁ B₂} → Id⊕ {A} G ≈ Id ⊕ G
    Id⊕-Id {G = G} = ≈-begin
          Id⊕ G
      ≈˘⟨ ⨹-cong₁ leftId ⟩
          Id ⊕ G
      □

    ⊕-Id : {A B : Obj} → Id {A} ⊕ Id {B} ≈ Id {A ⊞ B}
    ⊕-Id {A} {B} = ≈-sym (⨹-unique
        (≈-begin
              ι ⨾ Id {A ⊞ B}
          ≈⟨ rightId ⟩
              ι
          ≈˘⟨ leftId ⟩
              Id ⨾ ι
          □)
        (≈-begin
              κ ⨾ Id {A ⊞ B}
          ≈⟨ rightId ⟩
              κ
          ≈˘⟨ leftId ⟩
              Id ⨾ κ
          □)
        )


    ⊞-swap² : {A B : Obj} → ⊞-swap {A} {B} ⨾ ⊞-swap ≈ Id
    ⊞-swap² = ≈-begin
                ⊞-swap ⨾ ⊞-swap
            ≈⟨ ⊞-swap-⨾-⨹ ⟩
                ι ⨹ κ
            ≈⟨ Id⊞≈Id ⟩
                Id
            □
```

```
⊞-swap-Iso : {A B : Obj} → Iso (A ⊞ B) (B ⊞ A)
⊞-swap-Iso = record {isoMor = ⊞-swap; isIso = record {_⁻¹ = ⊞-swap
                    ; rightInverse = ⊞-swap²; leftInverse = ⊞-swap²}}


⊞-join : {A : Obj} → Mor (A ⊞ A) A
⊞-join {A} = Id ⨹ Id

⊞-join-assoc    : {A : Obj}
                → (⊞-join {A} ⊕ Id {A}) ⨾ ⊞-join {A}
                ≈ ⊞-assoc {A} {A} {A} ⨾ (Id {A} ⊕ ⊞-join {A}) ⨾ ⊞-join {A}
⊞-join-assoc {A} = ≈-begin
                  (⊞-join ⊕ Id) ⨾ ⊞-join
                ≈⟨ ⊕-⨾-⨹ ⟨≈≈⟩ ⨹-cong rightId rightId ⟩
                  ⊞-join ⨹ Id
                ≈˘⟨ ⊞-assoc-⨹⨹ ⟩
                  ⊞-assoc ⨾ (Id ⨹ ⊞-join)
                ≈˘⟨ ⨾-cong₂ (⊕-⨾-⨹ ⟨≈≈⟩ ⨹-cong rightId rightId) ⟩
                  ⊞-assoc ⨾ (Id ⊕ ⊞-join) ⨾ ⊞-join
                □

⊞-swap-⨾-join : {A : Obj} → ⊞-swap ⨾ ⊞-join {A} ≈ ⊞-join
⊞-swap-⨾-join = ⨹-⨾ ⟨≈≈⟩ ⨹-cong κ⨾⨹ ι⨾⨹
```

The coproduct injections can easily been shown monic only for A ⊞ B where there are morphisms between A and B:

```
ι-isMono : {A B : Obj} → Mor B A → isMono (ι {A} {B})
ι-isMono X {Z} {R} {S} R⨾ι≈S⨾ι = ≈-begin
    R
  ≈⟨ rightId ⟨≈˘≈˘⟩ ⨾-cong₂ ι⨾⨹ ⟩
    R ⨾ ι ⨾ (Id ⨹ X)
  ≈⟨ ⨾-cong₁&₂₁ R⨾ι≈S⨾ι ⟩
    S ⨾ ι ⨾ (Id ⨹ X)
  ≈⟨ ⨾-cong₂ ι⨾⨹ ⟨≈≈⟩ rightId ⟩
    S
  □

κ-isMono : {A B : Obj} → Mor A B → isMono (κ {A} {B})
κ-isMono X {Z} {R} {S} R⨾κ≈S⨾κ = ≈-begin
    R
  ≈⟨ rightId ⟨≈˘≈˘⟩ ⨾-cong₂ κ⨾⨹ ⟩
    R ⨾ κ ⨾ (X ⨹ Id)
  ≈⟨ ⨾-cong₁&₂₁ R⨾κ≈S⨾κ ⟩
    S ⨾ κ ⨾ (X ⨹ Id)
  ≈⟨ ⨾-cong₂ κ⨾⨹ ⟨≈≈⟩ rightId ⟩
    S
  □
```

In general, coproduct injections do not need to be monic — for an example consider the opposite category of $\mathcal{S}et$, the category of sets and total functions: In $\mathcal{S}et$, one of the product projections of A × B is not surjective (i.e., epi) iff exactly one of A and B is the empty set.

```
⊞-assoc⨾⊞-assocL : {A B C : Obj} → ⊞-assoc {A} {B} {C} ⨾ ⊞-assocL {A} {B} {C} ≈ Id {(A ⊞ B) ⊞ C}
⊞-assoc⨾⊞-assocL = ⊞-assoc-assocL ⟨≈≈⟩ Id⊞≈Id
⊞-assocL⨾⊞-assoc : {A B C : Obj} → ⊞-assocL {A} {B} {C} ⨾ ⊞-assoc {A} {B} {C} ≈ Id {A ⊞ (B ⊞ C)}
⊞-assocL⨾⊞-assoc = ⊞-assocL-assoc ⟨≈≈⟩ Id⊞≈Id


⊞-assoc-Iso : {A B C : Obj} → Iso ((A ⊞ B) ⊞ C) (A ⊞ (B ⊞ C))
⊞-assoc-Iso = record
  {isoMor = ⊞-assoc
```

```
    ; isIso = record
      {_⁻¹ = ⊞-assocL
      ; rightInverse = ⊞-assoc⨾⊞-assocL
      ; leftInverse = ⊞-assocL⨾⊞-assoc
      }
    }
⊞-assocL-Iso : {A B C : Obj} → Iso (A ⊞ (B ⊞ C)) ((A ⊞ B) ⊞ C)
⊞-assocL-Iso = invIso ⊞-assoc-Iso


⊞-assoc-pentagon : {A B C D : Obj}
  → ⊞-assoc {A ⊞ B} {C} {D} ⨾ ⊞-assoc {A} {B} {C ⊞ D}
  ≈ (⊞-assoc {A} {B} {C} ⊕ Id {D}) ⨾ ⊞-assoc {A} {B ⊞ C} {D}
    ⨾ (Id {A} ⊕ ⊞-assoc {B} {C} {D})
⊞-assoc-pentagon {A} {B} {C} {D}
  = ⊞-assoc-pentagon₀ {A} {B} {C} {D} ⟨≈≈⟩ ⨾-cong (⊕Id-Id {D}) (⨾-cong₂ (Id⊕-Id {A}))


⊞-assocL-pentagon : {A B C D : Obj}
  → ⊞-assocL {A} {B} {C ⊞ D} ⨾ ⊞-assocL {A ⊞ B} {C} {D}
  ≈ ((Id {A} ⊕ ⊞-assocL) ⨾ ⊞-assocL) ⨾ (⊞-assocL ⊕ Id {D})
⊞-assocL-pentagon {A} {B} {C} {D}
  = ⊞-assocL-pentagon₀ {A} {B} {C} {D} ⟨≈≈⟩ ⨾-cong (⨾-cong₁ (Id⊕-Id {A})) (⊕Id-Id {D})


⊞-swap-monoidal : {A B C : Obj}
  → (Id {A} ⊕ ⊞-swap {B} {C}) ⨾ ⊞-assocL {A} {C} {B} ⨾ (⊞-swap {A} {C} ⊕ Id {B})
  ≈ ⊞-assocL {A} {B} {C} ⨾ ⊞-swap {A ⊞ B} {C} ⨾ ⊞-assocL {C} {A} {B}
⊞-swap-monoidal {A} {B} {C}
  = ⨾-cong (Id⊕-Id {A}) (⨾-cong₂ (⊕Id-Id {B})) ⟨≈˘≈⟩ ⊞-swap-monoidal₀ {A} {B} {C}
⊞-swap-monoidal˘ : {A B C : Obj}
  → ((⊞-swap {A} {B} ⊕ Id {C}) ⨾ ⊞-assoc {B} {A} {C}) ⨾ (Id {B} ⊕ ⊞-swap {A} {C})
  ≈ (⊞-assoc {A} {B} {C} ⨾ ⊞-swap {A} {B ⊞ C}) ⨾ ⊞-assoc {B} {C} {A}
⊞-swap-monoidal˘ {A} {B} {C}
  = ⨾-cong (⨾-cong₁ (⊕Id-Id {C})) (Id⊕-Id {B}) ⟨≈˘≈⟩ ⊞-swap-monoidal˘₀ {A} {B} {C}


⊕-⨾-I⚠I : {A B C : Obj} {F : Mor A C} {G : Mor B C} → (F ⊕ G) ⨾ (Id ⚠ Id) ≈ F ⚠ G
⊕-⨾-I⚠I {F = F} {G} = ⊕-⨾-⚠ ⟨≈≈⟩ ⚠-cong rightId rightId


I⚠I-assoc˘ : {A : Obj}
                    → ((Id ⚠ Id) ⊕ Id) ⨾ (Id ⚠ Id)
                      ≈ (⊞-assoc {A} {A} {A} ⨾ (Id ⊕ (Id ⚠ Id))) ⨾ (Id ⚠ Id)
I⚠I-assoc˘ = ⊞-assoc-⨾-⊕⚠-⨾-⚠ ⟨≈˘≈⟩ ⨾-assocL


I⚠I-monoidal˘ : {A B : Obj}
          → ((Id ⊕ ((⊞-swap ⊕ Id) ⨾ ⊞-assoc)) ⨾ ⊞-assocL {A} {B} {A ⊞ B}) ⨾ (Id ⚠ Id)
          ≈ ((Id ⊕ ⊞-assoc) ⨾ ⊞-assocL {A} {A} {B ⊞ B}) ⨾ ((Id ⚠ Id) ⊕ (Id ⚠ Id))
I⚠I-monoidal˘ = ≈-begin
          ((Id ⊕ ((⊞-swap ⊕ Id) ⨾ ⊞-assoc)) ⨾ ⊞-assocL) ⨾ (Id ⚠ Id)
      ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ (⨾-cong₂ (⚠-cong Id⊞≈Id Id⊞≈Id) ⟨≈˘≈⟩ ⊞-assocL-⚠⚠) ⟩
          (Id ⊕ ((⊞-swap ⊕ Id) ⨾ ⊞-assoc)) ⨾ (ι ⚠ (κ ⚠ Id⊞))
      ≈⟨ ⊕-⨾-⚠ ⟨≈≈⟩ ⚠-cong leftId (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⊞-assoc-⚠⚠) ⟩
          ι ⚠ ((⊞-swap ⊕ Id) ⨾ ((κ ⚠ ι) ⚠ κ))
      ≈⟨ ⚠-cong₂ (⊕-⨾-⚠ ⟨≈≈⟩ ⚠-cong ⊞-swap-⨾-⚠ leftId) ⟩
          ι ⚠ ((ι ⚠ κ) ⚠ κ)
      ≈˘⟨ ⊕-⨾-⚠ ⟨≈≈⟩ ⚠-cong leftId ⊞-assoc-⚠⚠ ⟩
          (Id ⊕ ⊞-assoc) ⨾ (ι ⚠ (ι ⚠ (κ ⚠ κ)))
      ≈˘⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⊞-assocL-⚠⚠ ⟩
```

```
        ((Id ⊕ ⊞-assoc) ⨾ ⊞-assocL) ⨾ ((ι ◬ ι) ◬ (κ ◬ κ))
    ≈˘⟨ ⨾-cong₂ (◬-cong (◬-⨾ ⟨≈≈⟩ ◬-cong leftId leftId) (◬-⨾ ⟨≈≈⟩ ◬-cong leftId leftId)) ⟩
        ((Id ⊕ ⊞-assoc) ⨾ ⊞-assocL) ⨾ ((Id ◬ Id) ⊕ (Id ◬ Id))
    □
```

```
⊞-join-monoidal  : {A B : Obj}
                 → (Id {A} ⊕ ((⊞-swap {A} {B} ⊕ Id {B}) ⨾ ⊞-assoc {B} {A} {B}))
                   ⨾ ⊞-assocL {A} {B} {A ⊞ B} ⨾ ⊞-join {A ⊞ B}
                 ≈ (Id {A} ⊕ ⊞-assoc {A} {B} {B})
                   ⨾ ⊞-assocL {A} {A} {B ⊞ B} ⨾ (⊞-join {A} ⊕ ⊞-join {B})
⊞-join-monoidal = ≈-begin
    (Id ⊕ ((⊞-swap ⊕ Id) ⨾ ⊞-assoc)) ⨾ ⊞-assocL ⨾ ⊞-join
  ≈⟨ ⨾-cong₂  (◬-⨾
               ⟨≈≈⟩ ◬-cong₂ ◬-⨾
               ⟨≈≈⟩ ◬-cong (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ι⨾◬ ⟨≈≈⟩ rightId)
                           (◬-cong (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ι⨾◬ ⟨≈≈⟩ rightId)
                                   κ⨾◬)) ⟩
    (Id ⊕ ((⊞-swap ⊕ Id) ⨾ ⊞-assoc)) ⨾ (ι ◬ (κ ◬ Id))
  ≈⟨ ⊕-⨾-◬ ⟩
    Id ⨾ ι ◬ ((⊞-swap ⊕ Id) ⨾ ⊞-assoc) ⨾ (κ ◬ Id)
  ≈⟨ ◬-cong₂ (⨾-cong₁ ⊕-⨾-◬) ⟩
    Id ⨾ ι ◬ (⊞-swap ⨾ (ι ◬ ι ⨾ κ) ◬ Id ⨾ (κ ⨾ κ)) ⨾ (κ ◬ Id)
  ≈⟨ ◬-cong₂ (⨾-cong₁ (◬-cong ⊞-swap-⨾-◬ leftId)) ⟩
    Id ⨾ ι ◬ ((ι ⨾ κ ◬ ι) ◬ κ ⨾ κ) ⨾ (κ ◬ Id)
  ≈⟨ ◬-cong₂ (◬-⨾ ⟨≈≈⟩ ◬-cong₁ ◬-⨾) ⟩
    Id ⨾ ι ◬ (((ι ⨾ κ) ⨾ (κ ◬ Id) ◬ ι ⨾ (κ ◬ Id)) ◬ (κ ⨾ κ) ⨾ (κ ◬ Id))
  ≈⟨ ◬-cong₂ (◬-cong (◬-cong₁ ⨾-assoc) ⨾-assoc) ⟩
    Id ⨾ ι ◬ ((ι ⨾ (κ ⨾ (κ ◬ Id)) ◬ ι ⨾ (κ ◬ Id)) ◬ κ ⨾ (κ ⨾ (κ ◬ Id)))
  ≈⟨ ◬-cong₂ (◬-cong (◬-cong (⨾-cong₂ κ⨾◬ ⟨≈≈⟩ rightId) ι⨾◬)
                     (⨾-cong₂ κ⨾◬ ⟨≈≈⟩ rightId)) ⟩
    Id ⨾ ι ◬ ((ι ◬ κ) ◬ κ)
  ≈˘⟨ ◬-cong leftId (◬-cong (◬-cong leftId leftId) leftId) ⟩
    Id ⨾ (Id ⨾ ι) ◬ ((Id ⨾ ι ◬ Id ⨾ κ) ◬ Id ⨾ κ)
  ≈˘⟨ ◬-cong₂ ⊞-assoc-◬◬ ⟩
    Id ⨾ (Id ⨾ ι) ◬ ⊞-assoc ⨾ (Id ⨾ ι ◬ (Id ⨾ κ ◬ Id ⨾ κ))
  ≈˘⟨ ⊕-⨾-◬ ⟩
    (Id ⊕ ⊞-assoc) ⨾ (Id ⨾ ι ◬ (Id ⨾ ι ◬ (Id ⨾ κ ◬ Id ⨾ κ)))
  ≈˘⟨ ⨾-cong₂ ⊞-assocL-◬◬ ⟩
    (Id ⊕ ⊞-assoc) ⨾ ⊞-assocL ⨾ ((Id ⨾ ι ◬ Id ⨾ ι) ◬ (Id ⨾ κ ◬ Id ⨾ κ))
  ≈˘⟨ ⨾-cong₂₂ (◬-cong ◬-⨾ ◬-⨾) ⟩
    (Id ⊕ ⊞-assoc) ⨾ ⊞-assocL ⨾ (⊞-join ⊕ ⊞-join)
  □
```

⊕join is used in Categoric.Monad.FunctorMonad.Coproduct.

```
⊕join : {A B : Obj} → Mor (A ⊞ (A ⊞ B)) (A ⊞ B)
⊕join {A} {B} = ι ◬ Id {A ⊞ B}
⊕join-contract : {A B : Obj} → ⊞-assocL {A} {A} {B} ⨾ ⊞-join ⊕Id ≈ ⊕join {A} {B}
⊕join-contract = ≈-begin
    ⊞-assocL ⨾ ⊞-join ⊕Id
  ≈⟨⟩
    (ι ⨾ ι ◬ (κ ⊕Id)) ⨾ ⊞-join ⊕Id
  ≈⟨ ◬-⨾ ⟨≈≈⟩ ◬-cong (⨾-cong₁₂&₂ ι⨾⊕Id ⟨≈≈⟩ ⨾-cong₁ ι⨾◬ ⟨≈≈⟩ leftId)
                     (⨾-⊕Id ⟨≈˘≈⟩ ⊕Id-cong κ⨾◬) ⟩
    ι ◬ Id ⊕Id
  ≈⟨ ◬-cong₂ (⊕Id-Id ⟨≈≈⟩ ⊕-Id) ⟩
    ι ◬ Id
  □
```

```
  open HasCoproductsCatProps public
```

Renamings

```
module HasCoproducts₁ (HDS : HasCoproducts) where
  open FinColimits.HasCoproducts₁ HDS public
  open HasCoproducts.HasCoproductsCatProps HDS public renaming
    (Id⊞≈Id               to Id⊞₁≈Id
    ; ⊕Id-Id              to ⊕₁Id-Id
    ; Id⊕-Id              to Id⊕₁-Id
    ; ⊕-Id                to ⊕₁-Id
    ; ⊞-swap²             to ⊞₁-swap²
    ; ⊞-swap-Iso          to ⊞₁-swap-Iso
    ; ⊞-join              to ⊞₁-join
    ; ⊞-join-assoc        to ⊞₁-join-assoc
    ; ⊞-swap-⨾-join       to ⊞₁-swap-⨾-join
    ; ι-isMono            to ι₁-isMono
    ; κ-isMono            to κ₁-isMono
    ; ⊞-assoc⨾⊞-assocL    to ⊞₁-assoc⨾⊞-assocL
    ; ⊞-assocL⨾⊞-assoc    to ⊞₁-assocL⨾⊞-assoc
    ; ⊞-assoc-Iso         to ⊞₁-assoc-Iso
    ; ⊞-assocL-Iso        to ⊞₁-assocL-Iso
    ; ⊞-assoc-pentagon    to ⊞₁-assoc-pentagon
    ; ⊞-assocL-pentagon   to ⊞₁-assocL-pentagon
    ; ⊞-swap-monoidal     to ⊞₁-swap-monoidal
    ; ⊞-swap-monoidal˘    to ⊞₁-swap-monoidal˘
    ; ⊕-⨾-I⚠I             to ⊕₁-⨾-I⚠I
    ; I⚠I-assoc˘          to I⚠₁I-assoc˘
    ; I⚠I-monoidal˘       to I⚠₁I-monoidal˘
    ; ⊞-join-monoidal     to ⊞₁-join-monoidal
    )

module HasCoproducts₂ (HDS : HasCoproducts) where
  open FinColimits.HasCoproducts₂ HDS public
  open HasCoproducts.HasCoproductsCatProps HDS public renaming
    (Id⊞≈Id               to Id⊞₂≈Id
    ; ⊕Id-Id              to ⊕₂Id-Id
    ; Id⊕-Id              to Id⊕₂-Id
    ; ⊕-Id                to ⊕₂-Id
    ; ⊞-swap²             to ⊞₂-swap²
    ; ⊞-swap-Iso          to ⊞₂-swap-Iso
    ; ⊞-join              to ⊞₂-join
    ; ⊞-join-assoc        to ⊞₂-join-assoc
    ; ⊞-swap-⨾-join       to ⊞₂-swap-⨾-join
    ; ι-isMono            to ι₂-isMono
    ; κ-isMono            to κ₂-isMono
    ; ⊞-assoc⨾⊞-assocL    to ⊞₂-assoc⨾⊞-assocL
    ; ⊞-assocL⨾⊞-assoc    to ⊞₂-assocL⨾⊞-assoc
    ; ⊞-assoc-Iso         to ⊞₂-assoc-Iso
    ; ⊞-assocL-Iso        to ⊞₂-assocL-Iso
    ; ⊞-assoc-pentagon    to ⊞₂-assoc-pentagon
    ; ⊞-assocL-pentagon   to ⊞₂-assocL-pentagon
    ; ⊞-swap-monoidal     to ⊞₂-swap-monoidal
    ; ⊞-swap-monoidal˘    to ⊞₂-swap-monoidal˘
    ; ⊕-⨾-I⚠I             to ⊕₂-⨾-I⚠I
    ; I⚠I-assoc˘          to I⚠₂I-assoc˘
    ; I⚠I-monoidal˘       to I⚠₂I-monoidal˘
```

```
    ; ⊞-join-monoidal        to ⊞₂-join-monoidal
    )

module HasCoproducts₃ (HDS : HasCoproducts) where
  open FinColimits.HasCoproducts₃ HDS public
  open HasCoproducts.HasCoproductsCatProps HDS public renaming
    (Id⊞≈Id                  to Id⊞₃≈Id
    ; ⊕Id-Id                 to ⊕₃Id-Id
    ; Id⊕-Id                 to Id⊕₃-Id
    ; ⊕-Id                   to ⊕₃-Id
    ; ⊞-swap²                to ⊞₃-swap²
    ; ⊞-swap-Iso             to ⊞₃-swap-Iso
    ; ⊞-join                 to ⊞₃-join
    ; ⊞-join-assoc           to ⊞₃-join-assoc
    ; ⊞-swap-⨾-join          to ⊞₃-swap-⨾-join
    ; ι-isMono               to ι₃-isMono
    ; κ-isMono               to κ₃-isMono
    ; ⊞-assoc⨾⊞-assocL       to ⊞₃-assoc⨾⊞-assocL
    ; ⊞-assocL⨾⊞-assoc       to ⊞₃-assocL⨾⊞-assoc
    ; ⊞-assoc-Iso            to ⊞₃-assoc-Iso
    ; ⊞-assocL-Iso           to ⊞₃-assocL-Iso
    ; ⊞-assoc-pentagon       to ⊞₃-assoc-pentagon
    ; ⊞-assocL-pentagon      to ⊞₃-assocL-pentagon
    ; ⊞-swap-monoidal        to ⊞₃-swap-monoidal
    ; ⊞-swap-monoidal˘       to ⊞₃-swap-monoidal˘
    ; ⊕-⨾-I⚠I                to ⊕₃-⨾-I⚠I
    ; I⚠I-assoc˘             to I⚠₃I-assoc˘
    ; I⚠I-monoidal˘          to I⚠₃I-monoidal˘
    ; ⊞-join-monoidal        to ⊞₃-join-monoidal
    )
```

4.11.3 Finite Coproducts

If a CompOp has binary coproducts (HasCoproducts) and zero-ary coproducts (HasInitialObject), then it has all finite coproducts.

We prove in particular the properties necessary to show that these give rise to a monoidal category.

```
module HasFiniteCoproducts (hasCoproducts : HasCoproducts) (hasInit : HasInitialObject) where
  open HasCoproducts hasCoproducts
  open HasInitialObject hasInit

  ⊞-leftUnit : {A : Obj} → Mor (⓵ ⊞ A) A
  ⊞-leftUnit {A} = ⓘ ⚠ Id

  ⊞-leftUnit⁻¹ : {A : Obj} → Mor A (⓵ ⊞ A)
  ⊞-leftUnit⁻¹ {A} = κ

  ⊞-rightUnit : {A : Obj} → Mor (A ⊞ ⓵) A
  ⊞-rightUnit {A} = Id ⚠ ⓘ

  ⊞-rightUnit⁻¹ : {A : Obj} → Mor A (A ⊞ ⓵)
  ⊞-rightUnit⁻¹ {A} = ι


  ⊞-leftUnit-naturality : {A B : Obj} {F : Mor A B} → (Id {⓵} ⊕ F) ⨾ ⊞-leftUnit ≈ ⊞-leftUnit ⨾ F
  ⊞-leftUnit-naturality {A} {B} {F} = ≈-begin
        (Id ⊕ F) ⨾ (ⓘ ⚠ Id)
      ≈⟨ ⊕-⨾-⚠ ⟨≈≈⟩ ⚠-cong ≈ⓘ rightId ⟩
        ⓘ ⚠ F
      ≈˘⟨ ⚠-⨾ ⟨≈≈⟩ ⚠-cong ≈ⓘ leftId ⟩
```

```
          (ⓘ ⨹ Id) ⨾ F
        □

⊞-rightUnit-naturality : {A B : Obj} {F : Mor A B} → (F ⊕ Id {⦶}) ⨾ ⊞-rightUnit ≈ ⊞-rightUnit ⨾ F
⊞-rightUnit-naturality {A} {B} {F} = ≈-begin
          (F ⊕ Id) ⨾ (Id ⨹ ⓘ)
        ≈⟨ ⊕-⨾-⨹ ⟨≈≈⟩ ⨹-cong rightId ≈ⓘ ⟩
          F ⨹ ⓘ
        ≈˘⟨ ⨹-⨾ ⟨≈≈⟩ ⨹-cong leftId ≈ⓘ ⟩
          (Id ⨹ ⓘ) ⨾ F
        □

⊞-leftUnit⁻¹-naturality : {A B : Obj} {F : Mor A B} → ⊞-leftUnit⁻¹ ⨾ (Id {⦶} ⊕ F) ≈ F ⨾ ⊞-leftUnit⁻¹
⊞-leftUnit⁻¹-naturality {A} {B} {F} = κ⨾⊕

⊞-rightUnit⁻¹-naturality : {A B : Obj} {F : Mor A B} → ⊞-rightUnit⁻¹ ⨾ (F ⊕ Id {⦶}) ≈ F ⨾ ⊞-rightUnit⁻¹
⊞-rightUnit⁻¹-naturality {A} {B} {F} = ι⨾⊕

⊞-leftUnit-⦶ : ⊞-leftUnit {⦶} ≈ ⊞-rightUnit {⦶}
⊞-leftUnit-⦶ = ⨹-cong (≈-sym ≈ⓘ) ≈ⓘ

⊞-leftUnit⁻¹-⦶ : ⊞-leftUnit⁻¹ {⦶} ≈ ⊞-rightUnit⁻¹ {⦶}
⊞-leftUnit⁻¹-⦶ = ⦶≈

⊞-rightUnit-⦶ : ⊞-rightUnit {⦶} ≈ ⊞-leftUnit {⦶}
⊞-rightUnit-⦶ = ≈-sym ⊞-leftUnit-⦶

⊞-leftUnit-leftUnit⁻¹ : {A : Obj} → ⊞-leftUnit ⨾ ⊞-leftUnit⁻¹ ≈ Id {⦶ ⊞ A}
⊞-leftUnit-leftUnit⁻¹ = ≈-begin
            (ⓘ ⨹ Id) ⨾ κ
        ≈⟨ ⨹-⨾ ⟨≈≈⟩ ⨹-cong ⦶≈ leftId ⟩
            ι ⨹ κ
        ≈⟨ Id⊞≈Id ⟩
            Id
        □

⊞-leftUnit⁻¹-leftUnit : {A : Obj} → ⊞-leftUnit⁻¹ ⨾ ⊞-leftUnit ≈ Id {A}
⊞-leftUnit⁻¹-leftUnit = κ⨾⨹

⊞-rightUnit-rightUnit⁻¹ : {A : Obj} → ⊞-rightUnit ⨾ ⊞-rightUnit⁻¹ ≈ Id {A ⊞ ⦶}
⊞-rightUnit-rightUnit⁻¹ = ≈-begin
            (Id ⨹ ⓘ) ⨾ ι
        ≈⟨ ⨹-⨾ ⟨≈≈⟩ ⨹-cong leftId ⦶≈ ⟩
            ι ⨹ κ
        ≈⟨ Id⊞≈Id ⟩
            Id
        □

⊞-rightUnit⁻¹-rightUnit : {A : Obj} → ⊞-rightUnit⁻¹ ⨾ ⊞-rightUnit ≈ Id {A}
⊞-rightUnit⁻¹-rightUnit = ι⨾⨹

⊞-leftUnit-Iso : {A : Obj} → Iso (⦶ ⊞ A) A
⊞-leftUnit-Iso = record
  {isoMor = ⊞-leftUnit
  ; isIso = record   {_⁻¹ = ⊞-leftUnit⁻¹
                     ; rightInverse = ⊞-leftUnit-leftUnit⁻¹
                     ; leftInverse = ⊞-leftUnit⁻¹-leftUnit
                     }
  }

⊞-leftUnit⁻¹-Iso : {A : Obj} → Iso A (⦶ ⊞ A)
⊞-leftUnit⁻¹-Iso = invIso ⊞-leftUnit-Iso

⊞-rightUnit-Iso : {A : Obj} → Iso (A ⊞ ⦶) A
⊞-rightUnit-Iso = record
  {isoMor = ⊞-rightUnit
  ; isIso = record   {_⁻¹ = ⊞-rightUnit⁻¹
```

```
                    ; rightInverse = ⊞-rightUnit-rightUnit⁻¹
                    ; leftInverse = ⊞-rightUnit⁻¹-rightUnit
                    }
  }
⊞-rightUnit⁻¹-Iso : {A : Obj} → Iso A (A ⊞ 𝕀)
⊞-rightUnit⁻¹-Iso = invIso ⊞-rightUnit-Iso

ⓘ≈LU⁻¹⨾ⓘ⊕ⓘ : {A B : Obj} → ⓘ {A ⊞ B} ≈ ⊞-leftUnit⁻¹ {𝕀} ⨾ (ⓘ {A} ⊕ ⓘ {B})
ⓘ≈LU⁻¹⨾ⓘ⊕ⓘ = 𝕀≈

𝕀◬𝕀≈LU : {F G : Mor 𝕀 𝕀} → F ◬ G ≈ ⊞-leftUnit {𝕀}
𝕀◬𝕀≈LU {F} {G} = ≈-begin
        F ◬ G
      ≈⟨ ◬-cong ≈ⓘ 𝕀≈ ⟩
        ⓘ ◬ Id
      ≈⟨ ≈-refl ⟩
        ⊞-leftUnit
      □

Id⊕ⓘ-⨾-I◬I : {A : Obj} → (Id {A} ⊕ ⓘ {A}) ⨾ (Id ◬ Id) ≈ ⊞-rightUnit
Id⊕ⓘ-⨾-I◬I = ≈-begin
                    (Id ⊕ ⓘ) ⨾ (Id ◬ Id)
                  ≈⟨ ⊕-⨾-I◬I ⟩
                    Id ◬ ⓘ
                  ≈⟨ ≈-refl ⟩
                    ⊞-rightUnit
                  □

⊞-triangle   : {A B : Obj}
             → ⊞-assoc {A} {𝕀} {B} ⨾ (Id {A} ⊕ ⊞-leftUnit {B}) ≈ ⊞-rightUnit {A} ⊕ Id {B}
⊞-triangle {A} {B} = ≈-begin
        ⊞-assoc ⨾ (Id ⊕ (ⓘ ◬ Id))
      ≈⟨ ≈-refl ⟩
        ⊞-assoc ⨾ (Id ⨾ ι ◬ (ⓘ ◬ Id) ⨾ κ)
      ≈⟨ ⨾-cong₂ (◬-cong₂ ◬-⨾) ⟩
        ⊞-assoc ⨾ (Id ⨾ ι ◬ (ⓘ ⨾ κ ◬ Id ⨾ κ))
      ≈⟨ ⊞-assoc-◬◬ ⟩
        (Id ⨾ ι ◬ ⓘ ⨾ κ) ◬ Id ⨾ κ
      ≈⟨ ◬-cong (◬-cong₂ 𝕀≈) leftId ⟩
        (Id ⨾ ι ◬ ⓘ ⨾ ι) ◬ κ
      ≈˘⟨ ◬-cong ◬-⨾ leftId ⟩
        (Id ◬ ⓘ) ⊕ Id
      □

⊞-triangle⁻¹ :      {A B : Obj}
               → (Id {A} ⊕ ⊞-leftUnit⁻¹ {B}) ⨾ ⊞-assocL {A} {𝕀} {B} ≈ ⊞-rightUnit⁻¹ ⊕ Id {B}
⊞-triangle⁻¹ {A} {B} = ≈-begin
          (Id {A} ⊕     ⊞-leftUnit⁻¹ {B}) ⨾ ⊞-assocL {A} {𝕀} {B}
        ≈⟨ ⊕-⨾-◬ ⟩
            Id ⨾ (ι ⨾ ι) ◬ κ ⨾ (κ ⨾ ι ◬ κ)
        ≈⟨ ◬-cong leftId κ⨾◬ ⟩
            ι ⨾ ι ◬ κ
        ≈˘⟨ ◬-cong₂ leftId ⟩
          ι ⨾ ι ◬ Id ⨾ κ
        ≈⟨ ≈-refl ⟩
          ι ⊕ Id
        □

⊞-swapInit²≈Id : ⊞-swap {𝕀} {𝕀} ≈ Id {𝕀 ⊞ 𝕀}
⊞-swapInit²≈Id = ≈-begin
```

```
                          κ ⨹ ι
                        ≈⟨ ⨹-cong ⦶≈ ⦶≈ ⟩
                          ι ⨹ κ
                        ≈⟨ Id⊞≈Id ⟩
                          Id
                        □

  ⊞-swap-leftUnit : {A : Obj} → ⊞-swap {A} {⦶} ⨾ ⊞-leftUnit ≈ ⊞-rightUnit
  ⊞-swap-leftUnit = ⊞-swap-⨾-⨹

  ⊞-swap-leftUnit⁻¹ : {A : Obj} → ⊞-leftUnit⁻¹ ⨾ ⊞-swap {⦶} {A} ≈ ⊞-rightUnit⁻¹
  ⊞-swap-leftUnit⁻¹ = κ⨾⨹


  Id⊕κ-⨾-⊞-assocL : {A B C : Obj} → (Id ⊕ κ) ⨾ ⊞-assocL {A} {B} {C} ≈ ι ⊕ Id
  Id⊕κ-⨾-⊞-assocL {A} {B} {C} = ≈-begin
          (Id ⊕ κ) ⨾ ⊞-assocL {A} {B} {C}
      ≈⟨ ⊕-⨾-⨹ ⟨≈≈⟩ ⨹-cong leftId κ⨾⊕Id ⟩
          ι ⨾ ι ⨹ κ
      ≈˘⟨ ⨹-cong₂ leftId ⟩
          ι ⨾ ι ⨹ Id ⨾ κ
      ≈⟨ ≈-refl ⟩
          ι ⊕ Id
      □


⨾≈Id-⊕-⨾≈Id : {A B C D : Obj}
  {f : Mor A B} {g : Mor C D}
  {f⁻¹ : Mor B A} {g⁻¹ : Mor D C}
  → (f ⨾ f⁻¹ ≈ Id) → (g ⨾ g⁻¹ ≈ Id)
  → (f ⊕ g) ⨾ (f⁻¹ ⊕ g⁻¹) ≈ Id
⨾≈Id-⊕-⨾≈Id {f = f} {g} {f⁻¹} {g⁻¹} ff⁻¹≈Id gg⁻¹≈Id =
  ≈-begin
    (f ⊕ g) ⨾ (f⁻¹ ⊕ g⁻¹)
  ≈˘⟨ ⨾-⊕-⨾ ⟩
    (f ⨾ f⁻¹) ⊕ (g ⨾ g⁻¹)
  ≈⟨ ⊕-cong ff⁻¹≈Id gg⁻¹≈Id ⟩
    Id ⊕ Id
  ≈⟨ ⊕-Id ⟩
    Id
  □
```

4.11.4 Pushouts

A span $A \xleftarrow{\mathsf{Id}} A \xrightarrow{\mathsf{F}} B$ containing an identity morphism has the cospan $A \xrightarrow{\mathsf{F}} B \xleftarrow{\mathsf{Id}} A$ as pushout:

```
idPushout : {A B : Obj} → (F : Mor A B) → Pushout Id F
idPushout F = record
  {left  = F
  ; right = Id
  ; prf   = record
    {commutes = leftId ⟨≈≈˘⟩ rightId
    ; universal   = λ {Z} {R′} {S′} Id⨾R′≈F⨾S′ → record
      {univMor = S′
      ; univMor-factors-left   = Id⨾R′≈F⨾S′ ⟨≈˘≈⟩ leftId
      ; univMor-factors-right = leftId
      ; univMor-unique = λ {V} F⨾V≈R′ Id⨾V≈S′ → leftId ⟨≈˘≈⟩ Id⨾V≈S′
      }
    }
  }
```

More generally every pushout for a span $\mathsf{A} \xleftarrow{\mathsf{Id}} \mathsf{A} \xrightarrow{\mathsf{F}} \mathsf{B}$ contains an iso as “right” morphism:

```
id₁Pushout-Iso : {A B C : Obj} {G : Mor A B} {R : Mor A C} {S : Mor B C}
  → IsPushout Id G R S → IsIso S
id₁Pushout-Iso {A} {B} {C} {G} {R} {S} isPO = record
  {_⁻¹ = U.univMor
  ; rightInverse = U.univMor-factors-right
  ; leftInverse = CoCone2Univ.univMor-unique′
    (universal {C} {R} {S} commutes)
    {U.univMor ⨾ S}
    {Id}
    (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ U.univMor-factors-left ⟨≈≈⟩ (commutes ⟨≈˘≈⟩ leftId))
    (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ U.univMor-factors-right ⟨≈≈⟩ leftId)
    rightId
    rightId
  }
  where
    open IsPushout isPO
    module U = CoCone2Univ (universal {B} {G} {Id} (leftId ⟨≈≈˘⟩ rightId))


id₂Pushout-Iso : {A B C : Obj} {F : Mor A B} {R : Mor B C} {S : Mor A C}
  → IsPushout F Id R S → IsIso R
id₂Pushout-Iso {A} {B} {C} {F} {R} {S} isPO = record
  {_⁻¹ = U.univMor
  ; rightInverse = U.univMor-factors-left
  ; leftInverse = CoCone2Univ.univMor-unique′
    (universal {C} {R} {S} commutes)
    {U.univMor ⨾ R}
    {Id}
    (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ U.univMor-factors-left ⟨≈≈⟩ leftId)
    (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ U.univMor-factors-right ⟨≈≈⟩ commutes ⟨≈≈⟩ leftId)
    rightId
    rightId
  }
  where
    open IsPushout isPO
    module U = CoCone2Univ (universal {B} {Id} {F} (rightId ⟨≈≈˘⟩ leftId))
```

Yet more generally, for a span $\mathsf{A} \xleftarrow{\mathsf{F}} \mathsf{A} \xrightarrow{\mathsf{G}} \mathsf{B}$ where F is iso, every pushout contains an iso as “right” morphism:

```
iso₁Pushout-Iso : {A B C D : Obj} {F : Mor A B} {G : Mor A C} {R : Mor B D} {S : Mor C D}
                  → IsPushout F G R S → IsIso F → IsIso S
iso₁Pushout-Iso {A} {B} {C} {D} {F} {G} {R} {S} isPO F-isIso = record
  {_⁻¹ = U.univMor
  ; rightInverse = U.univMor-factors-right
  ; leftInverse  = CoCone2Univ.univMor-unique′
    (universal {D} {R} {S} commutes)
    {U.univMor ⨾ S}
    {Id}
    (≈-begin
          R ⨾ U.univMor ⨾ S
        ≈⟨ ⨾-cong₁&₂₁ U.univMor-factors-left ⟩
          F._⁻¹ ⨾ G ⨾ S
        ≈˘⟨ ⨾-cong₂ commutes ⟩
          F._⁻¹ ⨾ F ⨾ R
        ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ F.leftInverse ⟨≈≈⟩ leftId ⟩
          R
          □)
```

```
      (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ U.univMor-factors-right ⟨≈≈⟩ leftId)
      rightId
      rightId
  }
  where
    open IsPushout isPO
    module F = IsIso F-isIso
    module U = CoCone2Univ (universal {C} {F._⁻¹ ⨾ G} {Id}
                   (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ F.rightInverse ⟨≈≈⟩ leftId ⟨≈≈˘⟩ rightId))
```

Any object that is isomorphic to a pushout object gives rise to a pushout:

```
IsPushout-Iso-extend : {A B C D D′ : Obj} {F : Mor A B} {G : Mor A C} {R : Mor B D} {S : Mor C D}
                     → IsPushout F G R S → (J : Iso D D′)
                     → IsPushout F G (R ⨾ isoMor J) (S ⨾ isoMor J)
IsPushout-Iso-extend {A} {B} {C} {D} {D'} {F} {G} {R} {S} isPO J = record
  {commutes = ⨾-cong₁&₂₁ commutes
  ; universal = λ {Z} {R′} {S′} F⨾R′≈G⨾S′ → let
    open CoCone2Univ (universal {Z} {R′} {S′} F⨾R′≈G⨾S′)
    in record
      {univMor = J ⁻¹ ⨾ univMor
      ; univMor-factors-left = ≈-begin
          (R ⨾ isoMor J) ⨾ J ⁻¹ ⨾ univMor
        ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (rightInverse J) ⟨≈≈⟩ leftId) ⟩
          R ⨾ univMor
        ≈⟨ univMor-factors-left ⟩
          R′
        □
      ; univMor-factors-right = ⨾-assoc ⟨≈≈⟩
          ⨾-cong₂ (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (rightInverse J) ⟨≈≈⟩ leftId) ⟨≈≈⟩
          univMor-factors-right
      ; univMor-unique = λ {V} R⨾J⨾V≈R′ S⨾J⨾V≈S′ → ≈-begin
          V
        ≈˘⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (leftInverse J) ⟨≈≈⟩ leftId ⟩
          J ⁻¹ ⨾ isoMor J ⨾ V
        ≈⟨ ⨾-cong₂ (univMor-unique (⨾-assocL ⟨≈≈⟩ R⨾J⨾V≈R′) (⨾-assocL ⟨≈≈⟩ S⨾J⨾V≈S′)) ⟩
          J ⁻¹ ⨾ univMor
        □
      }
  }
  where
    open IsPushout isPO using (commutes; universal)
```

If this is to be used for proving the pushout property for existing morphisms, a different shape will be more useful:

```
IsPushout-⨾-Iso : {A B C D D′ : Obj} {F : Mor A B} {G : Mor A C}
                 {R : Mor B D} {S : Mor C D}
                 {R′ : Mor B D′} {S′ : Mor C D′}
  → IsPushout F G R S → (J : Iso D D′)
  → R′ ≈ R ⨾ isoMor J → S′ ≈ S ⨾ isoMor J
  → IsPushout F G R′ S′
IsPushout-⨾-Iso {A} {B} {C} {D} {D'} {F} {G} {R} {S} {R′} {S′} isPO J R′≈R⨾J S′≈S⨾J = record
  {commutes = ⨾-cong₂ R′≈R⨾J ⟨≈≈⟩ ⨾-cong₁&₂₁ commutes ⟨≈≈˘⟩ ⨾-cong₂ S′≈S⨾J
  ; universal = λ {Z} {R″} {S″} F⨾R″≈G⨾S″ → let
    open CoCone2Univ (universal {Z} {R″} {S″} F⨾R″≈G⨾S″)
    in record
      {univMor = J ⁻¹ ⨾ univMor
      ; univMor-factors-left = ≈-begin
          R′ ⨾ J ⁻¹ ⨾ univMor
```

```
      ≈⟨ ⨾-cong₁ R′≈R⨾J ⟨≈≈⟩ ⨾-assoc ⟩
        R ⨾ isoMor J ⨾ J ⁻¹ ⨾ univMor
      ≈⟨ ⨾-cong₂ (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (rightInverse J) ⟨≈≈⟩ leftId) ⟩
        R ⨾ univMor
      ≈⟨ univMor-factors-left ⟩
        R″
      □
    ; univMor-factors-right = ⨾-cong₁ S′≈S⨾J ⟨≈≈⟩ ⨾-assoc ⟨≈≈⟩
        ⨾-cong₂ (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (rightInverse J) ⟨≈≈⟩ leftId) ⟨≈≈⟩
        univMor-factors-right
    ; univMor-unique = λ {V} R′⨾V≈R″ S′⨾V≈S″ → ≈-begin
        V
      ≈˘⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (leftInverse J) ⟨≈≈⟩ leftId ⟩
        J ⁻¹ ⨾ isoMor J ⨾ V
      ≈⟨ ⨾-cong₂ (univMor-unique {isoMor J ⨾ V}
        (⨾-assocL ⟨≈≈⟩ (⨾-cong₁ R′≈R⨾J ⟨≈˘≈⟩ R′⨾V≈R″))
        (⨾-assocL ⟨≈≈⟩ (⨾-cong₁ S′≈S⨾J ⟨≈˘≈⟩ S′⨾V≈S″))) ⟩
        J ⁻¹ ⨾ univMor
      □
    }
  }
  where
    open IsPushout isPO using (commutes; universal)
```

4.11.5 Collecting Re-Export

```
module CatFinColimits {i j k : Level} {Obj : Set i} (C : Category j k Obj) where
  open Category C using (semigroupoid)
  open Categoric.Semigroupoid.FinColimits semigroupoid public
    hiding (module HasCoproducts; module HasCoproducts₁
          ; module HasCoproducts₂; module HasCoproducts₃)
  open CatFinColimits₀ C public
```

4.12 Categoric.Category.FinLimits

As in Categoric.FinLimits (Sect. 4.8), we dualise what we have for colimits to obtain our material for limits.

Here we have the additional problem that isomorphisms are of different type than isomorphisms in the opposite category (see Categoric.IdOp (Sect. 3.13)), so in order to avoid limits as being perceived to be "treated unfairly" in comparison with colimits, we need to wrap all functions involving isomorphisms with the appropriate conversions.

As usual, we avoid "**using**" so that overlooked renamings are more likely trigger an error earlier. To record that we didn't forget about the material concerning strict initial objects, we include renamings for all of it, although these will almost certainly never be used — for that reason we did not bother to add isomorphism wrappers for these functions.

We do hide most of the material concerning strict initial objects, at least as long as we are not aware of any actual uses for strict terminal objects.

```
module CatFinLimits₀ {i j k : Level} {Obj : Set i} (C : Category j k Obj) where
  open Category C
  private opC = oppositeCategory C
  private module FinLimits = Categoric.Semigroupoid.FinLimits semigroupoid
  open FinLimits hiding (module HasProducts)
  private module opFinColimits = CatFinColimits₀ opC
```

```
open opFinColimits public hiding
      (module HasCoproducts
      ; module HasCoproducts₁
      ; module HasCoproducts₂
      ; module HasCoproducts₃
      ; module HasFiniteCoproducts
      ; module StrictInitial
      ; module StrictInitial′
      ; HasStrictInitialObject
      ; module HasStrictInitial
      ; module HasStrictInitial₁
      ; module HasStrictInitial₂
      ; IsInitial-Iso
      ; IsInitial-⨾-Iso
      ; IsCoproduct-Iso
      ; IsCoproduct-⨾-Iso
      ; id₁Pushout-Iso
      ; id₂Pushout-Iso
      ; iso₁Pushout-Iso
      ; IsPushout-Iso-extend
      ; IsPushout-⨾-Iso
      )
  renaming
      (IsStrictInitial                    to IsStrictTerminal
      ; strictInitialSG-IsInitial         to strictTerminalSG-IsTerminal
      ; strictInitialSG-IsStrictInitial   to strictTerminalSG-IsStrictTerminal
      ; strictInitial-IsStrictInitialSG   to strictTerminal-IsStrictTerminalSG
      ; idPushout                         to idPullback
      )

IsTerminal-Iso  : {T₁ : Obj} → IsTerminal T₁
                → {T₂ : Obj} → IsTerminal T₂
                → Iso T₁ T₂
IsTerminal-Iso T₁-isT T₂-isT = unopIso⁻¹ (opFinColimits.IsInitial-Iso T₁-isT T₂-isT)
IsTerminal-⨾-Iso : {T₁ : Obj} → IsTerminal T₁
                 → {T₂ : Obj} → Iso T₁ T₂
                 → IsTerminal T₂
IsTerminal-⨾-Iso T₁-isT J =  opFinColimits.IsInitial-⨾-Iso T₁-isT (opIso⁻¹ J)


IsProduct-Iso : {A B : Obj}
              → {P₁ : Obj} {π₁ : Mor P₁ A} {ρ₁ : Mor P₁ B} → IsProduct π₁ ρ₁
              → {P₂ : Obj} {π₂ : Mor P₂ A} {ρ₂ : Mor P₂ B} → IsProduct π₂ ρ₂
              → Σ Φ : Iso P₂ P₁ • ((isoMor Φ ⨾ π₁ ≈ π₂ × isoMor Φ ⨾ ρ₁ ≈ ρ₂)
                                  × (Φ ⁻¹ ⨾ π₂ ≈ π₁ × Φ ⁻¹ ⨾ ρ₂ ≈ ρ₁))
IsProduct-Iso isProd₁ isProd₂ with opFinColimits.IsCoproduct-Iso isProd₁ isProd₂
... | Φ, prf = unopIso Φ, prf
IsProduct-⨾-Iso :     {A B : Obj}
                → {P₁ : Obj} {π₁ : Mor P₁ A} {ρ₁ : Mor P₁ B} → IsProduct π₁ ρ₁
                → {P₂ : Obj} → (Φ : Iso P₂ P₁)
                → IsProduct (isoMor Φ ⨾ π₁) (isoMor Φ ⨾ ρ₁)
IsProduct-⨾-Iso isProd₁ Φ = opFinColimits.IsCoproduct-⨾-Iso isProd₁ (opIso Φ)


id₁Pullback-Iso : {A B C : Obj} {G : Mor B A} {R : Mor C A} {S : Mor C B}
                → IsPullback Id G R S → IsIso S
id₁Pullback-Iso isPB = unopIsIso (opFinColimits.id₁Pushout-Iso isPB)
id₂Pullback-Iso :     {A B C : Obj} {F : Mor B A} {R : Mor C B} {S : Mor C A}
```

```
                   → IsPullback F Id R S → IsIso R
id₂Pullback-Iso isPB = unopIsIso (opFinColimits.id₂Pushout-Iso isPB)
iso₁Pullback-Iso : {A B C D : Obj} {F : Mor B A} {G : Mor C A} {R : Mor D B} {S : Mor D C}
                   → IsPullback F G R S → IsIso F → IsIso S
iso₁Pullback-Iso isPB F-isIso = unopIsIso (opFinColimits.iso₁Pushout-Iso isPB (opIsIso F-isIso))
IsPullback-Iso-extend : {A B C D D′ : Obj} {F : Mor B A} {G : Mor C A} {R : Mor D B} {S : Mor D C}
                        → IsPullback F G R S → (J : Iso D′ D)
                        → IsPullback F G (isoMor J ⨾ R) (isoMor J ⨾ S)
IsPullback-Iso-extend isPB J = opFinColimits.IsPushout-Iso-extend isPB (opIso J)
Iso-⨾-IsPullback : {A B C D D′ : Obj} {F : Mor B A} {G : Mor C A}
                   {R : Mor D B} {S : Mor D C}
                   {R′ : Mor D′ B} {S′ : Mor D′ C}
                 → IsPullback F G R S → (J : Iso D′ D)
                 → R′ ≈ isoMor J ⨾ R → S′ ≈ isoMor J ⨾ S
                 → IsPullback F G R′ S′
Iso-⨾-IsPullback isPB J = opFinColimits.IsPushout-⨾-Iso isPB (opIso J)

module HasProducts (HDS : HasProducts) where
  open FinLimits.HasProducts HDS public
  open CatFinColimits.HasCoproducts.HasCoproductsCatProps opC opHasCoproducts public renaming
    (Id⊞≈Id              to Id⊠≈Id
    ; ⊕Id-Id             to ⊗Id-Id
    ; Id⊕-Id             to Id⊗-Id
    ; ⊕-Id               to ⊗-Id
    ; ⊞-swap²            to ⊠-swap²
    ; ⊞-join             to ⊠-dup
    ; ⊞-join-assoc       to ⊠-dup-assoc
    ; ⊞-swap-⨾-join      to ⊠-dup-⨾-swap
    ; ι-isMono           to π-isEpi
    ; κ-isMono           to ρ-isEpi
    ; ⊞-assoc⨾⊞-assocL   to ⊠-assoc⨾⊠-assocL
    ; ⊞-assocL⨾⊞-assoc   to ⊠-assocL⨾⊠-assoc
    ; ⊞-assoc-Iso        to ⊠-assocL-Iso
    ; ⊞-assocL-Iso       to ⊠-assoc-Iso
    ; ⊞-assoc-pentagon   to ⊠-assocL-pentagon
    ; ⊞-assocL-pentagon  to ⊠-assoc-pentagon
    ; ⊞-swap-monoidal    to ⊠-swap-monoidal˘
    ; ⊞-swap-monoidal˘   to ⊠-swap-monoidal
    ; ⊕-⨾-I⚠I            to I▽I-⨾-⊗
    ; I⚠I-assoc˘         to I▽I-assoc
    ; I⚠I-monoidal˘      to I▽I-monoidal
    ; ⊞-join-monoidal    to ⊠-dup-monoidal
    )
module HasFiniteProducts (HasPrds : HasProducts) (hasTerm : HasTerminalObject) where
  open FinLimits.HasProducts HasPrds using (opHasCoproducts)
  open CatFinColimits.HasFiniteCoproducts opC (opHasCoproducts) hasTerm public renaming
    (⊞-leftUnit                  to ⊠-leftUnit⁻¹   -- : {A : Obj} → Mor A (Ⓣ ⊠ A)
    ; ⊞-leftUnit⁻¹               to ⊠-leftUnit     -- = ρ : {A : Obj} → Mor (Ⓣ ⊠ A) A
    ; ⊞-leftUnit-naturality      to ⊠-leftUnit⁻¹-naturality
    ; ⊞-leftUnit⁻¹-naturality    to ⊠-leftUnit-naturality   -- ... → (Id ⊠ F) ⨾ ⊞-leftUnit⁻¹ = ⊞-leftUnit⁻¹ ⨾ F
    ; ⊞-leftUnit-leftUnit⁻¹      to ⊠-leftUnit-leftUnit⁻¹
    ; ⊞-leftUnit⁻¹-leftUnit      to ⊠-leftUnit⁻¹-leftUnit
    ; ⊞-rightUnit                to ⊠-rightUnit⁻¹
    ; ⊞-rightUnit⁻¹              to ⊠-rightUnit
    ; ⊞-rightUnit-naturality     to ⊠-rightUnit⁻¹-naturality
    ; ⊞-rightUnit⁻¹-naturality   to ⊠-rightUnit-naturality
    ; ⊞-rightUnit-rightUnit⁻¹    to ⊠-rightUnit-rightUnit⁻¹
```

```
    ; ⊞-rightUnit⁻¹-rightUnit     to ⊠-rightUnit⁻¹-rightUnit
    ; ⓘ≈LU⁻¹⨾ⓘ⊕ⓘ                to ⓣ≈ⓣ⊗ⓣ⨾LU
    ; Ⓘ ⨹ Ⓘ ≈LU to Ⓣ ▽Ⓣ ≈LU⁻¹
    ; Id⊕ⓘ-⨾-I⨹I to I▽I-⨾-Id⊗ⓣ
    ; ⊞-triangle⁻¹  to ⊠-triangle
    ; ⊞-triangle    to ⊠-triangle⁻¹
    ; ⊞-leftUnit⁻¹-Ⓘ             to ⊠-leftUnit-Ⓣ
    ; ⊞-swapInit²≈Id              to ⊠-swapInit²≈Id
    ; ⊞-swap-leftUnit             to ⊠-swap-leftUnit⁻¹
    ; ⊞-swap-leftUnit⁻¹           to ⊠-swap-leftUnit
    ; Id⊕κ-⨾-⊞-assocL             to ⊠-assoc-⨾-Id⊗ρ
    ; ⨾≈Id-⊕-⨾≈Id to                  ⨾≈Id-⊗-⨾≈Id
    )
```

```
module CatFinLimits {i j k : Level} {Obj : Set i} (C : Category j k Obj) where
  open Category C using (semigroupoid)
  open Categoric.Semigroupoid.FinLimits semigroupoid public hiding (module HasProducts)
  open CatFinLimits₀ C public
```

4.13 Categoric.Category.Slice

If $\mathcal{C}$ is a category and X is an object of $\mathcal{C}$, then the *slice category* $\mathcal{C}/\mathsf{X}$ has as objects $\mathcal{C}$-morphisms with target X, and as morphisms from $\mathsf{f_1} : \mathsf{Mor}\ \mathsf{A_1}\ \mathsf{X}$ to $\mathsf{f_2} : \mathsf{Mor}\ \mathsf{A_2}\ \mathsf{X}$ it has $\mathcal{C}$-morphisms $\mathsf{g} : \mathsf{Mor}\ \mathsf{A_1}\ \mathsf{A_2}$ that make the resulting triangle commute, $\mathsf{g} \mathbin{⨾} \mathsf{f_2} \approx \mathsf{f_1}$.

Since our category concept has only propositional equality $\_\!\equiv\!\_$ as equality of objects, note that two different $\mathcal{C}$-morphisms $\mathsf{f_1}\ \mathsf{f_2} : \mathsf{Mor}\ \mathsf{A}\ \mathsf{X}$ give rise to two different objects in the slice category, even if they are *equivalent* in $\mathcal{C}$, that is, $\mathsf{f_1} \approx \mathsf{f_2}$.

```
module _ {ℓc₀ ℓc₁ ℓc₂ : Level} {Obj : Set ℓc₀} (𝒞 : Category ℓc₁ ℓc₂ Obj) (X : Obj) where
  open Category 𝒞
  private
    Obj′ : Set (ℓc₀ ⊍ ℓc₁)
    Obj′ = Σ A : Obj • Mor A X
    Mor′ : Obj′ → Obj′ → Set (ℓc₁ ⊍ ℓc₂)
    Mor′ (A₁ , f₁) (A₂ , f₂) = Σ g : Mor A₁ A₂ • g ⨾ f₂ ≈ f₁
    _≈′_ : {F₁ F₂ : Obj′} → Rel (Mor′ F₁ F₂) ℓc₂
    _≈′_ {A₁ , f₁} {A₂ , f₂} (g , g⨾f₂≈f₁) (h , h⨾f₂≈f₁) = g ≈ h
    Hom′ : Obj′ → Obj′ → Setoid (ℓc₁ ⊍ ℓc₂) ℓc₂
    Hom′ (A₁ , f₁) (A₂ , f₂) = record
      {Carrier = Mor′ (A₁ , f₁) (A₂ , f₂)
      ; _≈_ = _≈′_
      ; isEquivalence = record {refl = ≈-refl; sym = ≈-sym; trans = ≈-trans}
      }
    _⨾′_ : {F₁ F₂ F₃ : Obj′} → Mor′ F₁ F₂ → Mor′ F₂ F₃ → Mor′ F₁ F₃
    _⨾′_ {A₁ , f₁} {A₂ , f₂} {A₃ , f₃} (g , g⨾f₂≈f₁) (h , h⨾f₃≈f₂) = (g ⨾ h) , (≈-begin
        (g ⨾ h) ⨾ f₃
      ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ h⨾f₃≈f₂ ⟩
        g ⨾ f₂
      ≈⟨ g⨾f₂≈f₁ ⟩
        f₁
      □)
  SliceCat : Category (ℓc₁ ⊍ ℓc₂) ℓc₂ Obj′
  SliceCat = record
```

```
      {semigroupoid = record
        {Hom = Hom′
        ; compOp = record {_⨾_ = _⨾′_; ⨾-cong = ⨾-cong; ⨾-assoc = ⨾-assoc}
        }
      ; idOp = record {Id = Id, leftId; leftId = leftId; rightId = rightId}
      }
  SliceObj = Obj′
  SliceMor = Mor′
  SliceHom = Hom′


  open CatFinColimits

  SliceCoproducts : HasCoproducts 𝒞 → HasCoproducts SliceCat
  SliceCoproducts hasCoprod = record
        {_⊞_ = λ {(A₁, f₁) (A₂, f₂) → (A₁ ⊞ A₂), (f₁ ⩓ f₂)}
        ; ι = λ {{A₁, f₁} {A₂, f₂} → ι, ι⨾⩓}
        ; κ = λ {{A₁, f₁} {A₂, f₂} → κ, κ⨾⩓}
        ; isCoproduct = λ {{A₁, f₁} {A₂, f₂} {Z, z} (g₁, g₁⨾z≈f₁) (g₂, g₂⨾z≈f₂) → record
          {univMor = g₁ ⩓ g₂, (⩓-⨾ ⟨≈≈⟩ ⩓-cong g₁⨾z≈f₁ g₂⨾z≈f₂)
          ; univMor-factors-left = ι⨾⩓
          ; univMor-factors-right = κ⨾⩓
          ; univMor-unique = ⩓-unique
          }}
        }
    where
      open HasCoproducts 𝒞 hasCoprod


SliceInitial : HasInitialObject 𝒞 → HasInitialObject SliceCat
SliceInitial hasInit = record
  {⓪ = ⓪, ⓘ
  ; isInitial = (ⓘ, ≈ⓘ), (λ _ → ≈ⓘ)
  }
  where
    open HasInitialObject 𝒞 hasInit
```

Chapter 5

Sort-Indexed Product Semigroupoids and Categories

Given a category and a set of "sorts", we can construct the *sort-indexed product category* having sort-indexed families of objects as objects, and sort-indexed families of morphisms as morphisms. All operations are defined component-wise, which means that most properties will be inherited. This arrangement is used by Kahl (2011a) as an intermediate layer for the study of categories of generalised algebras over some base category.

5.1 Categoric.SortIndexedProduct

Reflection of the semigroupoid sub-identity porperties requires decidable equality on sorts; this is used by the modules starting at Sect. 23.32.

This module only re-exports its imports:

```
open import Categoric.SortIndexedProduct.Semigroupoid          public  -- Sect. 5.3
open import Categoric.SortIndexedProduct.Category              public  -- Sect. 5.4
open import Categoric.SortIndexedProduct.ConvSemigroupoid      public  -- Sect. 5.5
open import Categoric.SortIndexedProduct.ConvCategory          public  -- Sect. 5.6
open import Categoric.SortIndexedProduct.OrderedSemigroupoid   public  -- Sect. 23.1
open import Categoric.SortIndexedProduct.OrderedCategory       public  -- Sect. 23.2
open import Categoric.SortIndexedProduct.MeetOp                public  -- Sect. 23.3
open import Categoric.SortIndexedProduct.LSLSemigroupoid       public  -- Sect. 23.4
open import Categoric.SortIndexedProduct.JoinOp                public  -- Sect. 23.5
open import Categoric.SortIndexedProduct.USLSemigroupoid       public  -- Sect. 23.6
open import Categoric.SortIndexedProduct.USLCategory           public  -- Sect. 23.7
open import Categoric.SortIndexedProduct.LatticeSemigroupoid   public  -- Sect. 23.8
open import Categoric.SortIndexedProduct.HomLatticeDistr       public  -- Sect. 23.9
open import Categoric.SortIndexedProduct.DistrLatSemigroupoid  public  -- Sect. 23.10
open import Categoric.SortIndexedProduct.DomainSemigroupoid    public  -- Sect. 23.11
open import Categoric.SortIndexedProduct.OCD                   public  -- Sect. 23.12
open import Categoric.SortIndexedProduct.OSGC                  public  -- Sect. 23.13
open import Categoric.SortIndexedProduct.OCC                   public  -- Sect. 23.15
open import Categoric.SortIndexedProduct.LeftResOp             public  -- Sect. 23.16
open import Categoric.SortIndexedProduct.RightResOp            public  -- Sect. 23.17
open import Categoric.SortIndexedProduct.SyqOp                 public  -- Sect. 23.18
open import Categoric.SortIndexedProduct.USLSGC                public  -- Sect. 23.19
open import Categoric.SortIndexedProduct.USLCC                 public  -- Sect. 23.20
open import Categoric.SortIndexedProduct.Allegory              public  -- Sect. 23.21
open import Categoric.SortIndexedProduct.Collagory             public  -- Sect. 23.22
open import Categoric.SortIndexedProduct.ZeroMor               public  -- Sect. 23.23
open import Categoric.SortIndexedProduct.DistrAllegory         public  -- Sect. 23.24
open import Categoric.SortIndexedProduct.DivAllegory           public  -- Sect. 23.25
```

```
open import Categoric.SortIndexedProduct.TransClosOp          public   -- Sect. 23.26
open import Categoric.SortIndexedProduct.KleeneSemigroupoid   public   -- Sect. 23.27
open import Categoric.SortIndexedProduct.StarOp               public   -- Sect. 23.29
open import Categoric.SortIndexedProduct.KleeneCategory       public   -- Sect. 23.30
open import Categoric.SortIndexedProduct.KSGC                 public   -- Sect. 23.28
open import Categoric.SortIndexedProduct.KCC                  public   -- Sect. 23.31
open import Categoric.SortIndexedProduct.OSGSubIdReflect      public   -- Sect. 23.32
open import Categoric.SortIndexedProduct.OSGD                 public   -- Sect. 23.33
open import Categoric.SortIndexedProduct.SemiAllegory         public   -- Sect. 23.34
open import Categoric.SortIndexedProduct.SemiCollagory        public   -- Sect. 23.35
open import Categoric.SortIndexedProduct.LeftRestrResOp       public   -- Sect. 23.36
open import Categoric.SortIndexedProduct.RightRestrResOp      public   -- Sect. 23.37
```

5.2 Categoric.SortIndexedProduct.LESGraph

For easy access to basic entities of sort-indexed product (SIP) semigroupoids from within the context of the base semigroupoid, we define them with SIP-prefixed names:

```
module SIPCore (Sort : Set) {i j k : Level} {Obj : Set i}
               (Base : LocalSetoid Obj j k) where
  open LocalEdgeSetoid Base
  infix 4 _≈SIPMor_
  SIPObj : Set i
  SIPObj = Sort → Obj
  SIPMor : SIPObj → SIPObj → Set j
  SIPMor A B = (s : Sort) → Edge (A s) (B s)
  _≈SIPMor_ : {A B : SIPObj} → Rel (SIPMor A B) k
  F ≈SIPMor G = (s : Sort) → F s ≈ G s
  ≈SIPMor-isEquivalence : {A B : SIPObj} → IsEquivalence (_≈SIPMor_ {A} {B})
  ≈SIPMor-isEquivalence = record
      {refl  = λ        s → ≈-refl
      ;sym   = λ eq     s → ≈-sym   (eq s)
      ;trans = λ fg gh s → ≈-trans (fg s) (gh s)
      }
```

The items above constitute the sort-indexed product LES-graph constructed form the Base LES-graph:

```
SIPGraph : LocalSetoid SIPObj j k
SIPGraph A B = record
  {Carrier         = SIPMor A B
  ; _≈_            = _≈SIPMor_
  ; isEquivalence  = ≈SIPMor-isEquivalence
  }
```

5.3 Categoric.SortIndexedProduct.Semigroupoid

For easy access to basic entities of sort-indexed product (SIP) semigroupoids from within the context of the base semigroupoid, we define the following module:

```
module SGSIP (Sort : Set) {i j k : Level} {Obj : Set i}
             (Base : Semigroupoid j k Obj) where
  open Semigroupoid Base
  open SIPCore Sort Hom public
```

```
  infixr 9 _⨾SIP_
  _⨾SIP_ : {A B C : SIPObj} → SIPMor A B → SIPMor B C → SIPMor A C
  F ⨾SIP G = λ s → F s ⨾ G s
  ⨾SIP-cong    : {A B C : SIPObj} {F₁ F₂ : SIPMor A B} {G₁ G₂ : SIPMor B C}
    → F₁ ≈SIPMor F₂ → G₁ ≈SIPMor G₂ → F₁ ⨾SIP G₁ ≈SIPMor F₂ ⨾SIP G₂
  ⨾SIP-cong    = λ eqF eqG s → ⨾-cong (eqF s) (eqG s)
  ⨾SIP-assoc   : {A B C D : SIPObj} {F : SIPMor A B} {G : SIPMor B C} {H : SIPMor C D}
    → (F ⨾SIP G) ⨾SIP H ≈SIPMor F ⨾SIP (G ⨾SIP H)
  ⨾SIP-assoc   = λ s → ⨾-assoc
```

Defining the complete SIP Semigroupoid then just collects these entities together into the corresponding records:

```
SIPSemigroupoid : Semigroupoid {i} j k SIPObj
SIPSemigroupoid = record
  {Hom = SIPGraph
  ;compOp = record
    {_⨾_       = _⨾SIP_
    ;⨾-cong    = ⨾SIP-cong
    ;⨾-assoc   = ⨾SIP-assoc
    }
  }

open import Categoric.Semigroupoid.Span Base
open import Categoric.Semigroupoid.Span SIPSemigroupoid using () renaming
  (module Span to Spanˢ; module Cospan to Cospanˢ
  ; Span to Spanˢ; Cospan to Cospanˢ
  )
open Semigroupoid SIPSemigroupoid using () renaming
  (_⨾_ to compˢ; Hom to Homˢ; Mor to Morˢ; _≈_ to eqˢ)
open import Categoric.Semigroupoid.FinColimits
open import Categoric.Semigroupoid.FinColimits SIPSemigroupoid using () renaming
  (CoCone2Univ to CoCone2Univˢ; Pushout to Pushoutˢ
  ; module CoCone2Univ to CoCone2Univˢ; module Pushout to Pushoutˢ
  )

SIP-isLeftIdentity : {A : SIPObj} {I : SIPMor A A} → ((s : Sort) → isLeftIdentity (I s))
                   → Semigroupoid.isLeftIdentity SIPSemigroupoid I
SIP-isLeftIdentity {A} {I} left {B} {R} s = left s {B s} {R s}
SIP-isRightIdentity : {A : SIPObj} {I : SIPMor A A} → ((s : Sort) → isRightIdentity (I s))
                    → Semigroupoid.isRightIdentity SIPSemigroupoid I
SIP-isRightIdentity {A} {I} right {Z} {S} s = right s {Z s} {S s}
SIP-isIdentity : {A : SIPObj} {I : SIPMor A A} → ((s : Sort) → isIdentity (I s))
               → Semigroupoid.isIdentity SIPSemigroupoid I
SIP-isIdentity {A} {I} I-isId = SIP-isLeftIdentity (proj₁ ∘ I-isId)
                               , SIP-isRightIdentity (proj₂ ∘ I-isId)

unSIPSpan : {A B C : SIPObj} (FG : Spanˢ A B C) (s : Sort) → Span (A s) (B s) (C s)
unSIPSpan FG s = mkSpan (Spanˢ.left FG s) (Spanˢ.right FG s)

unSIPCospan : {A B C : SIPObj} (RS : Cospanˢ A B C) (s : Sort) → Cospan (A s) (B s) (C s)
unSIPCospan RS s = mkCospan (Cospanˢ.left RS s) (Cospanˢ.right RS s)

SIPPushout : HasPushouts Base
           → {A B C : SIPObj} (F : Morˢ A B) (G : Morˢ A C)
           → Σ [PO : Pushoutˢ F G]
```

```
                  let open Pushoutˢ PO using (commutes)
                     renaming (left to R; right to S) in
                  ((s : Sort) → POUniversal Base (F s) (G s) (R s) (S s))
SIPPushout BasePO {A} {B} {C} F G = record
  {obj = D; left = H; right = K
  ; prf = record
    {commutes = λ s → commutes (F s) (G s)
    ; universal = λ {Z} {X} {Y} comm → let
        open CoCone2Univ Base
        un = λ s → universal (F s) (G s) {Z s} {X s} {Y s} (comm s)
      in record
        {univMor                = univMor                ∘ un
        ; univMor-factors-left  = univMor-factors-left  ∘ un
        ; univMor-factors-right = univMor-factors-right ∘ un
        ; univMor-unique        = λ {V} H⨾V≈X K⨾V≈Y s → univMor-unique (un s) {V s} (H⨾V≈X s) (K⨾V≈Y s)
        }
    }
  }, (λ s → universal (F s) (G s))
  where
    module _ {a b c : Obj} (f : Mor a b) (g : Mor a c) where
      open Pushout Base (BasePO f g) public using (obj; left; right; commutes; universal)
    D : SIPObj
    D s = obj (F s) (G s)
    H : SIPMor B D
    H s = left (F s) (G s)
    K : SIPMor C D
    K s = right (F s) (G s)

SIPHasPushouts : HasPushouts Base → HasPushouts SIPSemigroupoid
SIPHasPushouts BasePO F G = proj₁ (SIPPushout BasePO F G)

open SGSIP public

module SIPSG-Setup (Sort : Set) {i j k : Level} {Obj : Set i} (Base : Semigroupoid j k Obj)
  where
  SIPSG = SIPSemigroupoid Sort Base
  module SIPSG = Semigroupoid SIPSG
  open Semigroupoid₁ Base   public
  open Semigroupoid₂ SIPSG public
```

5.4 Categoric.SortIndexedProduct.Category

```
SIPCategory : (Sort : Set) {i j k : Level} {Obj : Set i}
            → Category {i} j k Obj
            → Category {i} j k (Sort → Obj)
SIPCategory Sort Base = let open Category Base in record
  {semigroupoid = SIPSemigroupoid Sort semigroupoid
  ; idOp = record
    {Id      = λ {A} s → Id {A s}
    ; leftId  = λ s → leftId
    ; rightId = λ s → rightId
    }
  }

module SIPCat-Setup (Sort : Set) {i j k : Level} {Obj : Set i} (Base : Category j k Obj)
  where
```

```
  SIPCat = SIPCategory Sort Base
  open Category₁ Base public
  open Category₂ SIPCat public
```

5.5 Categoric.SortIndexedProduct.ConvSemigroupoid

```
SIPConvSemigroupoid : (Sort : Set) {i j k : Level} {Obj : Set i}
                    → ConvSemigroupoid {i} j k Obj
                    → ConvSemigroupoid {i} j k (Sort → Obj)
SIPConvSemigroupoid Sort Base = let open ConvSemigroupoid Base in record
  {semigroupoid = SIPSemigroupoid Sort semigroupoid
  ; convOp = record
    {_˘          = λ R s → (R s) ˘
    ; ˘-cong      = λ eq s → ˘-cong (eq s)
    ; ˘˘          = λ s → ˘˘
    ; ˘-involution = λ s → ˘-involution
    }
  }
```

5.6 Categoric.SortIndexedProduct.ConvCategory

```
SIPConvCategory : (Sort : Set) {i j k : Level} {Obj : Set i}
                → ConvCategory {i} j k Obj
                → ConvCategory {i} j k (Sort → Obj)
SIPConvCategory Sort Base = let open ConvCategory Base in record
  {convSemigroupoid = SIPConvSemigroupoid Sort convSemigroupoid
  ; idOp = Category.idOp (SIPCategory Sort category)
  }
```

Chapter 6

Functors

6.1 Categoric.SGFunctor.Setup

```
module SGFunctorSetup {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (Src : Semigroupoid j₁ k₁ Obj₁)
                      {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (Trg : Semigroupoid j₂ k₂ Obj₂)
  where
  open Semigroupoid₁ Src public
  open Semigroupoid₂ Trg public
```

6.2 Categoric.SGFunctor.Core

Inside IsSGFunctor, we turn the Semigroupoid argument names Src and Trg also into module names, to be able to access Semigroupoid material using simple qualified names.

We still open also SGFunctorSetup Src Trg for access to subscripted infix operators and frequently-used items like Mor_1.

```
record IsSGFunctor {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (Src : Semigroupoid j₁ k₁ Obj₁)
                   {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (Trg : Semigroupoid j₂ k₂ Obj₂)
                   (obj : Obj₁ → Obj₂)
                   (mor : {A B : Obj₁} → Semigroupoid.Mor Src A B
                                       → Semigroupoid.Mor Trg (obj A) (obj B))
                   : Set (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂ ⊍ k₁ ⊍ k₂) where
  private
    module Src = Semigroupoid Src
    module Trg = Semigroupoid Trg
  open SGFunctorSetup Src Trg
  field
    mor-cong : {A B : Obj₁} → {f g : Mor₁ A B} → f ≈₁ g → mor f ≈₂ mor g
    mor-⨾ : {A B C : Obj₁} → {f : Mor₁ A B} → {g : Mor₁ B C}
          → mor (f ⨾₁ g) ≈₂ mor f ⨾₂ mor g
```

```
record SGFunctor {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (Src : Semigroupoid j₁ k₁ Obj₁)
                 {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (Trg : Semigroupoid j₂ k₂ Obj₂)
                 : Set (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂ ⊍ k₁ ⊍ k₂) where
  open Semigroupoid using (Mor)
  field
    obj : Obj₁ → Obj₂
    mor : {A B : Obj₁} → Mor Src A B → Mor Trg (obj A) (obj B)
    isSGFunctor : IsSGFunctor Src Trg obj mor
  open IsSGFunctor isSGFunctor public
```

```
retractSGFunctor : {i₁ i₂ j k : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                 → (F : Obj₂ → Obj₁) (C : Semigroupoid j k Obj₁)
                 → SGFunctor (retractSemigroupoid F C) C
retractSGFunctor F C = record
  {obj = F
  ; mor = λ x → x
  ; isSGFunctor = record {mor-cong = λ x → x; mor-⨾ = Semigroupoid.≈-refl C}
  }
```

6.3 Categoric.SGFunctor.Equality

In general, functor equivalence only demands objects to be mapped to isomorphic images, and will be defined using natural transformations.

We will have uses for a stronger functor equality, established by propositional equality of the object images and simple equivalence of morphisms:

```
module _ {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} {Src : Semigroupoid j₁ k₁ Obj₁}
         {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} {Trg : Semigroupoid j₂ k₂ Obj₂} where
  open SGFunctorSetup Src Trg
  open SGFunctor
  private
    module Src = Semigroupoid Src
    module Trg = Semigroupoid Trg


  infix 4 _≡F≈_
  record _≡F≈_ (F G : SGFunctor Src Trg) : Set (i₁ ⊍ j₁ ⊍ i₂ ⊍ k₂) where
    field
      obj≡ : {A : Obj₁} → obj F A ≡ obj G A
      mor≈ : {A B : Obj₁} {f : Mor₁ A B}
           → Trg.≡-substTrg obj≡ (Trg.≡-substSrc obj≡ (mor F f)) ≈₂ mor G f
    mor≈′ : {A B : Obj₁} {f g : Mor₁ A B} → f ≈₁ g
           → Trg.≡-substTrg obj≡ (Trg.≡-substSrc obj≡ (mor F f)) ≈₂ mor G g
    mor≈′ {f = f} {g} f≈g = ≈₂-begin
        Trg.≡-substTrg obj≡ (Trg.≡-substSrc obj≡ (mor F f))
      ≈₂⟨ Trg.≡-substTrg-cong obj≡ (Trg.≡-substSrc-cong obj≡ (mor-cong F f≈g)) ⟩
        Trg.≡-substTrg obj≡ (Trg.≡-substSrc obj≡ (mor F g))
      ≈₂⟨ mor≈ ⟩
        mor G g
      □₂


  ≡F≈-refl : {F : SGFunctor Src Trg} → F ≡F≈ F
  ≡F≈-refl {F} = record
    {obj≡ = λ {A} → ≡-refl
    ; mor≈ = λ {A} {B} {f} → ≈₂-begin
        Trg.≡-substTrg ≡-refl (Trg.≡-substSrc ≡-refl (mor F f))
      ≈₂≡⟨ Trg.≡-substTrg-contract ≡-refl ⟩
        Trg.≡-substSrc ≡-refl (mor F f)
      ≈₂≡⟨ Trg.≡-substSrc-contract ≡-refl ⟩
        mor F f
      □₂
    }


  ≡F≈-sym : {F G : SGFunctor Src Trg} → F ≡F≈ G → G ≡F≈ F
  ≡F≈-sym {F} {G} F≈G = let open _≡F≈_ F≈G in record
```

```
    {obj≡ = λ {A} → ≡-sym obj≡
    ; mor≈ = λ {A} {B} {f} → ≈₂-begin
            Trg.≡-substTrg (≡-sym obj≡) (Trg.≡-substSrc (≡-sym obj≡) (mor G f))
      ≈₂⟨ Trg.≡-substTrg-cong (≡-sym obj≡) (Trg.≡-substSrc-cong (≡-sym obj≡) (≈₂-sym mor≈)) ⟩
            Trg.≡-substTrg (≡-sym obj≡) (Trg.≡-substSrc (≡-sym obj≡)
            (Trg.≡-substTrg obj≡ (Trg.≡-substSrc obj≡ (mor F f))))
      ≈₂≡⟨ ≡-cong (Trg.≡-substTrg (≡-sym obj≡)) (Trg.≡-substSrcTrg (≡-sym obj≡) obj≡) ⟩
            Trg.≡-substTrg (≡-sym obj≡) (Trg.≡-substTrg obj≡ (Trg.≡-substSrc (≡-sym obj≡)
            (Trg.≡-substSrc obj≡ (mor F f))))
      ≈₂≡⟨ Trg.≡-substTrgTrg-contract obj≡ (≡-sym obj≡) ⟩
            Trg.≡-substSrc (≡-sym obj≡) (Trg.≡-substSrc obj≡ (mor F f))
      ≈₂≡⟨ Trg.≡-substSrcSrc-contract obj≡ (≡-sym obj≡) ⟩
            mor F f
      □₂
    }

  ≡F≈-trans : {F G H : SGFunctor Src Trg} → F ≡F≈ G → G ≡F≈ H → F ≡F≈ H
  ≡F≈-trans {F} {G} {H} F≈G G≈H = let open _≡F≈_ in record
    {obj≡ = λ {A} → ≡-trans (obj≡ F≈G) (obj≡ G≈H)
    ; mor≈ = λ {A} {B} {f} → ≈₂-begin
            Trg.≡-substTrg (≡-trans (obj≡ F≈G) (obj≡ G≈H))
              (Trg.≡-substSrc (≡-trans (obj≡ F≈G) (obj≡ G≈H)) (mor F f))
      ≈₂≡˘⟨ ≡-cong (Trg.≡-substTrg (≡-trans (obj≡ F≈G) (obj≡ G≈H)))
         (Trg.≡-substSrcSrc (obj≡ F≈G) (obj≡ G≈H)) ⟩
            Trg.≡-substTrg (≡-trans (obj≡ F≈G) (obj≡ G≈H))
              (Trg.≡-substSrc (obj≡ G≈H) (Trg.≡-substSrc (obj≡ F≈G) (mor F f)))
      ≈₂≡˘⟨ Trg.≡-substTrgTrg (obj≡ F≈G) (obj≡ G≈H) ⟩
            Trg.≡-substTrg (obj≡ G≈H) (Trg.≡-substTrg (obj≡ F≈G)
              (Trg.≡-substSrc (obj≡ G≈H) (Trg.≡-substSrc (obj≡ F≈G) (mor F f))))
      ≈₂≡⟨ ≡-cong (Trg.≡-substTrg (obj≡ G≈H)) (Trg.≡-substTrgSrc (obj≡ G≈H) (obj≡ F≈G)) ⟩
            Trg.≡-substTrg (obj≡ G≈H) (Trg.≡-substSrc (obj≡ G≈H)
              (Trg.≡-substTrg (obj≡ F≈G) (Trg.≡-substSrc (obj≡ F≈G) (mor F f))))
      ≈₂⟨ Trg.≡-substTrg-cong (obj≡ G≈H) (Trg.≡-substSrc-cong (obj≡ G≈H) (mor≈ F≈G)) ⟩
            Trg.≡-substTrg (obj≡ G≈H) (Trg.≡-substSrc (obj≡ G≈H) (mor G f))
      ≈₂⟨ mor≈ G≈H ⟩
            mor H f
      □₂
    }

  ≡F≈-isEquivalence : IsEquivalence _≡F≈_
  ≡F≈-isEquivalence = record {refl = ≡F≈-refl; sym = ≡F≈-sym; trans = ≡F≈-trans}

≡F≈-Setoid : {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (Src : Semigroupoid j₁ k₁ Obj₁)
           → {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (Trg : Semigroupoid j₂ k₂ Obj₂)
           → Setoid (i₁ ⊍ j₁ ⊍ k₁ ⊍ i₂ ⊍ j₂ ⊍ k₂) (i₁ ⊍ j₁ ⊍ i₂ ⊍ k₂)
≡F≈-Setoid Src Trg = record
  {Carrier = SGFunctor Src Trg
  ; _≈_ = _≡F≈_
  ; isEquivalence = ≡F≈-isEquivalence
  }
```

6.4 Categoric.SGFunctor.Composition

```
module _ {i j k : Level} {Obj : Set i} (C : Semigroupoid j k Obj) where
  open Semigroupoid C
```

```
  Identity : SGFunctor C C
  Identity = record
    {obj = λ x → x
    ; mor = λ x → x
    ; isSGFunctor = record {mor-cong = λ x → x; mor-⨾ = ≈-refl}
    }
  Identity-isFaithful : SGF₀.IsFaithful Identity
  Identity-isFaithful {A} {B} = idF, (λ _ → ≈-refl)
  Identity-isFull : SGF₀.IsFull Identity
  Identity-isFull {A} {B} = idF, (λ _ → ≈-refl)
  Identity-isFullAndFaithful : SGF₀.IsFullAndFaithful Identity
  Identity-isFullAndFaithful {A} {B} = idF, record {left-inverse-of = λ _ → ≈-refl
                                                   ; right-inverse-of = λ _ → ≈-refl}
  Identity-preservesIdentity : SGF₀.PreservesIdentity Identity
  Identity-preservesIdentity isId = isId


infixr 9 _⨾⨾_
_⨾⨾_ : {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} {C₁ : Semigroupoid j₁ k₁ Obj₁}
       {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} {C₂ : Semigroupoid j₂ k₂ Obj₂}
       {i₃ j₃ k₃ : Level} {Obj₃ : Set i₃} {C₃ : Semigroupoid j₃ k₃ Obj₃}
       (F : SGFunctor C₁ C₂) → (G : SGFunctor C₂ C₃) → SGFunctor C₁ C₃
_⨾⨾_ {C₃ = C₃} F G = record
  {obj = λ x → SGFunctor.obj G (SGFunctor.obj F x)
  ; mor = λ x → SGFunctor.mor G (SGFunctor.mor F x)
  ; isSGFunctor = record
    {mor-cong = λ x → SGFunctor.mor-cong G (SGFunctor.mor-cong F x)
    ; mor-⨾ = Semigroupoid.≈-trans C₃ (SGFunctor.mor-cong G (SGFunctor.mor-⨾ F)) (SGFunctor.mor-⨾ G)
    }
  }
```

For construction of the inverse (up to _≡F≈_) of a full-and-faithful functor and for showing the left-inverse property we only require a left-inverse of the object mapping obj:

```
module _ {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} {C₁ : Semigroupoid j₁ k₁ Obj₁}
         {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} {C₂ : Semigroupoid j₂ k₂ Obj₂}
         (F : SGFunctor C₁ C₂)
         (isFF : SGF₀.IsFullAndFaithful F)
         (obj⁻¹ : Obj₂ → Obj₁)
         (obj-obj⁻¹ : {A′ : Obj₂} → SGFunctor.obj F (obj⁻¹ A′) ≡ A′)
  where
  open SGF₀.FullAndFaithful F isFF
  open SGF-FF-Inverse F obj⁻¹ obj-obj⁻¹
                      mor⁻¹ mor⁻¹-cong mor⁻¹-mor mor-mor⁻¹
    renaming (FFInverse to sgFunctor⁻¹)
  FFInverse : SGFunctor C₂ C₁
  FFInverse = sgFunctor⁻¹
  open SGFunctorSetup C₁ C₂
  open SGFunctor F
  FFInverse-leftInv : FFInverse ⨾⨾ F ≡F≈ Identity C₂
  FFInverse-leftInv = record
      {obj≡ = obj-obj⁻¹
      ; mor≈ = λ {A} {B} {f} → ≈₂-begin
          trg₂-contract (src₂-contract (mor (mor⁻¹′ f)))
        ≈₂⟨ ≈₂-refl ⟩
          trg₂-contract (src₂-contract (mor (mor⁻¹ (Mor₂-expand f))))
        ≈₂⟨ trg₂-contract-cong (src₂-contract-cong mor-mor⁻¹) ⟩
```

```
        trg₂-contract (src₂-contract (src₂-expand (trg₂-expand f)))
      ≈₂≡⟨ ≡-cong trg₂-contract src₂-contract-expand ⟩
        trg₂-contract (trg₂-expand f)
      ≈₂≡⟨ trg₂-contract-expand ⟩
        f
      □₂
    }
```

For showing the right-inverse property we also require a right-inverse of the object mapping obj:

```
  module _ (obj⁻¹-obj : {A : Obj₁} → obj⁻¹ (obj A) ≡ A) where
    private
      module C₁ = Semigroupoid C₁
      module C₂ = Semigroupoid C₂
    FFInverse-rightInv : F ⨾ FFInverse   ≡F≈ Identity C₁
    FFInverse-rightInv = record
      {obj≡ = obj⁻¹-obj
      ; mor≈ = λ {A} {B} {f} → ≈₁-begin
          C₁.≡-substTrg obj⁻¹-obj (C₁.≡-substSrc obj⁻¹-obj (mor⁻¹′ (mor f)))
        ≈₁⟨ ≈₁-refl ⟩
          C₁.≡-substTrg obj⁻¹-obj
            (C₁.≡-substSrc obj⁻¹-obj (mor⁻¹ (src₂-expand (trg₂-expand (mor f)))))
        ≈₁≡˘⟨ ≡-cong (C₁.≡-substTrg obj⁻¹-obj) (mor⁻¹-≡-substSrc _ obj⁻¹-obj) ⟩
          C₁.≡-substTrg obj⁻¹-obj
            (mor⁻¹ (C₂.≡-substSrc (≡-cong obj obj⁻¹-obj) (src₂-expand (trg₂-expand (mor f)))))
        ≈₁≡⟨ ≡-cong   (C₁.≡-substTrg obj⁻¹-obj ∘ mor⁻¹)
                      (C₂.≡-substSrcSrc-contract Obj₂-expand (≡-cong obj obj⁻¹-obj)) ⟩
          C₁.≡-substTrg obj⁻¹-obj (mor⁻¹ (trg₂-expand (mor f)))
        ≈₁≡˘⟨ mor⁻¹-≡-substTrg _ obj⁻¹-obj ⟩
          mor⁻¹ (C₂.≡-substTrg (≡-cong obj obj⁻¹-obj) (trg₂-expand (mor f)))
        ≈₁≡⟨ ≡-cong mor⁻¹ (C₂.≡-substTrgTrg-contract Obj₂-expand (≡-cong obj obj⁻¹-obj)) ⟩
          mor⁻¹ (mor f)
        ≈₁⟨ mor⁻¹-mor ⟩
          f
        □₁
      }
```

6.5 Categoric.SGFunctor.BasicProps

We collect basic derived SGFunctor material, that is, material not involving limits or colimits, into the sub-module IsSGFunctorBasicProps.

```
module Categoric.SGFunctor.BasicProps
    {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} {Src : Semigroupoid j₁ k₁ Obj₁}
    {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} {Trg : Semigroupoid j₂ k₂ Obj₂}
  where
  private
    module Src = Semigroupoid Src
    module Trg = Semigroupoid Trg
  open SGFunctorSetup Src Trg
  module _
    {obj : Obj₁ → Obj₂}
    {mor : {A B : Obj₁} → Mor₁ A B → Mor₂ (obj A) (obj B)}
    (IsSGF : IsSGFunctor Src Trg obj mor)
    where
```

```
module IsSGFunctorBasicProps where
  open IsSGFunctor IsSGF
  infixl 20 _(o)_ _(m)_
  _(o)_ : Obj₁ → Obj₂
  _(o)_ = obj
  _(m)_ : {A B : Obj₁} → Mor₁ A B → Mor₂ (obj A) (obj B)
  _(m)_ = mor
  lesHom : LESHom Hom₁ Hom₂
  lesHom = record {mapN = obj; mapE = mor; congE = mor-cong}
  mor-≡-substSrc : {A B : Obj₁} (F : Mor₁ A B) {A′ : Obj₁} (A≡A′ : A ≡ A′)
                 → mor (Src.≡-substSrc A≡A′ F) ≡ Trg.≡-substSrc (≡-cong obj A≡A′) (mor F)
  mor-≡-substSrc F ≡-refl = ≡-refl
  mor-≡-substTrg : {A B : Obj₁} (F : Mor₁ A B) {B′ : Obj₁} (B≡B′ : B ≡ B′)
                 → mor (Src.≡-substTrg B≡B′ F) ≡ Trg.≡-substTrg (≡-cong obj B≡B′) (mor F)
  mor-≡-substTrg F ≡-refl = ≡-refl
  mor′ : {A B : Obj₁} → Hom₁ A B ⟶ Hom₂ (obj A) (obj B)
  mor′ = record {_⟨$⟩_ = mor; cong = mor-cong}
  IsFaithful   : Set (i₁ ⊍ j₁ ⊍ j₂ ⊍ k₁ ⊍ k₂)
  IsFaithful   = {A B : Obj₁}
             → Σ [mor⁻¹ : Hom₂ (obj A) (obj B) ⟶ Hom₁ A B] mor⁻¹ LeftInverseOf mor′
  IsFull   : Set   (i₁ ⊍ j₁ ⊍ j₂ ⊍ k₁ ⊍ k₂)
  IsFull   = {A B : Obj₁}
             → Σ [mor⁻¹ : Hom₂ (obj A) (obj B) ⟶ Hom₁ A B] mor⁻¹ RightInverseOf mor′
  IsFullAndFaithful  : Set (i₁ ⊍ j₁ ⊍ j₂ ⊍ k₁ ⊍ k₂)
  IsFullAndFaithful  = {A B : Obj₁}
                     → Σ [mor⁻¹ : Hom₂ (obj A) (obj B) ⟶ Hom₁ A B] mor⁻¹ InverseOf mor′
  PreservesIdentity  :  Set (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂ ⊍ k₁ ⊍ k₂)
  PreservesIdentity  = {A : Obj₁} {I : Mor₁ A A}
                     → Src.isIdentity I → Trg.isIdentity (mor I)


  PreservesMonos : Set (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂ ⊍ k₁ ⊍ k₂)
  PreservesMonos = {A B : Obj₁} {f : Mor₁ A B} → Src.isMono f → Trg.isMono (mor f)
  PreservesEpis : Set (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂ ⊍ k₁ ⊍ k₂)
  PreservesEpis = {A B : Obj₁} {f : Mor₁ A B} → Src.isEpi f → Trg.isEpi (mor f)
```

The following argument Id₂-isIdentity could have its type called "KeepIdentity" analogous to KeepInitialObject above; the following in particular shows that it is equivalent to PreservesIdentity.

```
  module TrgIdentity (idOp : IdOp Hom₁ _⨾₁_)
                     (Id₂-isIdentity : {A : Obj₁} → Trg.isIdentity (mor (IdOp.Id idOp {A}))) where
    open IdOp idOp using (Id)
    Id₂ : {A : Obj₁} → Mor₂ (obj A) (obj A)
    Id₂ = mor Id
    leftId₂ : {A : Obj₁} → Trg.isLeftIdentity (Id₂ {A})
    leftId₂ = proj₁ Id₂-isIdentity
    rightId₂ : {A : Obj₁} → Trg.isRightIdentity (Id₂ {A})
    rightId₂ = proj₂ Id₂-isIdentity
    preservesId : PreservesIdentity
    preservesId {A} {I} I-isId = Trg.isIdentity-subst
      (≈₂-sym (mor-cong (CategoryProps.isIdentity-≈Id Src idOp I-isId)))
      Id₂-isIdentity


  module PreservedIdentity (presId : PreservesIdentity) (idOp : IdOp Hom₁ _⨾₁_) where
    open IdOp idOp using (Id)
    open CategoryProps Src idOp using (Id-isIdentity)
```

```
      Id₂-isIdentity : {A : Obj₁} → Trg.isIdentity (mor (Id {A}))
      Id₂-isIdentity = presId Id-isIdentity
      open TrgIdentity idOp Id₂-isIdentity public

    module morInv (inv : {A B : Obj₁} → Hom₂ (obj A) (obj B) ⟶ Hom₁ A B) where
      mor⁻¹ : {A B : Obj₁} → Mor₂ (obj A) (obj B) → Mor₁ A B
      mor⁻¹ = _⟨$⟩_ inv
      mor⁻¹-cong : {A B : Obj₁} {F G : Mor₂ (obj A) (obj B)}
                 → F ≈₂ G → mor⁻¹ F ≈₁ mor⁻¹ G
      mor⁻¹-cong = cong inv

    module FullAndFaithful (isFF : IsFullAndFaithful) where
      open morInv (proj₁ isFF) public
      mor-Inverse : {A B : Obj₁} → Inverse (Hom₁ A B) (Hom₂ (obj A) (obj B))
      mor-Inverse = record
        {to = mor′
        ;from = proj₁ isFF
        ;inverse-of = proj₂ isFF
        }
      mor⁻¹-mor : {A B : Obj₁} {F : Mor₁ A B} → mor⁻¹ (mor F) ≈₁ F
      mor⁻¹-mor {A} {B} {F} = _InverseOf_.left-inverse-of (proj₂ isFF) F
      mor-mor⁻¹ : {A B : Obj₁} {F : Mor₂ (obj A) (obj B)} → mor (mor⁻¹ F) ≈₂ F
      mor-mor⁻¹ {A} {B} {F} = _InverseOf_.right-inverse-of (proj₂ isFF) F
      mor⁻¹-≡-substSrc : {A B : Obj₁} (F : Mor₂ (obj A) (obj B)) {A′ : Obj₁} (A≡A′ : A ≡ A′)
                       → mor⁻¹ (Trg.≡-substSrc (≡-cong obj A≡A′) F) ≡ Src.≡-substSrc A≡A′ (mor⁻¹ F)
      mor⁻¹-≡-substSrc F ≡-refl = ≡-refl
      mor⁻¹-≡-substTrg : {A B : Obj₁} (F : Mor₂ (obj A) (obj B)) {B′ : Obj₁} (B≡B′ : B ≡ B′)
                       → mor⁻¹ (Trg.≡-substTrg (≡-cong obj B≡B′) F) ≡ Src.≡-substTrg B≡B′ (mor⁻¹ F)
      mor⁻¹-≡-substTrg F ≡-refl = ≡-refl

module SGF₀ (F : SGFunctor Src Trg) = IsSGFunctorBasicProps (SGFunctor.isSGFunctor F)
```

6.6 Categoric.SGFunctor.UpArrow

```
module Categoric.SGFunctor.UpArrow
    {ℓi₁ ℓj₁ ℓk₁ : Level} {Obj₁ : Set ℓi₁} {𝒮₁ : Semigroupoid ℓj₁ ℓk₁ Obj₁}
    {ℓi₂ ℓj₂ ℓk₂ : Level} {Obj₂ : Set ℓi₂} {𝒮₂ : Semigroupoid ℓj₂ ℓk₂ Obj₂}
    (ℱ : SGFunctor 𝒮₁ 𝒮₂) where
  open SGFunctorSetup 𝒮₁ 𝒮₂
  private
    module 𝒮₁ = Semigroupoid 𝒮₁
    module 𝒮₂ = Semigroupoid 𝒮₂
    module ℱ = SGFunctor ℱ

  _↣_ : (A : Obj₁) (B : Obj₂) → Set ℓj₂
  A ↣ B = Mor₂ (ℱ.obj A) B
  _↣′_ : (A : Obj₁) (B : Obj₂) → Setoid ℓj₂ ℓk₂
  A ↣′ B = Hom₂ (ℱ.obj A) B

  infixr 9 _⨾↣_
  _⨾↣_ : {X Y : Obj₁} {Z : Obj₂} (f : Mor₁ X Y) (U : Y ↣ Z) → X ↣ Z
```

```
_⨾↦_ f U = 𝓕.mor f ⨾₂ U
⨾↦-cong : {X Y : Obj₁} {Z : Obj₂} {f g : Mor₁ X Y} → f ≈₁ g
  → {U V : Y ↦ Z} → U ≈₂ V → f ⨾↦ U ≈₂ g ⨾↦ V
⨾↦-cong f≈g U≈V = 𝒮₂.⨾-cong (𝓕.mor-cong f≈g) U≈V
⨾↦-cong₁ : {X Y : Obj₁} {Z : Obj₂} {f g : Mor₁ X Y} {U : Y ↦ Z}
         → f ≈₁ g → f ⨾↦ U ≈₂ g ⨾↦ U
⨾↦-cong₁ f≈g = ⨾↦-cong f≈g ≈₂-refl
⨾↦-cong₂ : {X Y : Obj₁} {Z : Obj₂} {f : Mor₁ X Y} {U V : Y ↦ Z} → U ≈₂ V → f ⨾↦ U ≈₂ f ⨾↦ V
⨾↦-cong₂ U≈V = 𝒮₂.⨾-cong₂ U≈V
⨾↦-assoc : {X₁ X₂ X₃ : Obj₁} {Z : Obj₂} {f : Mor₁ X₁ X₂} {g : Mor₁ X₂ X₃} {U : X₃ ↦ Z}
         → (f ⨾↦ (g ⨾↦ U)) ≈₂ (f ⨾₁ g) ⨾↦ U
⨾↦-assoc {f = f} {g} {U} = ≈₂-begin
    f ⨾↦ (g ⨾↦ U)
  ≈₂⟨ 𝒮₂.≈-refl ⟩
    𝓕.mor f ⨾₂ (𝓕.mor g ⨾₂ U)
  ≈₂⟨ 𝒮₂.⨾-assocL ⟩
    (𝓕.mor f ⨾₂ 𝓕.mor g) ⨾₂ U
  ≈₂˘⟨ 𝒮₂.⨾-cong₁ 𝓕.mor-⨾ ⟩
    𝓕.mor (f ⨾₁ g) ⨾₂ U
  ≈₂⟨ 𝒮₂.≈-refl ⟩
    (f ⨾₁ g) ⨾↦ U
  □₂
```

For the comma categories along $\mathcal{F}$ to a target object X we introduce a variant of SliceCat defined in Categoric.Category.Slice (Sect. 4.13):

```
UpSliceObj : (X : Obj₂) → Set (ℓi₁ ⊍ ℓj₂)
UpSliceObj X = Σ A : Obj₁ • A ↦ X
UpSliceMor : {X : Obj₂} → UpSliceObj X → UpSliceObj X → Set (ℓj₁ ⊍ ℓk₂)
UpSliceMor (A₁, f₁) (A₂, f₂) = Σ g : Mor₁ A₁ A₂ • g ⨾↦ f₂ ≈₂ f₁
UpSliceSG : (X : Obj₂) → Semigroupoid (ℓj₁ ⊍ ℓk₂) ℓk₁ (UpSliceObj X)
UpSliceSG X = record
  {Hom = λ A₁ A₂ → record
    {Carrier = UpSliceMor A₁ A₂
    ; _≈_ = λ G₁ G₂ → proj₁ G₁ ≈₁ proj₁ G₂
    ; isEquivalence = record {refl = ≈₁-refl; sym = ≈₁-sym; trans = ≈₁-trans}
    }
  ; compOp = record
    {_⨾_ = λ {{A₁, f₁} {A₂, f₂} {A₃, f₃} (g, g⨾↦f₂≈f₁) (h, h⨾↦f₃≈f₂) → (g ⨾₁ h), (≈₂-begin
        (g ⨾₁ h) ⨾↦ f₃
      ≈₂⟨ ⨾↦-assoc ⟨≈₂˘≈⟩ ⨾↦-cong₂ h⨾↦f₃≈f₂ ⟩
        g ⨾↦ f₂
      ≈₂⟨ g⨾↦f₂≈f₁ ⟩
        f₁
      □₂)}
    ; ⨾-cong = 𝒮₁.⨾-cong
    ; ⨾-assoc = 𝒮₁.⨾-assoc
    }
  }
```

6.7 Categoric.SGFunctor.Coproduct

```
module Categoric.SGFunctor.Coproduct
  {ℓi₁ ℓj₁ ℓk₁ : Level} {Obj₁ : Set ℓi₁} {𝒮₁ : Semigroupoid ℓj₁ ℓk₁ Obj₁}
  {ℓi₂ ℓj₂ ℓk₂ : Level} {Obj₂ : Set ℓi₂} {𝒮₂ : Semigroupoid ℓj₂ ℓk₂ Obj₂}
```

```
  (𝓕 : SGFunctor 𝓢₁ 𝓢₂)
where
  private
    module 𝓢₁ = Semigroupoid 𝓢₁
    module 𝓢₂ = Semigroupoid 𝓢₂
  open SGFunctorSetup 𝓢₁ 𝓢₂
  open SGFunctor 𝓕
  open import Categoric.SGFunctor.UpArrow 𝓕


  PreservesCoproduct    : Set (ℓi₁ ⊍ ℓi₂ ⊍ ℓj₁ ⊍ ℓj₂ ⊍ ℓk₁ ⊍ ℓk₂)
  PreservesCoproduct    = {A B S : Obj₁} {ι : Mor₁ A S} {κ : Mor₁ B S}
                        → IsCoproduct 𝓢₁ ι κ → IsCoproduct 𝓢₂ (mor ι) (mor κ)


  module PreservedCoproduct
      {A B S : Obj₁} {ι : Mor₁ A S} {κ : Mor₁ B S} (isCoproduct : IsCoproduct 𝓢₁ ι κ)
      (isCoproduct₂ : IsCoproduct 𝓢₂ {obj A} {obj B} (mor ι) (mor κ)) where
    open IsCoproduct 𝓢₁ isCoproduct

    S₂ = obj S
    ι₂ : A ↝ S₂
    ι₂ = mor ι
    κ₂ : B ↝ S₂
    κ₂ = mor κ

    private
      𝓢₂′ = retractSemigroupoid obj 𝓢₂
      module 𝓢₂′ = Semigroupoid 𝓢₂′

      -- We cannot get material involving _⨹_ from IsCoproduct₂
      -- since the first two implicit arguments need to be from Obj₁, not from Obj₂.
    open IsCoproduct₂ 𝓢₂′ (retractIsCoproduct 𝓢₂ obj isCoproduct₂) public using (Id⊞₂)

    infixr 5 _⨹₂_
    _⨹₂_ : {X : Obj₂} (f₁ : A ↝ X) (f₂ : B ↝ X) → Mor₂ S₂ X
    _⨹₂_ f₁ f₂ = CoCone2Univ.univMor 𝓢₂ (isCoproduct₂ f₁ f₂)

    ι⨾⨹₂ : {C : Obj₂} {F : A ↝ C} {G : B ↝ C} → ι₂ ⨾₂ (F ⨹₂ G) ≈₂ F
    ι⨾⨹₂ {_} {F} {G} = CoCone2Univ.univMor-factors-left 𝓢₂ (isCoproduct₂ F G)
    κ⨾⨹₂ : {C : Obj₂} {F : A ↝ C} {G : B ↝ C} → κ₂ ⨾₂ (F ⨹₂ G) ≈₂ G
    κ⨾⨹₂ {_} {F} {G} = CoCone2Univ.univMor-factors-right 𝓢₂ (isCoproduct₂ F G)

    ⨹₂-unique     : {C : Obj₂} {F : A ↝ C} {G : B ↝ C} {U : Mor₂ S₂ C}
                  → ι₂ ⨾₂ U ≈₂ F → κ₂ ⨾₂ U ≈₂ G → U ≈₂ F ⨹₂ G
    ⨹₂-unique {_} {F} {G} {U} ι⨾U≈F κ⨾U≈G = CoCone2Univ.univMor-unique 𝓢₂
                                              (isCoproduct₂ F G) ι⨾U≈F κ⨾U≈G

    ⨹₂-cong    :    {C : Obj₂} {F₁ F₂ : A ↝ C} → F₁ ≈₂ F₂
               →           {G₁ G₂ : B ↝ C} → G₁ ≈₂ G₂
               → F₁ ⨹₂ G₁ ≈₂ F₂ ⨹₂ G₂
    ⨹₂-cong {F₁ = F₁} {F₂} F₁≈F₂ {G₁} {G₂} G₁≈G₂ = ⨹₂-unique {_} {F₂} {G₂} {F₁ ⨹₂ G₁}
      (ι⨾⨹₂ ⟨≈₂≈⟩ F₁≈F₂) (κ⨾⨹₂ ⟨≈₂≈⟩ G₁≈G₂)

    ⨹₂-cong₁    : {C : Obj₂} {F₁ F₂ : A ↝ C} {G : B ↝ C}
                → F₁ ≈₂ F₂ → F₁ ⨹₂ G ≈₂ F₂ ⨹₂ G
    ⨹₂-cong₁ F₁≈F₂ = ⨹₂-cong F₁≈F₂ 𝓢₂.≈-refl

    ⨹₂-cong₂    : {C : Obj₂} {F : A ↝ C} {G₁ G₂ : B ↝ C}
                → G₁ ≈₂ G₂ → F ⨹₂ G₁ ≈₂ F ⨹₂ G₂
    ⨹₂-cong₂ G₁≈G₂ = ⨹₂-cong 𝓢₂.≈-refl G₁≈G₂

    ⨹₂-⨾   :    {C D : Obj₂} {F₁ : A ↝ C} {F₂ : B ↝ C} {G : Mor₂ C D}
            → (F₁ ⨹₂ F₂) ⨾₂ G ≈₂ F₁ ⨾₂ G ⨹₂ F₂ ⨾₂ G
    ⨹₂-⨾ {F₁ = F₁} {F₂} {G} = ⨹₂-unique {_} {F₁ ⨾₂ G} {F₂ ⨾₂ G} {(F₁ ⨹₂ F₂) ⨾₂ G}
      (≈₂-begin
            ι₂ ⨾₂ (F₁ ⨹₂ F₂) ⨾₂ G
```

```
        ≈₂⟨ 𝒮₂.⨾-assocL ⟨≈₂≈⟩ 𝒮₂.⨾-cong₁ ι⨾⨹₂ ⟩
              F₁ ⨾₂ G
        □₂)
     (≈₂-begin
              κ₂ ⨾₂ (F₁ ⨹₂ F₂) ⨾₂ G
        ≈₂⟨ 𝒮₂.⨾-assocL ⟨≈₂≈⟩ 𝒮₂.⨾-cong₁ κ⨾⨹₂ ⟩
              F₂ ⨾₂ G
        □₂)
  mor-⨹     :  {C : Obj₁} {F₁ : Mor₁ A C} {F₂ : Mor₁ B C}
               → mor (F₁ ⨹ F₂) ≈₂ mor F₁ ⨹₂ mor F₂
  mor-⨹ {F₁ = F₁} {F₂} = ⨹₂-unique {_} {mor F₁} {mor F₂} {mor (F₁ ⨹ F₂)}
     (≈₂-begin
              ι₂ ⨾₂ mor (F₁ ⨹ F₂)
        ≈₂˘⟨ mor-⨾ ⟩
              mor (ι ⨾₁ (F₁ ⨹ F₂))
        ≈₂⟨ mor-cong ι⨾⨹ ⟩
              mor F₁
        □₂)
     (≈₂-begin
              κ₂ ⨾₂ mor (F₁ ⨹ F₂)
        ≈₂˘⟨ mor-⨾ ⟩
              mor (κ ⨾₁ (F₁ ⨹ F₂))
        ≈₂⟨ mor-cong κ⨾⨹ ⟩
              mor F₂
        □₂)
  Id⊞₂-isLeftIdentity : 𝒮₂.isLeftIdentity Id⊞₂
  Id⊞₂-isLeftIdentity {X} {R} = ≈₂-begin
        Id⊞₂ ⨾₂ R
     ≈₂⟨ ⨹₂-⨾ ⟩
        ι₂ ⨾₂ R ⨹₂ κ₂ ⨾₂ R
     ≈₂˘⟨ ⨹₂-unique {_} {ι₂ ⨾₂ R} {κ₂ ⨾₂ R} {R} 𝒮₂.≈-refl 𝒮₂.≈-refl ⟩
        R
     □₂
  ⦶≈⊞₂   :    ({X : Obj₂} {F G : A ↝ X} → F ≈₂ G)
          →   ({X : Obj₂} {F G : B ↝ X} → F ≈₂ G)
          →   ({X : Obj₂} {F G : S ↝ X} → F ≈₂ G)
  ⦶≈⊞₂ ⦶≈₁ ⦶≈₂ {X} {F} {G} = ≈₂-begin
              F
        ≈₂⟨ Id⊞₂-isLeftIdentity ⟨≈₂˘≈⟩ ⨹₂-⨾ ⟩
              ι₂ ⨾₂ F ⨹₂ κ₂ ⨾₂ F
        ≈₂⟨ ⨹₂-cong ⦶≈₁ ⦶≈₂ ⟩
              ι₂ ⨾₂ G ⨹₂ κ₂ ⨾₂ G
        ≈₂⟨ ⨹₂-⨾ ⟨≈₂˘≈⟩ Id⊞₂-isLeftIdentity ⟩
              G
        □₂
```

```
KeepCoproducts : HasCoproducts 𝒮₁ → Set (ℓi₁ ⊍ ℓi₂ ⊍ ℓj₂ ⊍ ℓk₂)
KeepCoproducts hasCoproducts      =  (A B : Obj₁)
                                  → IsCoproduct 𝒮₂ (mor (ι {A} {B})) (mor (κ {A} {B}))
  where open HasCoproducts 𝒮₁ hasCoproducts using (ι; κ)
```

```
module TrgCoproducts (hasCoproducts      : HasCoproducts 𝒮₁)
                     (trgCoproduct       : KeepCoproducts hasCoproducts)
  where
  open HasCoproducts 𝒮₁ hasCoproducts
  _⊞₂_ : (A₁ A₂ : Obj₁) → Obj₂
  A₁ ⊞₂ A₂  =  obj (A₁ ⊞ A₂)
```

```
module _ {A B : Obj₁} where
  isCoproduct₂ : IsCoproduct 𝒮₂ {obj A} {obj B} (mor ι) (mor κ)
  isCoproduct₂ = trgCoproduct A B

  open PreservedCoproduct isCoproduct isCoproduct₂ public
private
  𝒮₂′ = retractSemigroupoid obj 𝒮₂
  module 𝒮₂′ = Semigroupoid 𝒮₂′
hasCoproducts₂ : HasCoproducts 𝒮₂′
hasCoproducts₂ = record {_⊞_ = _⊞_; ι = ι₂; κ = κ₂
                        ; isCoproduct = retractIsCoproduct 𝒮₂ obj isCoproduct₂}
  -- We cannot get material involving _⟁_ from hasCoproducts₂
  -- because we want the target object X (see below) to be arbitrary from Obj₂.
open HasCoproductsLocalProps₂ 𝒮₂′ hasCoproducts₂ public
⊕₂-⨾-⟁₂      :   {A₁ B₁ A₂ B₂ : Obj₁} {C : Obj₂} {F₁ : 𝒮₂′.Mor A₁ B₁} {G₁ : B₁ ↝ C}
             →   {F₂ : 𝒮₂′.Mor A₂ B₂} {G₂ : B₂ ↝ C}
             →   (F₁ ⊕₂ F₂) ⨾₂ (G₁ ⟁₂ G₂) ≈₂ F₁ ⨾₂ G₁ ⟁₂ F₂ ⨾₂ G₂
⊕₂-⨾-⟁₂ {F₁ = F₁} {G₁} {F₂} {G₂} = ⟁₂-unique {F = F₁ ⨾₂ G₁} {F₂ ⨾₂ G₂}
  {(F₁ ⊕₂ F₂) ⨾₂ (G₁ ⟁₂ G₂)}
  (≈₂-begin
          ι₂ ⨾₂ ((F₁ ⊕₂ F₂) ⨾₂ (G₁ ⟁₂ G₂))
    ≈₂⟨ 𝒮₂.⨾-cong₁&₂₁ ι⨾⊕₂ ⟩
          F₁ ⨾₂ ι₂ ⨾₂ (G₁ ⟁₂ G₂)
    ≈₂⟨ 𝒮₂.⨾-cong₂ ι⨾⟁₂ ⟩
          F₁ ⨾₂ G₁
    □₂)
  (≈₂-begin
          κ₂ ⨾₂ ((F₁ ⊕₂ F₂) ⨾₂ (G₁ ⟁₂ G₂))
    ≈₂⟨ 𝒮₂.⨾-cong₁&₂₁ κ⨾⊕₂ ⟩
          F₂ ⨾₂ κ₂ ⨾₂ (G₁ ⟁₂ G₂)
    ≈₂⟨ 𝒮₂.⨾-cong₂ κ⨾⟁₂ ⟩
          F₂ ⨾₂ G₂
    □₂)
mor-⊕      : {A₁ A₂ B₁ B₂ : Obj₁} {F₁ : Mor₁ A₁ B₁} {F₂ : Mor₁ A₂ B₂}
             → mor (F₁ ⊕ F₂) ≈₂ mor F₁ ⊕₂ mor F₂
mor-⊕ {F₁ = F₁} {F₂} = ≈₂-begin
          mor (F₁ ⊕ F₂)
  ≈₂⟨ mor-⟁ ⟩
          mor (F₁ ⨾₁ ι) ⟁₂ mor (F₂ ⨾₁ κ)
  ≈₂⟨ ⟁₂-cong mor-⨾ mor-⨾ ⟩
          mor F₁ ⊕₂ mor F₂
    □₂
mor-⊞-assoc : {A B C : Obj₁} → mor ⊞-assoc ≈₂ ⊞₂-assoc {A} {B} {C}
mor-⊞-assoc = ≈₂-begin
                    mor ⊞-assoc
                ≈₂⟨ 𝒮₂.≈-refl ⟩
                    mor ((ι ⟁ ι ⨾₁ κ) ⟁ (κ ⨾₁ κ))
                ≈₂⟨ mor-⟁ ⟩
                    mor (ι ⟁ ι ⨾₁ κ) ⟁₂ mor (κ ⨾₁ κ)
                ≈₂⟨ (⟁₂-cong mor-⟁ mor-⨾) ⟩
                    (mor ι ⟁₂ mor (ι ⨾₁ κ)) ⟁₂ mor κ ⨾₂ mor κ
                ≈₂⟨ ⟁₂-cong₁ (⟁₂-cong₂ mor-⨾) ⟩
                    (ι₂ ⟁₂ mor ι ⨾₂ mor κ) ⟁₂ κ₂ ⨾₂ κ₂
                ≈₂⟨ 𝒮₂.≈-refl ⟩
                    (ι₂ ⟁₂ ι₂ ⨾₂ κ₂) ⟁₂ κ₂ ⨾₂ κ₂
                □₂
mor-⊞-assocL : {A B C : Obj₁} → mor ⊞-assocL ≈₂ ⊞₂-assocL {A} {B} {C}
mor-⊞-assocL = ≈₂-begin
```

```
                     mor ⊞-assocL
                   ≈₂⟨ 𝒮₂.≈-refl ⟩
                     mor (ι ⨾₁ ι ◬ (κ ⨾₁ ι ◬ κ))
                   ≈₂⟨ mor-◬ ⟩
                     mor (ι ⨾₁ ι) ◬₂ mor (κ ⨾₁ ι ◬ κ)
                   ≈₂⟨ (◬₂-cong mor-⨾ mor-◬) ⟩
                     mor ι ⨾₂ mor ι ◬₂ (mor (κ ⨾₁ ι) ◬₂ mor κ)
                   ≈₂⟨ ◬₂-cong₂ (◬₂-cong₁ mor-⨾) ⟩
                     ι₂ ⨾₂ ι₂ ◬₂ (mor κ ⨾₂ mor ι ◬₂ κ₂)
                   ≈₂⟨ 𝒮₂.≈-refl ⟩
                     ι₂ ⨾₂ ι₂ ◬₂ (κ₂ ⨾₂ ι₂ ◬₂ κ₂)
                   □₂
⊞₂-assoc-◬◬ : {A B C : Obj₁} {D : Obj₂} {F : A ↛ D} {G : B ↛ D} {H : C ↛ D}
                   → ⊞₂-assoc ⨾₂ (F ◬₂ (G ◬₂ H)) ≈₂ (F ◬₂ G) ◬₂ H
⊞₂-assoc-◬◬ {F = F} {G} {H} = ≈₂-begin
      ⊞₂-assoc ⨾₂ (F ◬₂ (G ◬₂ H))
   ≈₂⟨ ◬₂-⨾ ⟩
      (ι₂ ◬₂ ι₂ ⨾₂ κ₂) ⨾₂ (F ◬₂ (G ◬₂ H)) ◬₂ (κ₂ ⨾₂ κ₂) ⨾₂ (F ◬₂ (G ◬₂ H))
   ≈₂⟨ ◬₂-cong ◬₂-⨾ (𝒮₂.⨾-assoc ⟨≈₂≈⟩ 𝒮₂.⨾-cong₂ κ⨾◬₂) ⟩
      (ι₂ ⨾₂ (F ◬₂ (G ◬₂ H)) ◬₂ (ι₂ ⨾₂ κ₂) ⨾₂ (F ◬₂ (G ◬₂ H))) ◬₂ κ₂ ⨾₂ (G ◬₂ H)
   ≈₂⟨ ◬₂-cong (◬₂-cong ι⨾◬₂ (𝒮₂.⨾-assoc ⟨≈₂≈⟩ 𝒮₂.⨾-cong₂ κ⨾◬₂)) κ⨾◬₂ ⟩
      (F ◬₂ ι₂ ⨾₂ (G ◬₂ H)) ◬₂ H
   ≈₂⟨ ◬₂-cong₁     (◬₂-cong₂ ι⨾◬₂) ⟩
      (F ◬₂ G) ◬₂ H
   □₂
⊞₂-assocL-◬◬ : {A B C : Obj₁} {D : Obj₂} {F : A ↛ D} {G : B ↛ D} {H : C ↛ D}
   →                  ⊞₂-assocL ⨾₂ ((F ◬₂ G) ◬₂ H) ≈₂ F ◬₂ (G ◬₂ H)
⊞₂-assocL-◬◬ {F = F} {G} {H} = ≈₂-begin
      ⊞₂-assocL ⨾₂ ((F ◬₂ G) ◬₂ H)
   ≈₂˘⟨ 𝒮₂.⨾-cong₂ ⊞₂-assoc-◬◬ ⟩
      ⊞₂-assocL ⨾₂ ⊞₂-assoc ⨾₂ (F ◬₂ (G ◬₂ H))
   ≈₂⟨ 𝒮₂.⨾-assocL ⟨≈₂≈⟩ 𝒮₂.⨾-cong₁ ⊞₂-assocL-assoc ⟩
      Id⊞₂ ⨾₂ (F ◬₂ (G ◬₂ H))
   ≈₂⟨ Id⊞₂-isLeftIdentity ⟩
      F ◬₂ (G ◬₂ H)
   □₂
mor-⊞-transpose₂ : {A B C D : Obj₁} → mor ⊞-transpose₂ ≈₂ ⊞₂-transpose₂ {A} {B} {C} {D}
mor-⊞-transpose₂ = ≈₂-begin
                     mor ⊞-transpose₂
                   ≈₂⟨ 𝒮₂.≈-refl ⟩
                     mor ((ι ⨾₁ ι ◬ ι ⨾₁ κ) ◬ (κ ⨾₁ ι ◬ κ ⨾₁ κ))
                   ≈₂⟨ mor-◬ ⟩
                     mor (ι ⨾₁ ι ◬ ι ⨾₁ κ) ◬₂ mor (κ ⨾₁ ι ◬ κ ⨾₁ κ)
                   ≈₂⟨ (◬₂-cong mor-◬ mor-◬) ⟩
                     (mor (ι ⨾₁ ι) ◬₂ mor (ι ⨾₁ κ)) ◬₂ (mor (κ ⨾₁ ι) ◬₂ mor (κ ⨾₁ κ))
                   ≈₂⟨ ◬₂-cong (◬₂-cong mor-⨾ mor-⨾) (◬₂-cong mor-⨾ mor-⨾) ⟩
                     (ι₂ ⨾₂ ι₂ ◬₂ ι₂ ⨾₂ κ₂) ◬₂ (κ₂ ⨾₂ ι₂ ◬₂ κ₂ ⨾₂ κ₂)
                   □₂
⊞₂-transpose₂-◬◬ :  {A B C D : Obj₁} {E : Obj₂}
                        {F : A ↛ E} {G : B ↛ E} {H : C ↛ E} {I : D ↛ E}
                    →  ⊞₂-transpose₂ ⨾₂ ((F ◬₂ G) ◬₂ (H ◬₂ I)) ≈₂ (F ◬₂ H) ◬₂ (G ◬₂ I)
⊞₂-transpose₂-◬◬ {F = F} {G} {H} {I} = ≈₂-begin
           ⊞₂-transpose₂ ⨾₂ ((F ◬₂ G) ◬₂ (H ◬₂ I))
   ≈₂⟨ ◬₂-⨾ ⟩
             (ι₂ ⨾₂ ι₂ ◬₂ ι₂ ⨾₂ κ₂) ⨾₂ ((F ◬₂ G) ◬₂ (H ◬₂ I))
           ◬₂ (κ₂ ⨾₂ ι₂ ◬₂ κ₂ ⨾₂ κ₂) ⨾₂ ((F ◬₂ G) ◬₂ (H ◬₂ I))
   ≈₂⟨ ◬₂-cong ◬₂-⨾ ◬₂-⨾ ⟩
             ((ι₂ ⨾₂ ι₂) ⨾₂ ((F ◬₂ G) ◬₂ (H ◬₂ I))
```

```
                    ⩓₂ (ι₂ ⨾₂ κ₂) ⨾₂ ((F ⩓₂ G) ⩓₂ (H ⩓₂ I))
                    )
                ⩓₂ ((κ₂ ⨾₂ ι₂) ⨾₂ ((F ⩓₂ G) ⩓₂ (H ⩓₂ I))
                    ⩓₂ (κ₂ ⨾₂ κ₂) ⨾₂ ((F ⩓₂ G) ⩓₂ (H ⩓₂ I))
                    )
        ≈₂⟨ ⩓₂-cong (⩓₂-cong (𝒮₂.⨾-assoc ⟨≈₂≈⟩ 𝒮₂.⨾-cong₂ ι⨾⩓₂)
                                (𝒮₂.⨾-assoc ⟨≈₂≈⟩ 𝒮₂.⨾-cong₂ κ⨾⩓₂))
                    (⩓₂-cong (𝒮₂.⨾-assoc ⟨≈₂≈⟩ 𝒮₂.⨾-cong₂ ι⨾⩓₂)
                                (𝒮₂.⨾-assoc ⟨≈₂≈⟩ 𝒮₂.⨾-cong₂ κ⨾⩓₂)) ⟩
            (ι₂ ⨾₂ (F ⩓₂ G) ⩓₂ ι₂ ⨾₂ (H ⩓₂ I)) ⩓₂ (κ₂ ⨾₂ (F ⩓₂ G) ⩓₂ κ₂ ⨾₂ (H ⩓₂ I))
        ≈₂⟨ ⩓₂-cong (⩓₂-cong ι⨾⩓₂ ι⨾⩓₂) (⩓₂-cong κ⨾⩓₂ κ⨾⩓₂) ⟩
            (F ⩓₂ H) ⩓₂ (G ⩓₂ I)
        □₂
```

For convenience, we add a direct alias in analogy with keptInitialObject:

```
  keptCoproducts = TrgCoproducts.hasCoproducts₂

  module PreservedCoproducts (presCoproduct : PreservesCoproduct)
                             (hasCoproducts : HasCoproducts 𝒮₁) where
    open HasCoproducts 𝒮₁ hasCoproducts using (isCoproduct)
    open TrgCoproducts hasCoproducts (λ A B → presCoproduct (isCoproduct {A} {B})) public
```

6.8 Categoric.SGFunctor.Pushout

```
module Categoric.SGFunctor.Pushout
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (Src : Semigroupoid j₁ k₁ Obj₁)
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (Trg : Semigroupoid j₂ k₂ Obj₂)
  where
private
  module Src = Semigroupoid Src
  module Trg = Semigroupoid Trg
open SGFunctorSetup Src Trg
module SGF-Pushout
  (obj : Obj₁ → Obj₂)
  (mor : {A B : Obj₁} → Mor₁ A B → Mor₂ (obj A) (obj B))
  (mor-cong : {A B : Obj₁} → {f g : Mor₁ A B} → f ≈₁ g → mor f ≈₂ mor g)
  (mor-⨾ : {A B C : Obj₁} → {f : Mor₁ A B} → {g : Mor₁ B C}
        → mor (f ⨾₁ g) ≈₂ mor f ⨾₂ mor g)
  where

  PreservesPushout : Set (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂ ⊍ k₁ ⊍ k₂)
  PreservesPushout = {A B C D : Obj₁} {F : Mor₁ A B} {G : Mor₁ A C} {R : Mor₁ B D} {S : Mor₁ C D}
                   → IsPushout Src F G R S → IsPushout Trg (mor F) (mor G) (mor R) (mor S)

  module PreservedPushout
      {A B C D : Obj₁} {F : Mor₁ A B} {G : Mor₁ A C} {R : Mor₁ B D} {S : Mor₁ C D}
      (isPushout : IsPushout Src F G R S)
      (isPushout₂ : IsPushout Trg {obj A} {obj B} {obj C} {obj D} (mor F) (mor G) (mor R) (mor S))
    where
    open IsPushout Src isPushout
    open IsPushout₂ Trg isPushout₂ public
    PO-obj₂ : Obj₂
    PO-obj₂ = obj D
```

```
    PO-left₂ : Mor₂ (obj B) (obj D)
    PO-left₂ = mor R
    PO-right₂ : Mor₂ (obj C) (obj D)
    PO-right₂ = mor S

  KeepsPushouts : HasPushouts Src → Set (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂ ⊍ k₂)
  KeepsPushouts hasPO = {A B C : Obj₁} (F : Mor₁ A B) (G : Mor₁ A C)
                       → let module PO = Pushout Src (hasPO F G) in
                         IsPushout Trg {obj A} {obj B} {obj C} (mor F) (mor G) (mor PO.left) (mor PO.right)
  module TrgPushouts (hasPO : HasPushouts Src) (trgPO : KeepsPushouts hasPO) where
    module _ {A B C : Obj₁} (F : Mor₁ A B) (G : Mor₁ A C) where
      private module PO = Pushout Src (hasPO F G)
      isPushout₂ : IsPushout Trg (mor F) (mor G) (mor PO.left) (mor PO.right)
      isPushout₂ = trgPO F G
      open PreservedPushout PO.prf isPushout₂ public

  module PreservedPushouts (presPO : PreservesPushout) (hasPO : HasPushouts Src) where
    open TrgPushouts hasPO (λ F G → presPO (Pushout.prf Src (hasPO F G))) public
```

6.9 Categoric.SGFunctor.Inverse

```
module Categoric.SGFunctor.Inverse
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} {Src : Semigroupoid j₁ k₁ Obj₁}
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} {Trg : Semigroupoid j₂ k₂ Obj₂}
  (F : SGFunctor Src Trg)
  where
private
  module Src = Semigroupoid Src
  module Trg = Semigroupoid Trg
open SGFunctorSetup Src Trg
open SGFunctor F
```

Instead of properly defining category equivalence, we just define inverse functors, with the object mappings mutually inverse with respect to propositional equality, since this is sufficient for our current applications, but still stretches the current abilities of Agda.

We start from the material available in a full and faithful functor, and additionally an inverse of obj:

```
module SGF-FF-Inverse
  (obj⁻¹ : Obj₂ → Obj₁)
  (obj-obj⁻¹ : {A′ : Obj₂} → obj (obj⁻¹ A′) ≡ A′)
  (mor⁻¹ : {A B : Obj₁} → Mor₂ (obj A) (obj B) → Mor₁ A B)
  (mor⁻¹-cong : {A B : Obj₁} → {f′ g′ : Mor₂ (obj A) (obj B)} → f′ ≈₂ g′ → mor⁻¹ f′ ≈₁ mor⁻¹ g′)
  (mor-leftInverse : {A B : Obj₁} {f : Mor₁ A B} → mor⁻¹ (mor f) ≈₁ f)
  (mor-rightInverse : {A B : Obj₁} → {f′ : Mor₂ (obj A) (obj B)} → mor (mor⁻¹ f′) ≈₂ f′)
  where
```

With this, it is easy to show that mor⁻¹ preserves composition:

```
mor⁻¹-⨾ : {A B C : Obj₁} {f′ : Mor₂ (obj A) (obj B)} {g′ : Mor₂ (obj B) (obj C)}
  → mor⁻¹ (f′ ⨾₂ g′) ≈₁ mor⁻¹ f′ ⨾₁ mor⁻¹ g′
mor⁻¹-⨾ {_} {_} {_} {f′} {g′} = ≈₁-begin
    mor⁻¹ (f′ ⨾₂ g′)
  ≈₁˘⟨ mor⁻¹-cong (Trg.⨾-cong mor-rightInverse mor-rightInverse) ⟩
```

```
    mor⁻¹ (mor (mor⁻¹ f′) ⨾₂ mor (mor⁻¹ g′))
  ≈₁˘⟨ mor⁻¹-cong mor-⨾ ⟩
    mor⁻¹ (mor (mor⁻¹ f′ ⨾₁ mor⁻¹ g′))
  ≈₁⟨ mor-leftInverse ⟩
    mor⁻¹ f′ ⨾₁ mor⁻¹ g′
  □₁
```

However, mor^{-1}-⨾ does not have the right type for being used directly in the inverse SGFunctor.

For being able to adapt the types of mor^{-1}, mor^{-1}-cong and mor^{-1}-⨾ for direct use in the inverse SGFunctor we first introduce the auxiliary function Obj_2-expand that exposes obj applications in target objects, and Mor_2-expand which applies this to the types of target morphisms:

```
Obj₂-contract : {A′ : Obj₂} → obj (obj⁻¹ A′) ≡ A′
Obj₂-contract = obj-obj⁻¹

src₂-contract : {A′ B′ : Obj₂} → Mor₂ (obj (obj⁻¹ A′)) B′ → Mor₂ A′ B′
src₂-contract = Trg.≡-substSrc Obj₂-contract

src₂-contract-cong : {A′ B′ : Obj₂} {f′ g′ : Mor₂ (obj (obj⁻¹ A′)) B′}
                   → f′ ≈₂ g′ → src₂-contract f′ ≈₂ src₂-contract g′
src₂-contract-cong = Trg.≡-substSrc-cong Obj₂-contract

trg₂-contract : {A′ B′ : Obj₂} → Mor₂ A′ (obj (obj⁻¹ B′)) → Mor₂ A′ B′
trg₂-contract = Trg.≡-substTrg Obj₂-contract

trg₂-contract-cong : {A′ B′ : Obj₂} {f′ g′ : Mor₂ A′ (obj (obj⁻¹ B′))}
                   → f′ ≈₂ g′ → trg₂-contract f′ ≈₂ trg₂-contract g′
trg₂-contract-cong = Trg.≡-substTrg-cong Obj₂-contract

Obj₂-expand : {A′ : Obj₂} → A′ ≡ obj (obj⁻¹ A′)
Obj₂-expand = ≡-sym obj-obj⁻¹

src₂-expand : {A′ B′ : Obj₂} → Mor₂ A′ B′ → Mor₂ (obj (obj⁻¹ A′)) B′
src₂-expand = Trg.≡-substSrc Obj₂-expand

src₂-expand-cong   : {A′ B′ : Obj₂} {f′ g′ : Mor₂ A′ B′}
                   → f′ ≈₂ g′ → src₂-expand f′ ≈₂ src₂-expand g′
src₂-expand-cong = Trg.≡-substSrc-cong Obj₂-expand

trg₂-expand : {A′ B′ : Obj₂} → Mor₂ A′ B′ → Mor₂ A′ (obj (obj⁻¹ B′))
trg₂-expand = Trg.≡-substTrg Obj₂-expand

trg₂-expand-cong   : {A′ B′ : Obj₂} {f′ g′ : Mor₂ A′ B′}
                   → f′ ≈₂ g′ → trg₂-expand f′ ≈₂ trg₂-expand g′
trg₂-expand-cong = Trg.≡-substTrg-cong Obj₂-expand


src₂-expand-contract : {A′ B′ : Obj₂} {f′ : Mor₂ (obj (obj⁻¹ A′)) B′}
  → src₂-expand (src₂-contract f′) ≡ f′
src₂-expand-contract = Trg.≡-substSrcSrc-contract Obj₂-contract Obj₂-expand

trg₂-expand-contract : {A′ B′ : Obj₂} {f′ : Mor₂ A′ (obj (obj⁻¹ B′))}
  → trg₂-expand (trg₂-contract f′) ≡ f′
trg₂-expand-contract = Trg.≡-substTrgTrg-contract Obj₂-contract Obj₂-expand


src₂-contract-expand : {A′ B′ : Obj₂} {f′ : Mor₂ A′ B′}
  → src₂-contract (src₂-expand f′) ≡ f′
src₂-contract-expand = Trg.≡-substSrcSrc-contract Obj₂-expand Obj₂-contract

trg₂-contract-expand : {A′ B′ : Obj₂} {f′ : Mor₂ A′ B′}
  → trg₂-contract (trg₂-expand f′) ≡ f′
trg₂-contract-expand = Trg.≡-substTrgTrg-contract Obj₂-expand Obj₂-contract


Mor₂-expand : {A′ B′ : Obj₂} → Mor₂ A′ B′ → Mor₂ (obj (obj⁻¹ A′)) (obj (obj⁻¹ B′))
Mor₂-expand = src₂-expand ∘ trg₂-expand
```

```
Mor₂-expand-cong : {A′ B′ : Obj₂} {f′ g′ : Mor₂ A′ B′}
                 → f′ ≈₂ g′ → Mor₂-expand f′ ≈₂ Mor₂-expand g′
Mor₂-expand-cong f′≈g′ = src₂-expand-cong (trg₂-expand-cong f′≈g′)
```

Mor₂-expand distributes over composition:

```
Mor₂-expand-⨾ : {A′ B′ C′ : Obj₂} {f′ : Mor₂ A′ B′} {g′ : Mor₂ B′ C′}
  → Mor₂-expand (f′ ⨾₂ g′) ≡ Mor₂-expand f′ ⨾₂ Mor₂-expand g′
Mor₂-expand-⨾ {f′ = f′} {g′} = ≡-begin
    src₂-expand (trg₂-expand (f′ ⨾₂ g′))
  ≡˘⟨ ≡-cong (src₂-expand) (Trg.⨾-≡-subst₃ Obj₂-expand) ⟩
    src₂-expand (f′ ⨾₂ trg₂-expand g′)
  ≡˘⟨ Trg.⨾-≡-subst₁ Obj₂-expand ⟩
    src₂-expand f′ ⨾₂ trg₂-expand g′
  ≡˘⟨ Trg.⨾-≡-subst₂ Obj₂-expand ⟩
    trg₂-expand (src₂-expand f′) ⨾₂ src₂-expand (trg₂-expand g′)
  ≡⟨ ≡-cong (flip _⨾₂_) (Trg.≡-substTrgSrc Obj₂-expand Obj₂-expand) ⟩
    src₂-expand (trg₂-expand f′) ⨾₂ src₂-expand (trg₂-expand g′)
  ≡∎
```

Now we can define the constituents of the inverse SGFunctor:

```
mor⁻¹′ : {A′ B′ : Obj₂} → Mor₂ A′ B′ → Mor₁ (obj⁻¹ A′) (obj⁻¹ B′)
mor⁻¹′ f′ = mor⁻¹ (Mor₂-expand f′)
mor⁻¹′-cong : {A′ B′ : Obj₂} {f′ g′ : Mor₂ A′ B′} → f′ ≈₂ g′ → mor⁻¹′ f′ ≈₁ mor⁻¹′ g′
mor⁻¹′-cong f′≈g′ = mor⁻¹-cong (Mor₂-expand-cong f′≈g′)

mor⁻¹′-⨾ : {A′ B′ C′ : Obj₂} {f′ : Mor₂ A′ B′} {g′ : Mor₂ B′ C′}
        → mor⁻¹′ (f′ ⨾₂ g′) ≈₁ mor⁻¹′ f′ ⨾₁ mor⁻¹′ g′
mor⁻¹′-⨾ = Src.≈-trans₂ (≡-cong mor⁻¹ Mor₂-expand-⨾) mor⁻¹-⨾

FFInverse : SGFunctor Trg Src
FFInverse = record
  {obj = obj⁻¹
  ;mor = mor⁻¹′
  ;isSGFunctor = record
    {mor-cong = mor⁻¹′-cong
    ;mor-⨾    = mor⁻¹′-⨾
    }
  }
```

6.10 Categoric.SGFunctor.Inverse0

```
module Categoric.SGFunctor.Inverse0 {i j₁ k₁ j₂ k₂ : Level} {Obj : Set i}
                                    (SG₁ : Semigroupoid j₁ k₁ Obj)
                                    (SG₂ : Semigroupoid j₂ k₂ Obj) where
private
  module SG₁ = Semigroupoid SG₁
  module SG₂ = Semigroupoid SG₂
open SGFunctorSetup SG₁ SG₂
```

Since type-checking applications of Categoric.SGFunctor.Inverse has had excessive ressources demands, we introduce a variant that considers two semigroupoids with the same objects.

We start from the material available in a full and faithful functor with identical object mapping:

```
module SGF-FF₀-Inverse
  (mor : {A B : Obj} → Mor₁ A B → Mor₂ A B)
  (mor-cong : {A B : Obj} → {f g : Mor₁ A B} → f ≈₁ g → mor f ≈₂ mor g)
  (mor-⨾ : {A B C : Obj} {f : Mor₁ A B} {g : Mor₁ B C}
        → mor (f ⨾₁ g) ≈₂ mor f ⨾₂ mor g)
  (mor⁻¹ : {A B : Obj} → Mor₂ A B → Mor₁ A B)
  (mor⁻¹-cong : {A B : Obj} → {f′ g′ : Mor₂ A B} → f′ ≈₂ g′ → mor⁻¹ f′ ≈₁ mor⁻¹ g′)
  (mor⁻¹-mor : {A B : Obj} {f : Mor₁ A B} → mor⁻¹ (mor f) ≈₁ f)
  (mor-mor⁻¹ : {A B : Obj} → {f′ : Mor₂ A B} → mor (mor⁻¹ f′) ≈₂ f′)
  where
```

With this, it is easy to show that mor⁻¹ preserves composition:

```
mor⁻¹-⨾ : {A B C : Obj} {f′ : Mor₂ A B} {g′ : Mor₂ B C}
  →     mor⁻¹ (f′ ⨾₂ g′) ≈₁ mor⁻¹ f′ ⨾₁ mor⁻¹ g′
mor⁻¹-⨾ {_} {_} {_} {f′} {g′} = ≈₁-begin
    mor⁻¹ (f′ ⨾₂ g′)
  ≈₁˘⟨ mor⁻¹-cong (SG₂.⨾-cong mor-mor⁻¹ mor-mor⁻¹) ⟩
    mor⁻¹ (mor (mor⁻¹ f′) ⨾₂ mor (mor⁻¹ g′))
  ≈₁˘⟨ mor⁻¹-cong mor-⨾ ⟩
    mor⁻¹ (mor (mor⁻¹ f′ ⨾₁ mor⁻¹ g′))
  ≈₁⟨ mor⁻¹-mor ⟩
    mor⁻¹ f′ ⨾₁ mor⁻¹ g′
  □₁


reflectCoEqualiser : {A B : Obj} {f g : Mor₁ A B}
                   → CoEqualiser SG₂ (mor f) (mor g)
                   → CoEqualiser SG₁ f g
reflectCoEqualiser {A} {B} {f₁} {g₁} CoEq = record
  {obj = Q
  ; mor = p₁
  ; prop = ≈₁-begin
      f₁ ⨾₁ p₁
    ≈₁˘⟨ SG₁.⨾-cong₁ mor⁻¹-mor ⟩
      mor⁻¹ (mor f₁) ⨾₁ mor⁻¹ p₂
    ≈₁˘⟨ mor⁻¹-⨾ ⟩
      mor⁻¹ (mor f₁ ⨾₂ p₂)
    ≈₁⟨ mor⁻¹-cong f₂⨾p₂≈g₂⨾p₂ ⟩
      mor⁻¹ (mor g₁ ⨾₂ p₂)
    ≈₁⟨ mor⁻¹-⨾ ⟩
      mor⁻¹ (mor g₁) ⨾₁ mor⁻¹ p₂
    ≈₁⟨ SG₁.⨾-cong₁ mor⁻¹-mor ⟩
      g₁ ⨾₁ p₁
    □₁
  ; universal = univ
  }
  where
    open CoEqualiser SG₂ CoEq renaming (obj to Q; mor to p₂; prop to f₂⨾p₂≈g₂⨾p₂)
    p₁ : Mor₁ B Q
    p₁ = mor⁻¹ p₂
    univ : {Z : Obj} {r₁ : Mor₁ B Z} (f₁⨾r₁≈g₁⨾r₁ : f₁ ⨾₁ r₁ ≈₁ g₁ ⨾₁ r₁)
        → ∃! _≈₁_ (λ u₁ → r₁ ≈₁ p₁ ⨾₁ u₁)
    univ {Z} {r₁} f₁⨾r₁≈g₁⨾r₁ with universal {Z} {mor r₁}
      (≈₂-begin
          mor f₁ ⨾₂ mor r₁
```

```
          ≈₂˘⟨ mor-⨾ ⟩
            mor (f₁ ⨾₁ r₁)
          ≈₂⟨ mor-cong f₁⨾r₁≈g₁⨾r₁ ⟩
            mor (g₁ ⨾₁ r₁)
          ≈₂⟨ mor-⨾ ⟩
            mor g₁ ⨾₂ mor r₁
          □₂)
    ... | u₂, r₂≈p₂⨾u₂, u₂-unique = u₁, r₁≈p₁⨾u₁, u₁-unique
      where
        u₁ : Mor₁ Q Z
        u₁ = mor⁻¹ u₂
        r₂ : Mor₂ B Z
        r₂ = mor r₁
        r₁≈p₁⨾u₁ : r₁ ≈₁ p₁ ⨾₁ u₁
        r₁≈p₁⨾u₁ = ≈₁-begin
              r₁
          ≈₁˘⟨ mor⁻¹-mor ⟩
              mor⁻¹ (mor r₁)
          ≈₁⟨ ≈₁-refl ⟩
              mor⁻¹ r₂
          ≈₁⟨ mor⁻¹-cong r₂≈p₂⨾u₂ ⟩
              mor⁻¹ (p₂ ⨾₂ u₂)
          ≈₁⟨ mor⁻¹-⨾ ⟩
              mor⁻¹ p₂ ⨾₁ mor⁻¹ u₂
          ≈₁⟨ ≈₁-refl ⟩
              p₁ ⨾₁ u₁
          □₁
        u₁-unique : {v₁ : Mor₁ Q Z} (r₁≈p₁⨾v₁ : r₁ ≈₁ p₁ ⨾₁ v₁) → u₁ ≈₁ v₁
        u₁-unique {v₁} r₁≈p₁⨾v₁ = ≈₁-begin
              u₁
          ≈₁⟨ ≈₁-refl ⟩
              mor⁻¹ u₂
          ≈₁⟨ mor⁻¹-cong (u₂-unique {mor v₁}
            (≈₂-begin
                mor r₁
              ≈₂⟨ mor-cong r₁≈p₁⨾v₁ ⟩
                mor (p₁ ⨾₁ v₁)
              ≈₂⟨ mor-⨾ ⟩
                mor p₁ ⨾₂ mor v₁
              ≈₂⟨ SG₂.⨾-cong₁ mor-mor⁻¹ ⟩
                p₂ ⨾₂ mor v₁
              □₂))
            ⟩
              mor⁻¹ (mor v₁)
          ≈₁⟨ mor⁻¹-mor ⟩
              v₁
          □₁
```

6.11 Categoric.SGFunctor

```
open import Categoric.SGFunctor.Core public
open import Categoric.SGFunctor.Equality public
open import Categoric.SGFunctor.Composition public
```

All derived material is collected in the sub-module IsSGFunctorProps for re-use in particular in Categoric.Functor.

```
module _
    {i1 j1 k1 : Level} {Obj1 : Set i1} {Src : Semigroupoid j1 k1 Obj1}
    {i2 j2 k2 : Level} {Obj2 : Set i2} {Trg : Semigroupoid j2 k2 Obj2}
  where
  private
    module Src = Semigroupoid Src
    module Trg = Semigroupoid Trg
  open SGFunctorSetup Src Trg
  module _
    {obj : Obj1 → Obj2}
    {mor : {A B : Obj1} → Mor1 A B → Mor2 (obj A) (obj B)}
    (IsSGF : IsSGFunctor Src Trg obj mor)
    where
    module IsSGFunctorProps where
      open IsSGFunctor IsSGF
      open IsSGFunctorBasicProps IsSGF public
      PreservesInitialObj :   Set (i1 ⊍ i2 ⊍ j1 ⊍ j2 ⊍ k1 ⊍ k2)
      PreservesInitialObj = {I : Obj1} → IsInitial Src I → IsInitial Trg (obj I)
      PreservesTerminalObj :   Set (i1 ⊍ i2 ⊍ j1 ⊍ j2 ⊍ k1 ⊍ k2)
      PreservesTerminalObj = {T : Obj1} → IsTerminal Src T → IsTerminal Trg (obj T)


      KeepInitialObject : HasInitialObject Src → Set (i2 ⊍ j2 ⊍ k2)
      KeepInitialObject srcInit = IsInitial Trg (obj (HasInitialObject.⦶ Src srcInit))
      keptInitialObject : (srcInit : HasInitialObject Src)
                        → KeepInitialObject srcInit → HasInitialObject Trg
      keptInitialObject srcInit isInit2 = record
        {⦶ = obj (HasInitialObject.⦶ Src srcInit)
        ; isInitial = isInit2
        }


      module PreservedInitialObj (presInit : PreservesInitialObj) (hasInit : HasInitialObject Src) where
        open HasInitialObject Src hasInit
        hasInit2 : HasInitialObject Trg
        hasInit2 = record {⦶ = obj ⦶; isInitial = presInit isInitial}
        open HasInitialObject2 Trg hasInit2 public


      open SGF-Pushout Src Trg obj mor mor-cong mor-⨾ public


module SGF
    {i1 j1 k1 : Level} {Obj1 : Set i1} {Src : Semigroupoid j1 k1 Obj1}
    {i2 j2 k2 : Level} {Obj2 : Set i2} {Trg : Semigroupoid j2 k2 Obj2}
    (F : SGFunctor Src Trg)
  where
    open IsSGFunctorProps (SGFunctor.isSGFunctor F) public
    open import Categoric.SGFunctor.Coproduct F public
```

6.11.1 Functor Composition

```
module _ {i j k : Level} {Obj : Set i} (C : Semigroupoid j k Obj) where
  open Semigroupoid C
  Identity-preservesInitialObj : SGF.PreservesInitialObj (Identity C)
  Identity-preservesInitialObj isInit = isInit
  Identity-preservesTerminalObj : SGF.PreservesTerminalObj (Identity C)
```

```
  Identity-preservesTerminalObj isTerm = isTerm
  Identity-preservesCoproduct : SGF.PreservesCoproduct (Identity C)
  Identity-preservesCoproduct isCoprod = isCoprod
  Identity-preservesPushout : SGF.PreservesPushout (Identity C)
  Identity-preservesPushout isPO = isPO
```

6.12 Categoric.SGFunctor.NatTrans

In this module, we deal only with horizontal composition of natural transformations, and therefore can turn the source and target semigroupoids into module parameters:

```
module Categoric.SGFunctor.NatTrans
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} {Src : Semigroupoid j₁ k₁ Obj₁}
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} {Trg : Semigroupoid j₂ k₂ Obj₂}
  where
private
  module Src = Semigroupoid Src
  module Trg = Semigroupoid Trg
open SGFunctorSetup Src Trg
open SGFunctor {Src = Src} {Trg = Trg}
open SGF {Src = Src} {Trg = Trg} using (_(o)_; _(m)_)


record SGNatTrans (F G : SGFunctor Src Trg) : Set (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂ ⊍ k₁ ⊍ k₂) where
  field
    indmor     : {A : Obj₁} → Mor₂ (F (o) A) (G (o) A)
    naturality : {A B : Obj₁} → {f : Mor₁ A B}
               → F (m) f ⨾₂ indmor {B} ≈₂ indmor {A} ⨾₂ G (m) f
open SGNatTrans public


module _ {F G : SGFunctor Src Trg} where
  infixr 4 _≋_
  _≋_ : SGNatTrans F G → SGNatTrans F G → Set (i₁ ⊍ k₂)
  α ≋ β = {A : Obj₁} → indmor α {A} ≈₂ indmor β {A}
  setoid : Setoid (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂ ⊍ k₁ ⊍ k₂) (i₁ ⊍ k₂)
  setoid = record
    {Carrier = SGNatTrans F G
    ; _≈_ = _≋_
    ; isEquivalence = record
      {refl = Trg.≈-refl
      ; sym = λ α≋β → Trg.≈-sym α≋β
      ; trans = λ α≋β β≋γ → Trg.≈-trans α≋β β≋γ
      }
    }
  ≋-refl  = Setoid.refl setoid
  ≋-sym   = Setoid.sym setoid
  ≋-trans = Setoid.trans setoid
Hom = λ F G → setoid {F} {G}


infixr 10 _;_
_;_ : {F G H : SGFunctor Src Trg} → SGNatTrans F G → SGNatTrans G H → SGNatTrans F H
_;_ {F} {G} {H} α β = record
  {indmor = indmor α ⨾₂ indmor β
  ; naturality = λ {A} {B} {f} → ≈₂-begin
```

```
      F (m) f ⨾₂ indmor α ⨾₂ indmor β
    ≈₂⟨ Trg.on-⨾-assocL (Trg.⨾-cong₁ (naturality α)) ⟩
      indmor α ⨾₂ G (m) f ⨾₂ indmor β
    ≈₂⟨ Trg.⨾-cong₂ (naturality β) ⟨≈₂≈⟩ Trg.⨾-assocL ⟩
      (indmor α ⨾₂ indmor β) ⨾₂ H (m) f
    □₂
  }
```

```
⍮-cong : {F G H : SGFunctor Src Trg} {α₁ α₂ : SGNatTrans F G} {β₁ β₂ : SGNatTrans G H}
       → α₁ ≋ α₂ → β₁ ≋ β₂ → α₁ ⍮ β₁ ≋ α₂ ⍮ β₂
⍮-cong α₁≋α₂ β₁≋β₂ = Trg.⨾-cong α₁≋α₂ β₁≋β₂
```

```
⍮-assoc : {F G H K : SGFunctor Src Trg}
           {α : SGNatTrans F G} {β : SGNatTrans G H} {γ : SGNatTrans H K}
        → (α ⍮ β) ⍮ γ ≋ α ⍮ (β ⍮ γ)
⍮-assoc {α = α} {β} {γ} = Trg.⨾-assoc
```

```
compOp : CompOp Hom
compOp = record
  { _⨾_ = _⍮_
  ; ⨾-cong = λ {F G H α₁ α₂ β₁ β₂} α₁≋α₂ β₁≋β₂ {A}
              → ⍮-cong {α₁ = α₁} {α₂} {β₁} {β₂} α₁≋α₂ β₁≋β₂ {A}
  ; ⨾-assoc = λ {F G H K α β γ} → ⍮-assoc {α = α} {β} {γ}
  }
```

```
SGNatTransSemigroupoid : Semigroupoid (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂ ⊍ k₁ ⊍ k₂) (i₁ ⊍ k₂) (SGFunctor Src Trg)
SGNatTransSemigroupoid = record {Hom = Hom; compOp = compOp}
```

6.13 Categoric.Functor

```
CatHomEq : {i j k : Level} {Obj : Set i} (C : Category j k Obj)
         → {A B : Obj} (F G : Category.Mor C A B) → Set k
CatHomEq C F G = Category._≈_ C F G
syntax CatHomEq C F G = F ≈⌊ C ⌋ G
```

```
module FunctorSetup {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (Src : Category j₁ k₁ Obj₁)
                    {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (Trg : Category j₂ k₂ Obj₂) where
  open Category₁ Src public
  open Category₂ Trg public
```

```
record Functor {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (Src : Category j₁ k₁ Obj₁)
               {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (Trg : Category j₂ k₂ Obj₂)
               : Set (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂ ⊍ k₁ ⊍ k₂) where
  private
    module Src = Category Src
    module Trg = Category Trg
  open FunctorSetup Src Trg
  field
    obj : Obj₁ → Obj₂
    mor : {A B : Obj₁} → Mor₁ A B → Mor₂ (obj A) (obj B)
    mor-cong : {A B : Obj₁} → {F G : Mor₁ A B} → F ≈₁ G → mor F ≈₂ mor G
```

```
    mor-⨾  : {A B C : Obj₁} → {F : Mor₁ A B} → {G : Mor₁ B C}
           → mor (F ⨾₁ G) ≈₂ mor F ⨾₂ mor G
    mor-Id : {A : Obj₁} → mor (Id₁ {A}) ≈₂ Id₂ {obj A}
  mor-⨾⨾ : {A B C D : Obj₁} {F : Mor₁ A B} {G : Mor₁ B C} {H : Mor₁ C D}
           → mor (F ⨾₁ G ⨾₁ H) ≈₂ mor F ⨾₂ mor G ⨾₂ mor H
  mor-⨾⨾ = mor-⨾ ⟨≈₂≈⟩ Trg.⨾-cong₂ mor-⨾
```

For the following definitions, we want the Src and Trg categories to be implicit arguments:

```
module _ {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} {Src : Category j₁ k₁ Obj₁}
         {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} {Trg : Category j₂ k₂ Obj₂}
  where
  open FunctorSetup Src Trg
  private
    module Src = Category Src
    module Trg = Category Trg


fromSGFunctor : (F : SGFunctor SG₁ SG₂)
              → ({A : Obj₁} → SGFunctor.mor F (Id₁ {A}) ≈₂ Id₂ {SGFunctor.obj F A})
              → Functor Src Trg
fromSGFunctor F mor-Id = record
  {obj      = obj
  ; mor      = mor
  ; mor-cong = mor-cong
  ; mor-⨾    = mor-⨾
  ; mor-Id   = mor-Id
  }
  where open SGFunctor F


ConstFunctor : Obj₂ → Functor Src Trg
ConstFunctor A = record {obj = λ _ → A; mor = λ _ → Id₂
                        ; mor-cong = λ _ → ≈₂-refl; mor-⨾ = ≈₂-sym leftId₂; mor-Id = ≈₂-refl}


module CatF (F : Functor Src Trg) where
  open Functor F
  sgFunctor : SGFunctor SG₁ SG₂
  sgFunctor = record
    {obj = obj
    ; mor = mor
    ; isSGFunctor = record {mor-cong = mor-cong; mor-⨾ = mor-⨾}
    }
  sgPreservesIdentity : SGF.PreservesIdentity sgFunctor
  sgPreservesIdentity {A} {I} I-isId = Trg.≈Id-isIdentity morI≈Id
    where
      I≈Id : I ≈₁ Id₁ {A}
      I≈Id = Src.isIdentity-≈Id I-isId
      morI≈Id : mor I ≈₂ Id₂
      morI≈Id = mor-cong I≈Id ⟨≈₂≈⟩ mor-Id
  presIsIso : {A B : Obj₁} {f : Mor₁ A B} → Src.IsIso f → Trg.IsIso (mor f)
  presIsIso {f = f} f-isIso = record
    {_⁻¹ = mor _⁻¹
    ; rightInverse = mor-⨾ ⟨≈₂˘≈⟩ mor-cong rightInverse ⟨≈₂≈⟩ mor-Id
    ; leftInverse  = mor-⨾ ⟨≈₂˘≈⟩ mor-cong leftInverse  ⟨≈₂≈⟩ mor-Id
    }
    where open Src.IsIso f-isIso
  presIso : {A B : Obj₁} → Src.Iso A B → Trg.Iso (obj A) (obj B)
```

```
  presIso f = record {isoMor = mor f.isoMor; isIso = presIsIso f.isIso}
    where module f = Src.Iso f
  open SGF sgFunctor public
```

6.13.1 Functor Composition

```
infixr 9 _⨾⨾_
_⨾⨾_ : {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} {C₁ : Category j₁ k₁ Obj₁}
       {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} {C₂ : Category j₂ k₂ Obj₂}
       {i₃ j₃ k₃ : Level} {Obj₃ : Set i₃} {C₃ : Category j₃ k₃ Obj₃}
       (F : Functor C₁ C₂) → (G : Functor C₂ C₃) → Functor C₁ C₃
_⨾⨾_ {C₃ = C₃} F G = record
  {obj = λ x → Functor.obj G (Functor.obj F x)
  ; mor = λ x → Functor.mor G (Functor.mor F x)
  ; mor-cong = λ x → Functor.mor-cong G (Functor.mor-cong F x)
  ; mor-⨾ = Category.≈-trans C₃ (Functor.mor-cong G (Functor.mor-⨾ F)) (Functor.mor-⨾ G)
  ; mor-Id = Category.≈-trans C₃ (Functor.mor-cong G (Functor.mor-Id F)) (Functor.mor-Id G)
  }

FFInverse : {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} {C₁ : Category j₁ k₁ Obj₁}
          → {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} {C₂ : Category j₂ k₂ Obj₂}
          → (F : Functor C₁ C₂) → CatF.IsFullAndFaithful F
          → (obj⁻¹ : Obj₂ → Obj₁)
          → (obj-obj⁻¹ : {A′ : Obj₂} → Functor.obj F (obj⁻¹ A′) ≡ A′)
          → Functor C₂ C₁
FFInverse {C₁ = C₁} {C₂ = C₂} F isFF obj⁻¹ obj-obj⁻¹ = fromSGFunctor
  sgFunctor⁻¹
  (λ {A′} → ≈₁-begin
      mor⁻¹′ Id₂
    ≈₁≡⟨ ≡-cong mor⁻¹ (Category.Id-≡-subst-Src≡Trg C₂ Obj₂-expand Obj₂-expand) ⟩
      mor⁻¹ Id₂
    ≈₁˘⟨ mor⁻¹-cong F.mor-Id ⟩
      mor⁻¹ (F.mor Id₁)
    ≈₁⟨ mor⁻¹-mor ⟩
      Id₁
    □₁)
  where
    open FunctorSetup C₁ C₂
    module F = Functor F
    open CatF.FullAndFaithful F isFF using (mor⁻¹; mor-mor⁻¹; mor⁻¹-mor; mor⁻¹-cong)
    open SGF-FF-Inverse (CatF.sgFunctor F) obj⁻¹ obj-obj⁻¹
                        mor⁻¹ mor⁻¹-cong mor⁻¹-mor mor-mor⁻¹
      using (mor⁻¹′; Obj₂-expand) renaming (FFInverse to sgFunctor⁻¹)

Identity : {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (C : Category j₁ k₁ Obj₁) → Functor C C
Identity C = record
  {obj = λ x → x
  ; mor = λ x → x
  ; mor-cong = λ x → x
  ; mor-⨾ = Category.≈-refl C
  ; mor-Id = Category.≈-refl C
  }

retractFunctor : {i₁ i₂ j₁ k₁ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
               → (F : Obj₂ → Obj₁) (C : Category j₁ k₁ Obj₁)
```

```
                  → Functor (retractCategory F C) C
retractFunctor F C = record
  {obj = F
  ; mor = λ x → x
  ; mor-cong = λ x → x
  ; mor-⨾ = Category.≈-refl C
  ; mor-Id = Category.≈-refl C
  }
```

6.13.2 Natural Transformations

```
module _ {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} {C₁ : Category j₁ k₁ Obj₁}
         {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} {C₂ : Category j₂ k₂ Obj₂} where
  private
    module C₁ = Category C₁
    module C₂ = Category C₂
  open FunctorSetup C₁ C₂

  record NatTrans (F G : Functor C₁ C₂) : Set (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂ ⊍ k₁ ⊍ k₂) where
    private
      module F = Functor F
      module G = Functor G
    field
      indmor : {A : Obj₁} → Mor₂ (F.obj A) (G.obj A)
      naturality : {A B : Obj₁} {f : Mor₁ A B} → F.mor f ⨾₂ indmor {B} ≈₂ indmor {A} ⨾₂ G.mor f
    ≡-naturality : {A B : Obj₁} (A≡B : A ≡ B)
                 → C₂.≡-substSrc (≡-cong F.obj A≡B) (indmor {A})
                 ≡ C₂.≡˘-substTrg (≡-cong G.obj A≡B) (indmor {B})
    ≡-naturality ≡-refl = ≡-refl
    ≡˘-naturality : {A B : Obj₁} (A≡B : A ≡ B)
                  → C₂.≡˘-substSrc (≡-cong F.obj A≡B) (indmor {B})
                  ≡ C₂.≡-substTrg (≡-cong G.obj A≡B) (indmor {A})
    ≡˘-naturality ≡-refl = ≡-refl
  module _ {F G : Functor C₁ C₂} where
    fromSGNatTrans : SGNatTrans (CatF.sgFunctor F) (CatF.sgFunctor G) → NatTrans F G
    fromSGNatTrans T = let open SGNatTrans T in record {indmor = indmor; naturality = naturality}
    open NatTrans
    sgNatTrans : NatTrans F G → SGNatTrans (CatF.sgFunctor F) (CatF.sgFunctor G)
    sgNatTrans T = record {indmor = indmor T; naturality = naturality T}
    infixr 4 _≋_
    _≋_ : NatTrans F G → NatTrans F G → Set (i₁ ⊍ k₂)
    α ≋ β = {A : Obj₁} → indmor α {A} ≈₂ indmor β {A}
    NatTransSetoid : Setoid (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂ ⊍ k₁ ⊍ k₂) (i₁ ⊍ k₂)
    NatTransSetoid = record
      {Carrier = NatTrans F G
      ; _≈_ = _≋_
      ; isEquivalence = record
        {refl = ≈₂-refl
        ; sym = λ α≋β → ≈₂-sym α≋β
        ; trans = λ α≋β β≋γ → ≈₂-trans α≋β β≋γ
        }
      }
      -- Avoiding open ... public using ...
      -- open Setoid NatTransSetoid public using () renaming (refl to ≋-refl; sym to ≋-sym; trans to ≋-trans)
    ≋-refl       = Setoid.refl       NatTransSetoid
```

```
    ≋-reflexive  = Setoid.reflexive NatTransSetoid
    ≋-sym        = Setoid.sym       NatTransSetoid
    ≋-trans      = Setoid.trans     NatTransSetoid
  infixr 10 _;_
  _;_ : {F G H : Functor C₁ C₂} → NatTrans F G → NatTrans G H → NatTrans F H
  _;_ α β = fromSGNatTrans (SGN._;_ (sgNatTrans α) (sgNatTrans β))

  ;-cong     : {F G H : Functor C₁ C₂} {α₁ α₂ : NatTrans F G} {β₁ β₂ : NatTrans G H}
             → α₁ ≋ α₂ → β₁ ≋ β₂ → α₁ ; β₁ ≋ α₂ ; β₂
  ;-cong α₁≋α₂ β₁≋β₂ = C₂.⨾-cong α₁≋α₂ β₁≋β₂

  ;-assoc : {F G H K : Functor C₁ C₂} {α : NatTrans F G} {β : NatTrans G H} {γ : NatTrans H K}
          → (α ; β) ; γ ≋ α ; (β ; γ)
  ;-assoc {α = α} {β} {γ} = C₂.⨾-assoc

  NatTrans≈ : LocalSetoid (Functor C₁ C₂) (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂ ⊍ k₁ ⊍ k₂) (i₁ ⊍ k₂)
  NatTrans≈ F G = NatTransSetoid {F} {G}

  NatTransCompOp : CompOp NatTrans≈
  NatTransCompOp = record
    {_⨾_     = _;_
    ; ⨾-cong  = λ {F G H α₁ α₂ β₁ β₂} α₁≋α₂ β₁≋β₂ {A}
              → ;-cong {α₁ = α₁} {α₂} {β₁} {β₂} α₁≋α₂ β₁≋β₂ {A}
    ; ⨾-assoc = λ {F G H K α β γ} → ;-assoc {α = α} {β} {γ}
    }

  NatTransSemigroupoid : Semigroupoid (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂ ⊍ k₁ ⊍ k₂) (i₁ ⊍ k₂) (Functor C₁ C₂)
  NatTransSemigroupoid = record {Hom = NatTrans≈; compOp = NatTransCompOp}


  IdTrans : (F : Functor C₁ C₂) → NatTrans F F
  IdTrans F = record
    {indmor = Id₂
    ; naturality = λ {A} {B} {f} → ≈₂-begin
              Functor.mor F f ⨾₂ Id₂
           ≈₂⟨ rightId₂ ⟩
              Functor.mor F f
           ≈₂˘⟨ leftId₂ ⟩
              Id₂ ⨾₂ Functor.mor F f
           □₂
    }

  NatTransIdOp : IdOp NatTrans≈ _;_
  NatTransIdOp = record
    {Id = IdTrans _
    ; leftId  = leftId₂
    ; rightId = rightId₂
    }

  NatTransCategory : Category (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂ ⊍ k₁ ⊍ k₂) (i₁ ⊍ k₂) (Functor C₁ C₂)
  NatTransCategory = record {semigroupoid = NatTransSemigroupoid; idOp = NatTransIdOp}


  record NatIso (F G : Functor C₁ C₂) : Set (i₁ ⊍ i₂ ⊍ j₁ ⊍ j₂ ⊍ k₁ ⊍ k₂) where
    private
      module F = Functor F
      module G = Functor G
    field
      indmor : {A : Obj₁} → Mor₂ (F.obj A) (G.obj A)
      naturality : {A B : Obj₁} → {f : Mor₁ A B}
                 → (F.mor f ⨾₂ indmor {B}) ≈₂ (indmor {A} ⨾₂ G.mor f)
      indmor⁻¹ : {A : Obj₁} → Mor₂ (G.obj A) (F.obj A)
      indmor-⨾-indmor⁻¹ : {A : Obj₁} → indmor {A} ⨾₂ indmor⁻¹ {A} ≈₂ Id₂ {F.obj A}
```

```
      indmor⁻¹-⨾-indmor : {A : Obj₁} → indmor⁻¹ {A} ⨾₂ indmor {A} ≈₂ Id₂ {G.obj A}
    naturality⁻¹  :  {A B : Obj₁} {f : Mor₁ A B}
                  → (G.mor f ⨾₂ indmor⁻¹ {B}) ≈₂ (indmor⁻¹ {A} ⨾₂ F.mor f)
    naturality⁻¹ {A} {B} {f}  =  ≈₂-begin
             G.mor f ⨾₂ indmor⁻¹ {B}
          ≈₂˘⟨ leftId₂ ⟩
             Id₂ ⨾₂ G.mor f ⨾₂ indmor⁻¹ {B}
          ≈₂˘⟨ C₂.⨾-cong₁ indmor⁻¹-⨾-indmor ⟩
             (indmor⁻¹ {A} ⨾₂ indmor {A}) ⨾₂ G.mor f ⨾₂ indmor⁻¹ {B}
          ≈₂⟨ C₂.⨾-cong₁₂&₂₁ (≈₂-sym naturality) ⟩
             indmor⁻¹ {A} ⨾₂ (F.mor f ⨾₂ indmor {B}) ⨾₂ indmor⁻¹ {B}
          ≈₂⟨ C₂.⨾-cong₂ (C₂.⨾-assoc ⟨≈₂≈⟩ C₂.⨾-cong₂ indmor-⨾-indmor⁻¹) ⟩
             indmor⁻¹ {A} ⨾₂ F.mor f ⨾₂ Id₂
          ≈₂⟨ C₂.⨾-cong₂ rightId₂ ⟩
             indmor⁻¹ {A} ⨾₂ F.mor f
          □₂
    to        :  NatTrans F G
    to        =  record {indmor  =  indmor; naturality  =  naturality}
    from      :  NatTrans G F
    from      =  record {indmor  =  indmor⁻¹; naturality  =  naturality⁻¹}
    ≡-naturality     :  {A B : Obj₁} (A≡B : A ≡ B)
                     → C₂.≡-substSrc (≡-cong F.obj A≡B) (indmor {A})
                     ≡  C₂.≡˘-substTrg (≡-cong G.obj A≡B) (indmor {B})
    ≡-naturality ≡-refl  =  ≡-refl
    ≡˘-naturality    :  {A B : Obj₁} (A≡B : A ≡ B)
                     → C₂.≡˘-substSrc (≡-cong F.obj A≡B) (indmor {B})
                     ≡  C₂.≡-substTrg (≡-cong G.obj A≡B) (indmor {A})
    ≡˘-naturality ≡-refl  =  ≡-refl
    ≡-naturality⁻¹   :  {A B : Obj₁} (A≡B : A ≡ B)
                     → C₂.≡-substSrc (≡-cong G.obj A≡B) (indmor⁻¹ {A})
                     ≡  C₂.≡˘-substTrg (≡-cong F.obj A≡B) (indmor⁻¹ {B})
    ≡-naturality⁻¹ ≡-refl  =  ≡-refl
    ≡˘-naturality⁻¹  :  {A B : Obj₁} (A≡B : A ≡ B)
                     → C₂.≡˘-substSrc (≡-cong G.obj A≡B) (indmor⁻¹ {B})
                     ≡  C₂.≡-substTrg (≡-cong F.obj A≡B) (indmor⁻¹ {A})
    ≡˘-naturality⁻¹ ≡-refl  =  ≡-refl

  NatIso˘ : {F G : Functor C₁ C₂} → NatIso F G → NatIso G F
  NatIso˘ T  =  let open NatIso T in record
    {indmor               =  indmor⁻¹
    ; naturality          =  naturality⁻¹
    ; indmor⁻¹            =  indmor
    ; indmor-⨾-indmor⁻¹   =  indmor⁻¹-⨾-indmor
    ; indmor⁻¹-⨾-indmor   =  indmor-⨾-indmor⁻¹
    }

  infixr 10 _≊;̲_  _;̲≊_   _≊;̲≊_
  _≊;̲_    : {F G H : Functor C₁ C₂} → NatIso F G → NatTrans G H → NatTrans F H
  α ≊;̲ β  =  NatIso.to α ;̲ β
  _;̲≊_    : {F G H : Functor C₁ C₂} → NatTrans F G → NatIso G H → NatTrans F H
  α ;̲≊ β  =  α ;̲ NatIso.to β
  _≊;̲≊_  : {F G H : Functor C₁ C₂} → NatIso F G → NatIso G H → NatIso F H
  α ≊;̲≊ β  =  let αβ  =  NatIso.to    α ;̲ NatIso.to    β
                  βα  =  NatIso.from β ;̲ NatIso.from α
```

```
                open NatIso
    in record
    {indmor              = NatTrans.indmor αβ
    ; naturality         = NatTrans.naturality αβ
    ; indmor⁻¹           = NatTrans.indmor βα
    ; indmor-⨾-indmor⁻¹  = ≈₂-begin
          (indmor α ⨾₂ indmor β) ⨾₂ (indmor⁻¹ β ⨾₂ indmor⁻¹ α)
        ≈₂⟨ C₂.⨾-cong₁₂&₂₁ (indmor-⨾-indmor⁻¹ β) ⟩
          indmor α ⨾₂ Id₂ ⨾₂ indmor⁻¹ α
        ≈₂⟨ C₂.⨾-cong₂ leftId₂ ⟨≈₂≈⟩ indmor-⨾-indmor⁻¹ α ⟩
          Id₂
        □₂
    ; indmor⁻¹-⨾-indmor  = ≈₂-begin
          (indmor⁻¹ β ⨾₂ indmor⁻¹ α) ⨾₂ (indmor α ⨾₂ indmor β)
        ≈₂⟨ C₂.⨾-cong₁₂&₂₁ (indmor⁻¹-⨾-indmor α) ⟩
          indmor⁻¹ β ⨾₂ Id₂ ⨾₂ indmor β
        ≈₂⟨ C₂.⨾-cong₂ leftId₂ ⟨≈₂≈⟩ indmor⁻¹-⨾-indmor β ⟩
          Id₂
        □₂
    }

  NatLeftId : {F : Functor C₁ C₂} → NatIso (Identity C₁ ⨾⨾ F) F
  NatLeftId {F} = record
    {indmor              = Id₂
    ; naturality         = rightId₂ ⟨≈₂≈˘⟩ leftId₂
    ; indmor⁻¹           = Id₂
    ; indmor-⨾-indmor⁻¹  = leftId₂
    ; indmor⁻¹-⨾-indmor  = leftId₂
    }

  NatRightId : {F : Functor C₁ C₂} → NatIso (F ⨾⨾ Identity C₂) F
  NatRightId {F} = record
    {indmor              = Id₂
    ; naturality         = rightId₂ ⟨≈₂≈˘⟩ leftId₂
    ; indmor⁻¹           = Id₂
    ; indmor-⨾-indmor⁻¹  = rightId₂
    ; indmor⁻¹-⨾-indmor  = rightId₂
    }

module _ {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} {C₁ : Category j₁ k₁ Obj₁}
         {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} {C₂ : Category j₂ k₂ Obj₂}
         {i₃ j₃ k₃ : Level} {Obj₃ : Set i₃} {C₃ : Category j₃ k₃ Obj₃} where
    private
      module C₁ = Category C₁
      module C₂ = Category C₂
      module C₃ = Category C₃
    open FunctorSetup C₁ C₂
    open Category₃ C₃
    open NatTrans
    _◃⨾⨾_ : {F G : Functor C₁ C₂} → NatTrans F G → (H : Functor C₂ C₃) → NatTrans (F ⨾⨾ H) (G ⨾⨾ H)
    _◃⨾⨾_ {F} {G} α H = record
      {indmor = H.mor (indmor α)
      ; naturality = λ {A} {B} {f} → ≈₃-begin
            H.mor (F.mor f) ⨾₃ H.mor (indmor α)
          ≈₃˘⟨ H.mor-⨾ ⟩
```

```
        H.mor (F.mor f ⨾₂ indmor α)
      ≈₃⟨ H.mor-cong (naturality α) ⟩
        H.mor (indmor α ⨾₂ G.mor f)
      ≈₃⟨ H.mor-⨾ ⟩
        H.mor (indmor α) ⨾₃ H.mor (G.mor f)
      □₃
    }
    where
      module F = Functor F
      module G = Functor G
      module H = Functor H

  _⨾⨾▹_ : (F : Functor C₁ C₂) {G H : Functor C₂ C₃} → NatTrans G H → NatTrans (F ⨾⨾ G) (F ⨾⨾ H)
  _⨾⨾▹_ F {G} {H} α = record
    {indmor = λ {A} → indmor α {F.obj A}
    ; naturality = naturality α
    }
    where module F = Functor F

  NatAssoc : {i₄ j₄ k₄ : Level} {Obj₄ : Set i₄} {C₄ : Category j₄ k₄ Obj₄}
           → {F : Functor C₁ C₂} {G : Functor C₂ C₃} {H : Functor C₃ C₄}
           → NatIso ((F ⨾⨾ G) ⨾⨾ H) (F ⨾⨾ (G ⨾⨾ H))
  NatAssoc {C₄ = C₄} = record
    {indmor = Id₄
    ; naturality = rightId₄ ⟨≈₄≈˘⟩ leftId₄
    ; indmor⁻¹ = Id₄
    ; indmor-⨾-indmor⁻¹ = leftId₄
    ; indmor⁻¹-⨾-indmor = leftId₄
    }
    where
      open Category₄ C₄
```

6.13.3 Bifunctors

```
record Bifunctor {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (Src₁ : Category j₁ k₁ Obj₁)
                 {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (Src₂ : Category j₂ k₂ Obj₂)
                 {i₃ j₃ k₃ : Level} {Obj₃ : Set i₃} (Trg : Category j₃ k₃ Obj₃)
                 : Set (i₁ ⊍ i₂ ⊍ i₃ ⊍ j₁ ⊍ j₂ ⊍ j₃ ⊍ k₁ ⊍ k₂ ⊍ k₃) where
  open Category
  field
    obj : Obj₁ → Obj₂ → Obj₃
    mor : {A₁ B₁ : Obj₁} → Mor Src₁ A₁ B₁
        → {A₂ B₂ : Obj₂} → Mor Src₂ A₂ B₂
        → Mor Trg (obj A₁ A₂) (obj B₁ B₂)
    mor-cong : {A₁ B₁ : Obj₁} → {F₁ F₂ : Mor Src₁ A₁ B₁} → F₁ ≈⌊ Src₁ ⌋ F₂
             → {A₂ B₂ : Obj₂} → {G₁ G₂ : Mor Src₂ A₂ B₂} → G₁ ≈⌊ Src₂ ⌋ G₂
                                → mor F₁ G₁ ≈⌊ Trg ⌋ mor F₂ G₂
    mor-⨾ : {A₁ B₁ C₁ : Obj₁} {F₁ : Mor Src₁ A₁ B₁} → {G₁ : Mor Src₁ B₁ C₁}
          → {A₂ B₂ C₂ : Obj₂} {F₂ : Mor Src₂ A₂ B₂} → {G₂ : Mor Src₂ B₂ C₂}
          → mor (_⨾_ Src₁ F₁ G₁) (_⨾_ Src₂ F₂ G₂) ≈⌊ Trg ⌋ _⨾_ Trg (mor F₁ F₂) (mor G₁ G₂)
    mor-Id : {A₁ : Obj₁} {A₂ : Obj₂}
           → mor (Id Src₁ {A₁}) (Id Src₂ {A₂}) ≈⌊ Trg ⌋ Id Trg {obj A₁ A₂}

  mor-cong₁ : {A₁ B₁ : Obj₁} {F G : Mor Src₁ A₁ B₁}
            → {A₂ B₂ : Obj₂} {H : Mor Src₂ A₂ B₂}
            → F ≈⌊ Src₁ ⌋ G → mor F H ≈⌊ Trg ⌋ mor G H
  mor-cong₁ F≈G = mor-cong F≈G (≈-refl Src₂)

  mor-cong₂ : {A₁ B₁ : Obj₁} {H : Mor Src₁ A₁ B₁}
            → {A₂ B₂ : Obj₂} {F G : Mor Src₂ A₂ B₂}
```

```
             → F ≈⌊ Src₂ ⌋ G → mor H F ≈⌊ Trg ⌋ mor H G
mor-cong₂ F≈G = mor-cong (≈-refl Src₁) F≈G
mor-₁⨾ :  {A₁ B₁ : Obj₁} {F₁ : Mor Src₁ A₁ B₁}
       →  {A₂ B₂ C₂ : Obj₂} {F₂ : Mor Src₂ A₂ B₂} {G₂ : Mor Src₂ B₂ C₂}
       →  mor F₁ (_⨾_ Src₂ F₂ G₂) ≈⌊ Trg ⌋ _⨾_ Trg (mor F₁ F₂) (mor (Id Src₁) G₂)
mor-₁⨾ = _⟨≈˘≈⟩_ Trg (mor-cong₁ (rightId Src₁)) mor-⨾
mor-⨾₁ :  {A₁ B₁ : Obj₁} {F₁ : Mor Src₁ A₁ B₁}
       →  {A₂ B₂ C₂ : Obj₂} {F₂ : Mor Src₂ A₂ B₂} {G₂ : Mor Src₂ B₂ C₂}
       →  mor F₁ (_⨾_ Src₂ F₂ G₂) ≈⌊ Trg ⌋ _⨾_ Trg (mor (Id Src₁) F₂) (mor F₁ G₂)
mor-⨾₁ = _⟨≈˘≈⟩_ Trg (mor-cong₁ (leftId Src₁)) mor-⨾
mor-₂⨾ :  {A₁ B₁ C₁ : Obj₁} {F₁ : Mor Src₁ A₁ B₁} {G₁ : Mor Src₁ B₁ C₁}
       →  {A₂ B₂ : Obj₂} {F₂ : Mor Src₂ A₂ B₂}
       →  mor (_⨾_ Src₁ F₁ G₁) F₂ ≈⌊ Trg ⌋ _⨾_ Trg (mor F₁ F₂) (mor G₁ (Id Src₂))
mor-₂⨾ = _⟨≈˘≈⟩_ Trg (mor-cong₂ (rightId Src₂)) mor-⨾
mor-⨾₂ :  {A₁ B₁ C₁ : Obj₁} {F₁ : Mor Src₁ A₁ B₁} {G₁ : Mor Src₁ B₁ C₁}
       →  {A₂ B₂ : Obj₂} {F₂ : Mor Src₂ A₂ B₂}
       →  mor (_⨾_ Src₁ F₁ G₁) F₂ ≈⌊ Trg ⌋ _⨾_ Trg (mor F₁ (Id Src₂)) (mor G₁ F₂)
mor-⨾₂ = _⟨≈˘≈⟩_ Trg (mor-cong₂ (leftId Src₂)) mor-⨾
mor-₁⨾₂ :  {A₁ B₁ : Obj₁} {F₁ : Mor Src₁ A₁ B₁}
        →  {A₂ B₂ : Obj₂} {F₂ : Mor Src₂ A₂ B₂}
        →  mor F₁ F₂ ≈⌊ Trg ⌋ _⨾_ Trg (mor F₁ (Id Src₂)) (mor (Id Src₁) F₂)
mor-₁⨾₂ = _⟨≈˘≈⟩_ Trg (mor-cong (rightId Src₁) (leftId Src₂)) mor-⨾
mor-₂⨾₁ :  {A₁ B₁ : Obj₁} {F₁ : Mor Src₁ A₁ B₁}
        →  {A₂ B₂ : Obj₂} {F₂ : Mor Src₂ A₂ B₂}
        →  mor F₁ F₂ ≈⌊ Trg ⌋ _⨾_ Trg (mor (Id Src₁) F₂) (mor F₁ (Id Src₂))
mor-₂⨾₁ = _⟨≈˘≈⟩_ Trg (mor-cong (leftId Src₁) (rightId Src₂)) mor-⨾
functor₁ : {A₂ : Obj₂} → Functor Src₁ Trg
functor₁ {A₂} = record
  {obj = λ A₁ → obj A₁ A₂
  ; mor = λ F → mor F (Id Src₂ {A₂})
  ; mor-cong = λ e → mor-cong e (≈-refl Src₂)
  ; mor-⨾ = λ {A₁} {B₁} {C₁} {F₁} {G₁}
        →  mor-cong (≈-refl Src₁) (≈-sym Src₂ (leftId Src₂))
           ⟨ ≈-trans Trg ⟩
           mor-⨾ {A₁} {B₁} {C₁} {F₁} {G₁}
                 {A₂} {A₂} {A₂} {Id Src₂ {A₂}} {Id Src₂ {A₂}}
  ; mor-Id = λ {A₁} → mor-Id {A₁} {A₂}
  }
functor₂ : {A₁ : Obj₁} → Functor Src₂ Trg
functor₂ {A₁} = record
  {obj = λ A₂ → obj A₁ A₂
  ; mor = λ F → mor (Id Src₁ {A₁}) F
  ; mor-cong = λ e → mor-cong (≈-refl Src₁) e
  ; mor-⨾ = λ {A₂} {B₂} {C₂} {F₂} {G₂}
        → mor-cong (≈-sym Src₁ (leftId Src₁)) (≈-refl Src₂)
          ⟨ ≈-trans Trg ⟩
          mor-⨾ {A₁} {A₁} {A₁} {Id Src₁ {A₁}} {Id Src₁ {A₁}}
                {A₂} {B₂} {C₂} {F₂} {G₂}
  ; mor-Id = λ {A₂} → mor-Id {A₁} {A₂}
  }
_≡-obj-≡_ : {A A′ : Obj₁} {B B′ : Obj₂} → A ≡ A′ → B ≡ B′ → obj A B ≡ obj A′ B′
_≡-obj-≡_ ≡-refl ≡-refl = ≡-refl
  -- Avoiding open ... public using ...:
  -- module FunctorProps₁ {A₂ : Obj₂} where
  -- open Functor (functor₁ {A₂}) public using () renaming
  -- (mor-≡-substSrc to mor₁-≡-substSrc
  -- ; mor-≡-substTrg to mor₁-≡-substTrg
```

```
    -- )
    -- module FunctorProps₂ {A₁ : Obj₁} where
    -- open Functor (functor₂ {A₁}) public using () renaming
    -- (mor-≡-substSrc to mor₂-≡-substSrc
    -- ; mor-≡-substTrg to mor₂-≡-substTrg
    -- )
    -- open FunctorProps₁ public
    -- open FunctorProps₂ public
  mor₁-≡-substSrc = λ {A₂ A B} F {A′} A≡A′
                   → CatF.mor-≡-substSrc (functor₁ {A₂}) {A} {B} F {A′} A≡A′
  mor₁-≡-substTrg = λ {A₂ A B} F {B′} B≡B′
                   → CatF.mor-≡-substTrg (functor₁ {A₂}) {A} {B} F {B′} B≡B′
  mor₂-≡-substSrc = λ {A₁ A B} F {A′} A≡A′
                   → CatF.mor-≡-substSrc (functor₂ {A₁}) {A} {B} F {A′} A≡A′
  mor₂-≡-substTrg = λ {A₁ A B} F {B′} B≡B′
                   → CatF.mor-≡-substTrg (functor₂ {A₁}) {A} {B} F {B′} B≡B′
  mor-≡-substSrc : {A₁ A₁′ B₁ : Obj₁} {A₂ A₂′ B₂ : Obj₂}
                    (F₁ : Mor Src₁ A₁ B₁) (F₂ : Mor Src₂ A₂ B₂) (A₁≡A₁′ : A₁ ≡ A₁′) (A₂≡A₂′ : A₂ ≡ A₂′)
                 → mor (≡-substSrc Src₁ A₁≡A₁′ F₁) (≡-substSrc Src₂ A₂≡A₂′ F₂)
                 ≡ ≡-substSrc Trg (≡-cong₂ obj A₁≡A₁′ A₂≡A₂′) (mor F₁ F₂)
  mor-≡-substSrc F₁ F₂ ≡-refl ≡-refl = ≡-refl
  mor-≡-substTrg : {A₁ B₁ B₁′ : Obj₁} {A₂ B₂ B₂′ : Obj₂}
                    (F₁ : Mor Src₁ A₁ B₁) (F₂ : Mor Src₂ A₂ B₂) (B₁≡B₁′ : B₁ ≡ B₁′) (B₂≡B₂′ : B₂ ≡ B₂′)
                 → mor (≡-substTrg Src₁ B₁≡B₁′ F₁) (≡-substTrg Src₂ B₂≡B₂′ F₂)
                 ≡ ≡-substTrg Trg (≡-cong₂ obj B₁≡B₁′ B₂≡B₂′) (mor F₁ F₂)
  mor-≡-substTrg F₁ F₂ ≡-refl ≡-refl = ≡-refl
  mor-≡-substSrc₁ : {A₁ A₁′ B₁ : Obj₁} {A₂ B₂ : Obj₂}
                    (F₁ : Mor Src₁ A₁ B₁) (F₂ : Mor Src₂ A₂ B₂) (A₁≡A₁′ : A₁ ≡ A₁′)
                 → mor (≡-substSrc Src₁ A₁≡A₁′ F₁) F₂
                 ≡ ≡-substSrc Trg (≡-cong (flip obj _) A₁≡A₁′) (mor F₁ F₂)
  mor-≡-substSrc₁ F₁ F₂ ≡-refl = ≡-refl
  mor-≡-substSrc₂ : {A₁ B₁ : Obj₁} {A₂ A₂′ B₂ : Obj₂}
                    (F₁ : Mor Src₁ A₁ B₁) (F₂ : Mor Src₂ A₂ B₂) (A₂≡A₂′ : A₂ ≡ A₂′)
                 → mor F₁ (≡-substSrc Src₂ A₂≡A₂′ F₂)
                 ≡ ≡-substSrc Trg (≡-cong (obj _) A₂≡A₂′) (mor F₁ F₂)
  mor-≡-substSrc₂ F₁ F₂ ≡-refl = ≡-refl
  mor-≡-substTrg₁ : {A₁ B₁ B₁′ : Obj₁} {A₂ B₂ : Obj₂}
                    (F₁ : Mor Src₁ A₁ B₁) (F₂ : Mor Src₂ A₂ B₂) (B₁≡B₁′ : B₁ ≡ B₁′)
                 → mor (≡-substTrg Src₁ B₁≡B₁′ F₁) F₂
                 ≡ ≡-substTrg Trg (≡-cong (flip obj _) B₁≡B₁′) (mor F₁ F₂)
  mor-≡-substTrg₁ F₁ F₂ ≡-refl = ≡-refl
  mor-≡-substTrg₂ : {A₁ B₁ : Obj₁} {A₂ B₂ B₂′ : Obj₂}
                    (F₁ : Mor Src₁ A₁ B₁) (F₂ : Mor Src₂ A₂ B₂) (B₂≡B₂′ : B₂ ≡ B₂′)
                 → mor F₁ (≡-substTrg Src₂ B₂≡B₂′ F₂)
                 ≡ ≡-substTrg Trg (≡-cong (obj _) B₂≡B₂′) (mor F₁ F₂)
  mor-≡-substTrg₂ F₁ F₂ ≡-refl = ≡-refl


Flip : {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} {Src₁ : Category j₁ k₁ Obj₁}
       {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} {Src₂ : Category j₂ k₂ Obj₂}
       {i₃ j₃ k₃ : Level} {Obj₃ : Set i₃} {Trg : Category j₃ k₃ Obj₃}
     → Bifunctor Src₁ Src₂ Trg
     → Bifunctor Src₂ Src₁ Trg
Flip B = let open Bifunctor B in record
  {obj = flip obj
  ; mor = λ f g → mor g f
  ; mor-cong = λ p q → mor-cong q p
  ; mor-⨾ = mor-⨾
```

```
  ; mor-Id = mor-Id
  }
```

[WK: *Better name?* **]**

```
ConstBifunctor : {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} {Src₁ : Category j₁ k₁ Obj₁}
     {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} {Src₂ : Category j₂ k₂ Obj₂}
     {i₃ j₃ k₃ : Level} {Obj₃ : Set i₃} {Trg : Category j₃ k₃ Obj₃}
  → Functor Src₂ Trg
  → Bifunctor Src₁ Src₂ Trg
ConstBifunctor F = record
  {obj = λ A B → F.obj B
  ; mor = λ f g → F.mor g
  ; mor-cong = λ p q → F.mor-cong q
  ; mor-⨾ = F.mor-⨾
  ; mor-Id = F.mor-Id
  }
  where module F = Functor F
```

[WK: *Better name?* **]**

```
Drop₁ : {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (Src₁ : Category j₁ k₁ Obj₁)
        {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (Src₂ : Category j₂ k₂ Obj₂)
  →     Bifunctor Src₁ Src₂ Src₂
Drop₁ _ Src₂ = record
  {obj = λ A B → B
  ; mor = λ f g → g
  ; mor-cong = λ p q → q
  ; mor-⨾ = Category.≈-refl Src₂
  ; mor-Id = Category.≈-refl Src₂
  }
```

[WK: *Better name?* **]**

```
Drop₂ : {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (Src₁ : Category j₁ k₁ Obj₁)
        {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (Src₂ : Category j₂ k₂ Obj₂)
  →     Bifunctor Src₁ Src₂ Src₁
Drop₂ Src₁ _ = record
  {obj = λ A B → A
  ; mor = λ f g → f
  ; mor-cong = λ p q → p
  ; mor-⨾ = Category.≈-refl Src₁
  ; mor-Id = Category.≈-refl Src₁
  }
```

6.14 Categoric.Functor.BiFunctor

A bifunctor for which the two source categories coincide can be seen as a functor from the product category indexed by Fin 2:

```
BiFunctor : {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} {Src : Category j₁ k₁ Obj₁}
            {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} {Trg : Category j₂ k₂ Obj₂}
          → (⊗ : Bifunctor Src Src Trg)
          → Functor (SIPCategory (Fin 2) Src) Trg
BiFunctor ⊗ = let open OTimes ⊗ using (_⊗o_; _⊗m_; ⊗-cong; ⨾-⊗-⨾; ⊗-Id) in record
```

```
  {obj  =  λ A → A 0 ₂ ⊗o A 1 ₂
  ; mor  =  λ {A} {B} F → F 0 ₂ ⊗m F 1 ₂
  ; mor-cong  =  λ {A} {B} {F} {G} F≈G → ⊗-cong (F≈G 0 ₂) (F≈G 1 ₂)
  ; mor-⨾  =  ⨾-⊗-⨾
  ; mor-Id     =  ⊗-Id
  }
```

Given two appropriately typed bifunctors, we can construct functors from the product category indexed by Fin 3 by nesting them one way or the other:

```
TriFunctorL :  {i₁ j₁ k₁  : Level} {Obj₁  : Set i₁} {C₁  : Category j₁ k₁ Obj₁}
               {i₂ j₂ k₂  : Level} {Obj₂  : Set i₂} {C₂  : Category j₂ k₂ Obj₂}
               {i₃ j₃ k₃  : Level} {Obj₃  : Set i₃} {C₃  : Category j₃ k₃ Obj₃}
            → (⊗ : Bifunctor C₁ C₁ C₂)
            → (B : Bifunctor C₂ C₁ C₃)
            → Functor (SIPCategory (Fin 3) C₁) C₃
TriFunctorL {C₁  =  C₁} {C₂  =  C₂} {C₃  =  C₃} ⊗ B  =  let
    open OTimes ⊗ using (_⊗o_; _⊗m_; ⊗-cong; ⨾-⊗-⨾; ⊗-Id)
    open Bifunctor B using (obj; mor; mor-cong; mor-cong₁; mor-⨾; mor-Id)
    open Category C₃ using (≈-begin_; _≈⟨_⟩_; _□; _⨾_; Id)
    open Category C₁ using () renaming (_⨾_ to _⨾₁_; Id to Id₁)
    open Category C₂ using () renaming (_⨾_ to _⨾₂_; Id to Id₂)
  in record
    {obj  =  λ A → obj (A 0 ₃ ⊗o A 1 ₃) (A 2 ₃)
    ; mor  =  λ {A} {B} F → mor (F 0 ₃ ⊗m F 1 ₃) (F 2 ₃)
    ; mor-cong  =  λ {A} {B} {F} {G} F≈G → mor-cong (⊗-cong (F≈G 0 ₃) (F≈G 1 ₃)) (F≈G 2 ₃)
    ; mor-⨾   =  λ {A} {B} {C} {F} {G} → ≈-begin
        mor ((F 0 ₃ ⨾₁ G 0 ₃) ⊗m (F 1 ₃ ⨾₁ G 1 ₃)) (F 2 ₃ ⨾₁ G 2 ₃)
      ≈⟨ mor-cong₁ ⨾-⊗-⨾ ⟩
        mor ((F 0 ₃ ⊗m F 1 ₃) ⨾₂ (G 0 ₃ ⊗m G 1 ₃)) (F 2 ₃ ⨾₁ G 2 ₃)
      ≈⟨ mor-⨾ ⟩
        mor (F 0 ₃ ⊗m F 1 ₃) (F 2 ₃) ⨾ mor (G 0 ₃ ⊗m G 1 ₃) (G 2 ₃)
      □
    ; mor-Id  =  λ {A} → ≈-begin
        mor (Id₁ ⊗m Id₁) Id₁
      ≈⟨ mor-cong₁ ⊗-Id ⟩
        mor Id₂ Id₁
      ≈⟨ mor-Id ⟩
        Id
      □
    }


TriFunctorR :  {i₁ j₁ k₁  : Level} {Obj₁  : Set i₁} {C₁  : Category j₁ k₁ Obj₁}
               {i₂ j₂ k₂  : Level} {Obj₂  : Set i₂} {C₂  : Category j₂ k₂ Obj₂}
               {i₃ j₃ k₃  : Level} {Obj₃  : Set i₃} {C₃  : Category j₃ k₃ Obj₃}
            → (⊗ : Bifunctor C₁ C₁ C₂)
            → (B : Bifunctor C₁ C₂ C₃)
            → Functor (SIPCategory (Fin 3) C₁) C₃
TriFunctorR {C₁  =  C₁} {C₂  =  C₂} {C₃  =  C₃} ⊗ B  =  let
    open OTimes ⊗ using (_⊗o_; _⊗m_; ⊗-cong; ⨾-⊗-⨾; ⊗-Id)
    open Bifunctor B using (obj; mor; mor-cong; mor-cong₂; mor-⨾; mor-Id)
    open Category C₃ using (≈-begin_; _≈⟨_⟩_; _□; _⨾_; Id)
    open Category C₁ using () renaming (_⨾_ to _⨾₁_; Id to Id₁)
    open Category C₂ using () renaming (_⨾_ to _⨾₂_; Id to Id₂)
  in record
    {obj  =  λ A → obj (A 0 ₃) (A 1 ₃ ⊗o A 2 ₃)
    ; mor  =  λ {A} {B} F → mor (F 0 ₃) (F 1 ₃ ⊗m F 2 ₃)
    ; mor-cong  =  λ {A} {B} {F} {G} F≈G → mor-cong (F≈G 0 ₃) (⊗-cong (F≈G 1 ₃) (F≈G 2 ₃))
    ; mor-⨾   =  λ {A} {B} {C} {F} {G} → ≈-begin
```

```
      mor (F 0 ₃ ⨾₁ G 0 ₃) ((F 1 ₃ ⨾₁ G 1 ₃) ⊗m (F 2 ₃ ⨾₁ G 2 ₃))
    ≈⟨ mor-cong₂ ⨾-⊗-⨾ ⟩
      mor (F 0 ₃ ⨾₁ G 0 ₃) ((F 1 ₃ ⊗m F 2 ₃) ⨾₂ (G 1 ₃ ⊗m G 2 ₃))
    ≈⟨ mor-⨾ ⟩
      mor (F 0 ₃) (F 1 ₃ ⊗m F 2 ₃) ⨾ mor (G 0 ₃) (G 1 ₃ ⊗m G 2 ₃)
    □
  ; mor-Id = λ {A} → ≈-begin
      mor Id₁ (Id₁ ⊗m Id₁)
    ≈⟨ mor-cong₂ ⊗-Id ⟩
      mor Id₁ Id₂
    ≈⟨ mor-Id ⟩
      Id
    □
  }
```

6.15 Categoric.Functor.OTimes

This module implements a naming convention for the components of bifunctors named “⊗”.

```
module OTimes {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} {C₁ : Category j₁ k₁ Obj₁}
              {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} {C₂ : Category j₂ k₂ Obj₂}
              {i₃ j₃ k₃ : Level} {Obj₃ : Set i₃} {C₃ : Category j₃ k₃ Obj₃}
              (⊗ : Bifunctor C₁ C₂ C₃)
  = Bifunctor ⊗ renaming
    (obj to _⊗o_
    ; mor to _⊗m_
    ; mor-⨾ to ⨾-⊗-⨾
    ; mor-₁⨾ to ⨾I-⊗-⨾
    ; mor-⨾₁ to I⨾-⊗-⨾
    ; mor-₂⨾ to ⨾-⊗-⨾I
    ; mor-⨾₂ to ⨾-⊗-I⨾
    ; mor-₁⨾₂ to ⨾I-⊗-I⨾
    ; mor-₂⨾₁ to I⨾-⊗-I⨾
    ; mor-cong to ⊗-cong
    ; mor-cong₁ to ⊗-cong₁
    ; mor-cong₂ to ⊗-cong₂
    ; mor-Id to ⊗-Id
    ; functor₁ to ⊗-functor₁
    ; functor₂ to ⊗-functor₂
    ; _≡-obj-≡_ to _≡⊗≡_
    ; mor₁-≡-substSrc to ⊗₁-≡-substSrc
    ; mor₂-≡-substSrc to ⊗₂-≡-substSrc
    ; mor₁-≡-substTrg to ⊗₁-≡-substTrg
    ; mor₂-≡-substTrg to ⊗₂-≡-substTrg
    )
```

6.16 Categoric.Functor.Coproduct

```
CoproductBifunctor : {i j k : Level} {Obj : Set i}
  → (C : Category j k Obj)
  → CatFinColimits.HasCoproducts C
  → Bifunctor C C C
CoproductBifunctor {Obj = Obj} C hasCoprod = let
  open Category C using (semigroupoid; idOp)
```

```
  open CatFinColimits C using (module HasCoproducts)
  open HasCoproducts hasCoprod using (_⊞_; _⊕_; ⊕-cong; ⨾-⊕-⨾; ⊕-Id)
  in record
    {obj = _⊞_
    ; mor = λ F G → F ⊕ G
    ; mor-cong = ⊕-cong
    ; mor-⨾ = ⨾-⊕-⨾
    ; mor-Id = ⊕-Id
    }
```

6.17 Categoric.Functor.Product

```
ProductBifunctor : {i j k : Level} {Obj : Set i}
  → (C : Category j k Obj)
  → CatFinLimits.HasProducts C
  → Bifunctor C C C
ProductBifunctor C hasProducts = let
    open CatFinLimits C using (module HasProducts)
    open HasProducts hasProducts using (_⊠_; _⊗_; ⊗-cong; ⨾-⊗-⨾; ⊗-Id)
  in record
    {obj = _⊠_
    ; mor = λ F G → F ⊗ G
    ; mor-cong = λ F₁≈F₂ G₁≈G₂ → ⊗-cong F₁≈F₂ G₁≈G₂
    ; mor-⨾ = ⨾-⊗-⨾
    ; mor-Id = ⊗-Id
    }
```

6.18 Categoric.Functor.CoEqualiser

```
module Categoric.Functor.CoEqualiser {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (C₁ : Category j₁ k₁ Obj₁)
                                     {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (C₂ : Category j₂ k₂ Obj₂) where
open FunctorSetup C₁ C₂
private
  module C₁ = Category C₁
  module C₂ = Category C₂
open CatFinColimits using (CoEqualiser; module CoEqualiser)
```

We show reflection of co-equalisers for each full and faithful functor that is surjective on objects (as witnessed by a left-inverse). (In the more general case, instead of obj-obj^{-1} there would be, for each A' : Obj_2, a B : Obj_1 together with an isomorphism from Functor.obj F B to A', that is, a natural isomorphism from G ⨾⨾ F to Identity, where G is the functor induced by obj^{-1}.)

```
FFReflectCoEqualiser :    (F : Functor C₁ C₂) → CatF.IsFullAndFaithful F
                     →    (obj⁻¹ : Obj₂ → Obj₁)
                     →    (obj-obj⁻¹ : {A′ : Obj₂} → Functor.obj F (obj⁻¹ A′) ≡ A′)
                     →    {A B : Obj₁} {f g : Mor₁ A B}
                     →    CoEqualiser C₂ (Functor.mor F f) (Functor.mor F g)
                     →    CoEqualiser C₁ f g
FFReflectCoEqualiser F isFF obj⁻¹ obj-obj⁻¹ {A} {B} {f₁} {g₁} CoEq = record
  {obj = Q₁
  ; mor = p₁
  ; prop = ≈₁-begin
```

```
      f₁ ⨾₁ p₁
    ≈₁⟨ C₁.⨾-cong₁ mor⁻¹-mor ⟨≈₁˘≈˘⟩ mor⁻¹-⨾ ⟩
      mor⁻¹ (mor f₁ ⨾₂ trg₂-expand p₂)
    ≈₁⟨ mor⁻¹-cong (C₂.⨾-≡-subst₃ Obj₂-expand ⟨≡≈₂⟩ trg₂-expand-cong f₂⨾p₂≈g₂⨾p₂
                                                    ⟨≈₂≡˘⟩ C₂.⨾-≡-subst₃ Obj₂-expand) ⟩
      mor⁻¹ (mor g₁ ⨾₂ trg₂-expand p₂)
    ≈₁⟨ mor⁻¹-⨾ ⟨≈₁≈⟩ C₁.⨾-cong₁ mor⁻¹-mor ⟩
      g₁ ⨾₁ p₁
    □₁
  ; universal = univ
  }
  where
    open Functor F
    open CatF F
    open FullAndFaithful isFF
    open SGF-FF-Inverse (CatF.sgFunctor F) obj⁻¹ obj-obj⁻¹
                        mor⁻¹ mor⁻¹-cong mor⁻¹-mor mor-mor⁻¹
    open CoEqualiser C₂ CoEq renaming (obj to Q₂; mor to p₂; prop to f₂⨾p₂≈g₂⨾p₂)
    Q₁ : Obj₁
    Q₁ = obj⁻¹ Q₂
    p₁ : Mor₁ B Q₁
    p₁ = mor⁻¹ (trg₂-expand p₂)
    univ : {Z₁ : Obj₁} {r₁ : Mor₁ B Z₁} (f₁⨾r₁≈g₁⨾r₁ : f₁ ⨾₁ r₁ ≈₁ g₁ ⨾₁ r₁)
        → ∃! _≈₁_ (λ u → r₁ ≈₁ p₁ ⨾₁ u)
    univ {Z₁} {r₁} f₁⨾r₁≈g₁⨾r₁ with universal {obj Z₁} {mor r₁}
      (≈₂-begin
          mor f₁ ⨾₂ mor r₁
        ≈₂˘⟨ mor-⨾ ⟩
          mor (f₁ ⨾₁ r₁)
        ≈₂⟨ mor-cong f₁⨾r₁≈g₁⨾r₁ ⟩
          mor (g₁ ⨾₁ r₁)
        ≈₂⟨ mor-⨾ ⟩
          mor g₁ ⨾₂ mor r₁
        □₂)
    ... | u₂, r₂≈p₂⨾u₂, u₂-unique = u₁, r₁≈p₁⨾u₁, (λ {y : Mor₁ (obj⁻¹ Q₂) Z₁} r₁≈p₁⨾y → ≈₁-begin
          u₁
        ≈₁⟨ ≈₁-refl ⟩
          mor⁻¹ (src₂-expand u₂)
        ≈₁⟨ mor⁻¹-cong (src₂-expand-cong (u₂-unique (≈₂-begin
            mor r₁
          ≈₂⟨ mor-cong r₁≈p₁⨾y ⟩
            mor (mor⁻¹ (trg₂-expand p₂) ⨾₁ y)
          ≈₂⟨ mor-⨾ ⟨≈₂≈⟩ C₂.⨾-cong₁ mor-mor⁻¹ ⟩
            trg₂-expand p₂ ⨾₂ mor y
          ≈₂≡˘⟨ C₂.⨾-≡-substSrc Obj₂-contract ⟩
            p₂ ⨾₂ src₂-contract (mor y)
          □₂))) ⟩
          mor⁻¹ (src₂-expand (src₂-contract (mor y)))
        ≈₁≡⟨ ≡-cong mor⁻¹ src₂-expand-contract ⟩
          mor⁻¹ (mor y)
        ≈₁⟨ mor⁻¹-mor ⟩
          y
        □₁)
      where
        u₁ : Mor₁ Q₁ Z₁
        u₁ = mor⁻¹ (src₂-expand u₂)
        r₂ : Mor₂ (obj B) (obj Z₁)
```

```
r₂ = mor r₁
r₁≈p₁⨾u₁ : r₁ ≈₁ p₁ ⨾₁ u₁
r₁≈p₁⨾u₁ = ≈₁-begin
      r₁
   ≈₁˘⟨ mor⁻¹-mor ⟩
      mor⁻¹ (mor r₁)
   ≈₁⟨ ≈₁-refl ⟩
      mor⁻¹ r₂
   ≈₁⟨ mor⁻¹-cong r₂≈p₂⨾u₂ ⟩
      mor⁻¹ (p₂ ⨾₂ u₂)
   ≈₁≡˘⟨ ≡-cong mor⁻¹ (C₂.⨾-≡-subst₂ Obj₂-expand) ⟩
      mor⁻¹ (trg₂-expand p₂ ⨾₂ src₂-expand u₂)
   ≈₁⟨ mor⁻¹-⨾ ⟩
      mor⁻¹ (trg₂-expand p₂) ⨾₁ mor⁻¹ (src₂-expand u₂)
   ≈₁⟨ ≈₁-refl ⟩
      p₁ ⨾₁ u₁
   □₁
```

Chapter 7

Monoidal Categories

7.1 Data.Fin.Fin2

```
0 ₂ : Fin 2
0 ₂ = zero
1 ₂ : Fin 2
1 ₂ = suc zero
```

```
[_,_] ₂ : {ℓ : Level} {S : Set ℓ} → S → S → Fin 2 → S
[a, b] ₂ zero = a
[a, b] ₂ (suc zero) = b
[a, b] ₂ (suc (suc ()))
```

```
β-[,] ₂ : {ℓ : Level} {S : Set ℓ} {A : Fin 2 → S} {k : Fin 2} → [A 0 ₂, A 1 ₂] ₂ k ≡ A k
β-[,] ₂ {ℓ} {S} {A} {zero} = ≡-refl
β-[,] ₂ {ℓ} {S} {A} {suc zero} = ≡-refl
β-[,] ₂ {ℓ} {S} {A} {suc (suc ())}
```

```
⟦_●_,_⟧₂ : {i m : Level} {Obj : Set i} (Mor : Obj → Obj → Set m)
            → let Mor′ = λ (A B : Fin 2 → Obj) → (s : Fin 2) → Mor (A s) (B s)
              in {A₁ B₁ A₂ B₂ : Obj} (F : Mor A₁ A₂) (G : Mor B₁ B₂)
            → Mor′ [A₁, B₁] ₂ [A₂, B₂] ₂
⟦_●_,_⟧₂ _ F G = λ {zero → F; (suc zero) → G; (suc (suc ()))}
```

7.2 Data.Fin.Fin3

```
0 ₃ : Fin 3
0 ₃ = zero
1 ₃ : Fin 3
1 ₃ = suc zero
2 ₃ : Fin 3
2 ₃ = suc (suc zero)
```

```
[_,_,_] ₃ : {ℓ : Level} {S : Set ℓ} → S → S → S → Fin 3 → S
[a, b, c] ₃ zero = a
[a, b, c] ₃ (suc zero) = b
[a, b, c] ₃ (suc (suc zero)) = c
[a, b, c] ₃ (suc (suc (suc ())))
```

```
⟦_•_,_,_⟧₃ : {i m : Level} {Obj : Set i} (Mor : Obj → Obj → Set m)
  → let Mor′ = λ (A B : Fin 3 → Obj) → (s : Fin 3) → Mor (A s) (B s)
    in {A₁ B₁ C₁ A₂ B₂ C₂ : Obj} (F : Mor A₁ A₂) (G : Mor B₁ B₂) (H : Mor C₁ C₂)
  → Mor′ [A₁, B₁, C₁] ₃ [A₂, B₂, C₂] ₃
⟦_•_,_,_⟧₃ _ F G H = λ {zero → F; (suc zero) → G; (suc (suc zero)) → H; (suc (suc (suc ())))}
```

7.3 Categoric.MonoidalCategory

As long as we abstain from using any equality on objects, we cannot model *strict* monoidal categories. Here, we provide a formalisation of general monoidal categories.

```
record MonoidalCategory {i : Level} (j k : Level) (Obj : Set i) : Set (i ⊍ ℓsuc j ⊍ ℓsuc k) where
  field category : Category j k Obj
  open Category category using
    (Mor; _⨾_; _≈_; Id; ≈-sym
    ; ≈-begin_; _≈⟨_⟩_; _≈˘⟨_⟩_; _□
    ; ⨾-cong₁; ⨾-cong₂; ⨾-cong₂₁; ⨾-cong₂₂; ⨾-cong₂₂₂; ⨾-assoc; ⨾-assocL
    ; ≈Id-isRightIdentity; ≈Id-isLeftIdentity; leftId; rightId; ⨾-assocL₄
    ; _⟨≈≈⟩_; _⟨≈˘≈⟩_; _⟨≈≈˘⟩_)

  field ⊗ : Bifunctor category category category
  open OTimes ⊗ using (_⊗o_; _⊗m_; ⨾-⊗-⨾; ⊗-cong; ⊗-cong₁; ⊗-cong₂; ⊗-Id)
```

For being able to express ⊗-associativity as a natural isomorphism, we create the triple-product base category as a sort-indexed product category:

```
  category³ : Category {i} j k (Fin 3 → Obj)
  category³ = SIPCategory (Fin 3) category

  ⊗-NestL : Functor category³ category
  ⊗-NestL = TriFunctorL ⊗ ⊗
  ⊗-NestR : Functor category³ category
  ⊗-NestR = TriFunctorR ⊗ ⊗

  field
    ⊗-Assoc : NatIso ⊗-NestL ⊗-NestR

  ⊗-assoc : {A B C : Obj} → Mor ((A ⊗o B) ⊗o C) (A ⊗o (B ⊗o C))
  ⊗-assoc {A} {B} {C} = NatIso.indmor ⊗-Assoc {[A, B, C] ₃}
  ⊗-assocL : {A B C : Obj} → Mor (A ⊗o (B ⊗o C)) ((A ⊗o B) ⊗o C)
  ⊗-assocL {A} {B} {C} = NatIso.indmor⁻¹ ⊗-Assoc {[A, B, C] ₃}
  ⊗-assoc-assocL : {A B C : Obj} → ⊗-assoc ⨾ ⊗-assocL ≈ Id {(A ⊗o B) ⊗o C}
  ⊗-assoc-assocL {A} {B} {C} = NatIso.indmor-⨾-indmor⁻¹ ⊗-Assoc {[A, B, C] ₃}
  ⊗-assocL-assoc : {A B C : Obj} → ⊗-assocL ⨾ ⊗-assoc ≈ Id {A ⊗o (B ⊗o C)}
  ⊗-assocL-assoc {A} {B} {C} = NatIso.indmor⁻¹-⨾-indmor ⊗-Assoc {[A, B, C] ₃}
  ⊗-assoc-natural : {A₁ B₁ C₁ A₂ B₂ C₂ : Obj} {F : Mor A₁ A₂} {G : Mor B₁ B₂} {H : Mor C₁ C₂}
                  → ((F ⊗m G) ⊗m H) ⨾ ⊗-assoc ≈ ⊗-assoc ⨾ (F ⊗m (G ⊗m H))
  ⊗-assoc-natural {A₁} {B₁} {C₁} {A₂} {B₂} {C₂} {F} {G} {H}
    = NatIso.naturality ⊗-Assoc {[A₁, B₁, C₁] ₃} {[A₂, B₂, C₂] ₃} {⟦ Mor • F, G, H ⟧₃}
  ⊗-assocL-natural : {A₁ B₁ C₁ A₂ B₂ C₂ : Obj} {F : Mor A₁ A₂} {G : Mor B₁ B₂} {H : Mor C₁ C₂}
                   → (F ⊗m (G ⊗m H)) ⨾ ⊗-assocL ≈ ⊗-assocL ⨾ ((F ⊗m G) ⊗m H)
```

```
⊗-assocL-natural {A₁} {B₁} {C₁} {A₂} {B₂} {C₂} {F} {G} {H}
  = NatIso.naturality⁻¹ ⊗-Assoc {[ A₁ , B₁ , C₁ ]₃} {[ A₂ , B₂ , C₂ ]₃} {⟦ Mor • F , G , H ⟧₃}


field
  ⊗-assoc-pentagon : {A B C D : Obj}
    → ⊗-assoc {A ⊗o B} {C} {D} ⨾ ⊗-assoc {A} {B} {C ⊗o D} ≈ (⊗-assoc ⊗m Id {D}) ⨾ ⊗-assoc ⨾ (Id {A} ⊗m ⊗-assoc)


⊗-assoc-pentagon₁ : {A B C D : Obj}
    → (⊗-assoc ⊗m Id {D}) ⨾ ⊗-assoc ≈ ⊗-assoc {A ⊗o B} {C} {D} ⨾ ⊗-assoc {A} {B} {C ⊗o D} ⨾ (Id {A} ⊗m ⊗-assocL)
⊗-assoc-pentagon₁ {A} {B} {C} {D} = ≈-begin
    (⊗-assoc ⊗m Id {D}) ⨾ ⊗-assoc
  ≈˘⟨ ⨾-cong₂ (≈Id-isRightIdentity ⊗-Id) ⟩
    (⊗-assoc ⊗m Id {D}) ⨾ ⊗-assoc ⨾ (Id {A} ⊗m Id {(B ⊗o C) ⊗o D})
  ≈˘⟨ ⨾-cong₂₂ (⊗-cong rightId ⊗-assoc-assocL) ⟩
    (⊗-assoc ⊗m Id {D}) ⨾ ⊗-assoc ⨾ ((Id {A} ⨾ Id {A}) ⊗m (⊗-assoc ⨾ ⊗-assocL))
  ≈⟨ ⨾-cong₂₂ ⨾-⊗-⨾ ⟩
    (⊗-assoc ⊗m Id {D}) ⨾ ⊗-assoc ⨾ (Id {A} ⊗m ⊗-assoc) ⨾ (Id {A} ⊗m ⊗-assocL)
  ≈⟨ ⨾-assocL₄ ⟨≈≈⟩ ⨾-cong₁ (⨾-assoc ⟨≈≈˘⟩ ⊗-assoc-pentagon) ⟨≈≈⟩ ⨾-assoc ⟩
    ⊗-assoc {A ⊗o B} {C} {D} ⨾ ⊗-assoc {A} {B} {C ⊗o D} ⨾ (Id {A} ⊗m ⊗-assocL)
  □

⊗-assocL-pentagon : {A B C D : Obj}
    → ⊗-assocL ⨾ ⊗-assocL ≈ (Id {A} ⊗m ⊗-assocL) ⨾ ⊗-assocL ⨾ (⊗-assocL ⊗m Id {D})
⊗-assocL-pentagon {A} {B} {C} {D} = ≈-begin
    ⊗-assocL ⨾ ⊗-assocL
  ≈˘⟨ ⨾-cong₂ (≈Id-isRightIdentity ⊗-Id) ⟩
    ⊗-assocL ⨾ ⊗-assocL ⨾ (Id {(A ⊗o B) ⊗o C} ⊗m Id {D})
  ≈˘⟨ ⨾-cong₂₂ (⊗-cong ⊗-assoc-assocL rightId) ⟩
    ⊗-assocL ⨾ ⊗-assocL ⨾ ((⊗-assoc ⨾ ⊗-assocL) ⊗m (Id {D} ⨾ Id {D}))
  ≈⟨ ⨾-cong₂₂ ⨾-⊗-⨾ ⟩
    ⊗-assocL ⨾ ⊗-assocL ⨾ (⊗-assoc ⊗m Id {D}) ⨾ (⊗-assocL ⊗m Id {D})
  ≈⟨ ⨾-cong₂₂₂ (≈Id-isLeftIdentity ⊗-assoc-assocL ⟨≈˘≈⟩ ⨾-assoc) ⟩
    ⊗-assocL ⨾ ⊗-assocL ⨾ (⊗-assoc ⊗m Id {D}) ⨾ ⊗-assoc ⨾ ⊗-assocL ⨾ (⊗-assocL ⊗m Id {D})
  ≈⟨ ⨾-cong₂₂ (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ ⊗-assoc-pentagon₁ ⟨≈≈⟩ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⨾-assoc) ⟩
    ⊗-assocL ⨾ ⊗-assocL ⨾ ⊗-assoc {A ⊗o B} {C} {D} ⨾ ⊗-assoc {A} {B} {C ⊗o D}
    ⨾ (Id {A} ⊗m ⊗-assocL) ⨾ ⊗-assocL ⨾ (⊗-assocL ⊗m Id {D})
  ≈⟨ ⨾-cong₂ (⨾-assocL ⟨≈≈⟩ ≈Id-isLeftIdentity ⊗-assocL-assoc) ⟩
    ⊗-assocL ⨾ ⊗-assoc {A} {B} {C ⊗o D} ⨾ (Id {A} ⊗m ⊗-assocL) ⨾ ⊗-assocL ⨾ (⊗-assocL ⊗m Id {D})
  ≈⟨ ⨾-assocL ⟨≈≈⟩ ≈Id-isLeftIdentity ⊗-assocL-assoc ⟩
    (Id {A} ⊗m ⊗-assocL) ⨾ ⊗-assocL ⨾ (⊗-assocL ⊗m Id {D})
  □


field
  ① : Obj


LeftUnit : Functor category category
LeftUnit = record
  {obj = λ A → ① ⊗o A
  ; mor = λ {A} {B} F → Id {①} ⊗m F
  ; mor-cong = λ {A} {B} {F} {G} → ⊗-cong₂
  ; mor-⨾ = λ {A} {B} {C} {F} {G} → ≈-begin
      Id {①} ⊗m (F ⨾ G)
    ≈˘⟨ ⊗-cong₁ leftId ⟩
      (Id {①} ⨾ Id {①}) ⊗m (F ⨾ G)
    ≈⟨ ⨾-⊗-⨾ ⟩
      (Id {①} ⊗m F) ⨾ (Id {①} ⊗m G)
    □
```

```
  ; mor-Id  =  λ {A} → ⊗-Id
  }
RightUnit : Functor category category
RightUnit = record
  {obj  =  λ A → A ⊗o 𝟙
  ; mor  =  λ {A} {B} F → F ⊗m Id {𝟙}
  ; mor-cong  =  λ {A} {B} {F} {G} → ⊗-cong₁
  ; mor-⨾  =  λ {A} {B} {C} {F} {G} → ≈-begin
        (F ⨾ G) ⊗m Id {𝟙}
      ≈˘⟨ ⊗-cong₂ leftId ⟩
        (F ⨾ G) ⊗m (Id {𝟙} ⨾ Id {𝟙})
      ≈⟨ ⨾-⊗-⨾ ⟩
        (F ⊗m Id {𝟙}) ⨾ (G ⊗m Id {𝟙})
      □
  ; mor-Id  =  λ {A} → ⊗-Id
  }

field
  ⊗-LeftUnit : NatIso LeftUnit (Identity _)
  ⊗-RightUnit : NatIso RightUnit (Identity _)

⊗-leftUnit      : {A : Obj} → Mor (𝟙 ⊗o A) A
⊗-leftUnit  =  NatIso.indmor ⊗-LeftUnit
⊗-leftUnit⁻¹    : {A : Obj} → Mor A (𝟙 ⊗o A)
⊗-leftUnit⁻¹  =  NatIso.indmor⁻¹ ⊗-LeftUnit
⊗-rightUnit     : {A : Obj} → Mor (A ⊗o 𝟙) A
⊗-rightUnit  =  NatIso.indmor ⊗-RightUnit
⊗-rightUnit⁻¹  : {A : Obj} → Mor A (A ⊗o 𝟙)
⊗-rightUnit⁻¹  =  NatIso.indmor⁻¹ ⊗-RightUnit

field
  ⊗-triangle : {A B : Obj} → ⊗-assoc {A} {𝟙} {B} ⨾ (Id {A} ⊗m ⊗-leftUnit) ≈ ⊗-rightUnit ⊗m Id {B}
  ⊗-leftUnit-𝟙  : ⊗-leftUnit {𝟙} ≈ ⊗-rightUnit {𝟙}

⊗-leftUnit-leftUnit⁻¹  : {A : Obj} → ⊗-leftUnit ⨾ ⊗-leftUnit⁻¹ ≈ Id {𝟙 ⊗o A}
⊗-leftUnit-leftUnit⁻¹  =  NatIso.indmor-⨾-indmor⁻¹ ⊗-LeftUnit
⊗-leftUnit⁻¹-leftUnit  : {A : Obj} → ⊗-leftUnit⁻¹ ⨾ ⊗-leftUnit ≈ Id {A}
⊗-leftUnit⁻¹-leftUnit  =  NatIso.indmor⁻¹-⨾-indmor ⊗-LeftUnit
⊗-rightUnit-rightUnit⁻¹  : {A : Obj} → ⊗-rightUnit ⨾ ⊗-rightUnit⁻¹ ≈ Id {A ⊗o 𝟙}
⊗-rightUnit-rightUnit⁻¹  =  NatIso.indmor-⨾-indmor⁻¹ ⊗-RightUnit
⊗-rightUnit⁻¹-rightUnit  : {A : Obj} → ⊗-rightUnit⁻¹ ⨾ ⊗-rightUnit ≈ Id {A}
⊗-rightUnit⁻¹-rightUnit  =  NatIso.indmor⁻¹-⨾-indmor ⊗-RightUnit
⊗-⨾-leftUnit  : {A B : Obj} {F : Mor A B} → Id {𝟙} ⊗m F ⨾ ⊗-leftUnit ≈ ⊗-leftUnit ⨾ F
⊗-⨾-leftUnit  =  NatIso.naturality ⊗-LeftUnit
⊗-⨾-rightUnit  : {A B : Obj} {F : Mor A B} → F ⊗m Id {𝟙} ⨾ ⊗-rightUnit ≈ ⊗-rightUnit ⨾ F
⊗-⨾-rightUnit  =  NatIso.naturality ⊗-RightUnit
⊗-leftUnit⁻¹-⨾  : {A B : Obj} {F : Mor A B} → ⊗-leftUnit⁻¹ ⨾ (Id {𝟙} ⊗m F) ≈ F ⨾ ⊗-leftUnit⁻¹
⊗-leftUnit⁻¹-⨾  =  ≈-sym (NatIso.naturality⁻¹ ⊗-LeftUnit)
⊗-rightUnit⁻¹-⨾  : {A B : Obj} {F : Mor A B} → ⊗-rightUnit⁻¹ ⨾ (F ⊗m Id {𝟙}) ≈ F ⨾ ⊗-rightUnit⁻¹
⊗-rightUnit⁻¹-⨾  =  ≈-sym (NatIso.naturality⁻¹ ⊗-RightUnit)

⊗-leftUnit⁻¹-𝟙  : ⊗-leftUnit⁻¹ {𝟙} ≈ ⊗-rightUnit⁻¹ {𝟙}
⊗-leftUnit⁻¹-𝟙  =  ≈-begin
    ⊗-leftUnit⁻¹ {𝟙}
```

```
  ≈˘⟨ ⨾-cong₂ ⊗-rightUnit-rightUnit⁻¹ ⟨≈≈⟩ rightId ⟩
    ⊗-leftUnit⁻¹ {𝟙} ⨾ ⊗-rightUnit {𝟙} ⨾ ⊗-rightUnit⁻¹ {𝟙}
  ≈˘⟨ ⨾-cong₂₁ ⊗-leftUnit-𝟙 ⟩
    ⊗-leftUnit⁻¹ {𝟙} ⨾ ⊗-leftUnit {𝟙} ⨾ ⊗-rightUnit⁻¹ {𝟙}
  ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ ⊗-leftUnit⁻¹-leftUnit ⟨≈≈⟩ leftId ⟩
    ⊗-rightUnit⁻¹ {𝟙}
  □

open OTimes ⊗ public
```

7.4 Categoric.MonoidalCategory.Sym

```
module SymNatTrans
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} {C₁ : Category j₁ k₁ Obj₁}
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} {C₂ : Category j₂ k₂ Obj₂}
  (⊗ : Bifunctor C₁ C₁ C₂)
  (Sym : NatTrans (BiFunctor ⊗) (BiFunctor (Flip ⊗)))
  where
  open OTimes ⊗ using (_⊗o_; _⊗m_)
  open Category C₁ using () renaming (Mor to Mor₁)
  open Category C₂ using (_⨾_; _≈_) renaming (Mor to Mor₂)
  swap : {A B : Obj₁} → Mor₂ (A ⊗o B) (B ⊗o A)
  swap {A} {B} = NatTrans.indmor Sym {[A, B]₂}
  swap-natural : {A₁ A₂ B₁ B₂ : Obj₁} {F : Mor₁ A₁ A₂} {G : Mor₁ B₁ B₂}
               → (F ⊗m G) ⨾ swap {A₂} {B₂} ≈ swap {A₁} {B₁} ⨾ (G ⊗m F)
  swap-natural {F = F} {G} = NatTrans.naturality Sym {f = ⟦ Mor₁ • F, G ⟧₂}
```

We follow Mac Lane (1971) with the conditions for symmetric monoidal categories:

```
record MonCatSym {i j k : Level} {Obj : Set i} (MC : MonoidalCategory j k Obj) : Set (i ⊍ j ⊍ k) where
  open MonoidalCategory MC using
    (category; _⊗o_; _⊗m_; 𝟙; ⊗
    ; ⨾-⊗-⨾; ⊗-cong; ⊗-cong₁; ⊗-cong₂; ⊗-Id
    ; ⊗-assoc; ⊗-assocL; ⊗-assoc-assocL; ⊗-assocL-assoc
    ; ⊗-leftUnit   ; ⊗-leftUnit⁻¹   ; ⊗-leftUnit-leftUnit⁻¹
    ; ⊗-rightUnit; ⊗-rightUnit⁻¹; ⊗-rightUnit⁻¹-rightUnit
    ; ⊗-assoc-natural; ⊗-assocL-natural)
  open Category category using
    (Mor; _⨾_; _≈_; Id; ≈-sym
    ; ≈-begin_; _≈˘⟨_⟩_; _≈⟨_⟩_; _□
    ; ⨾-cong₂; ⨾-assocL; ≈-trans; ⨾-cong₁; leftId; rightId
    ; pairedToIdMor³; hasInverse-cong
    ; ≈-refl; ⨾-assoc; _⟨≈≈⟩_; _⟨≈≈˘⟩_; ⨾-cong₁₂; ⨾-cong₁₁; ⨾-cong₂₁)
    -- field Sym : NatTrans (BiFunctor ⊗) (BiFunctor (Flip ⊗))
    -- open SymNatTrans ⊗ Sym
  field
    swap            : {A B : Obj} → Mor (A ⊗o B) (B ⊗o A)
    swap-natural    : {A₁ A₂ B₁ B₂ : Obj} {F : Mor A₁ A₂} {G : Mor B₁ B₂}
                    → (F ⊗m G) ⨾ swap {A₂} {B₂} ≈ swap {A₁} {B₁} ⨾ (G ⊗m F)
    swap-cancel     : {A B : Obj} → swap {A} {B} ⨾ swap {B} {A} ≈ Id
    swap-unit       : swap {𝟙} {𝟙} ≈ Id {𝟙 ⊗o 𝟙}
    swap-monoidal   : {A B C : Obj}
                    → (Id {A} ⊗m swap {B} {C}) ⨾ ⊗-assocL {A} {C} {B} ⨾ (swap {A} {C} ⊗m Id {B})
                    ≈ ⊗-assocL {A} {B} {C} ⨾ swap {A ⊗o B} {C} ⨾ ⊗-assocL {C} {A} {B}
    swap-⊗-leftUnit : {A : Obj} → swap ⨾ ⊗-leftUnit ≈ ⊗-rightUnit {A}
```

From swap, swap-natural, and swap-cancel we can assemble a natural isomorphism; the implicit arguments to swap in the definitions of indmor and indmor^{-1} are supplied only for documentation:

```
SymNatIso : NatIso (BiFunctor ⊗) (BiFunctor (Flip ⊗))
SymNatIso = record
  {indmor            = λ {A} → swap {A 0 ₂} {A 1 ₂}
  ; naturality       = λ {A} {B} {F} → swap-natural
  ; indmor⁻¹         = λ {A} → swap {A 1 ₂} {A 0 ₂}
  ; indmor-⨾-indmor⁻¹ = λ {A} → swap-cancel
  ; indmor⁻¹-⨾-indmor = λ {A} → swap-cancel
  }


⊗-⨾-swap : {A₁ A₂ B₁ B₂ : Obj} {F : Mor A₁ A₂} {G : Mor B₁ B₂}
         → (F ⊗m G) ⨾ swap {A₂} {B₂} ≈ swap {A₁} {B₁} ⨾ (G ⊗m F)
⊗-⨾-swap = swap-natural

swap-⨾-⊗ : {A₁ A₂ B₁ B₂ : Obj} {F : Mor A₁ A₂} {G : Mor B₁ B₂}
         → swap {A₁} {B₁} ⨾ (G ⊗m F) ≈ (F ⊗m G) ⨾ swap {A₂} {B₂}
swap-⨾-⊗ = ≈-sym swap-natural

swap-⊗-rightUnit : {A : Obj} → swap ⨾ ⊗-rightUnit ≈ ⊗-leftUnit {A}
swap-⊗-rightUnit {A} = ≈-begin
            swap ⨾ ⊗-rightUnit
          ≈˘⟨ ⨾-cong₂ swap-⊗-leftUnit ⟩
            swap ⨾ swap ⨾ ⊗-leftUnit
          ≈⟨ ⨾-assocL ⟨ ≈-trans ⟩ ⨾-cong₁ swap-cancel ⟨ ≈-trans ⟩ leftId ⟩
            ⊗-leftUnit
          □

⊗-rightUnit⁻¹-swap : {A : Obj} → ⊗-rightUnit⁻¹ {A} ⨾ swap ≈ ⊗-leftUnit⁻¹
⊗-rightUnit⁻¹-swap {A} = ≈-begin
            ⊗-rightUnit⁻¹ {A} ⨾ swap
          ≈˘⟨ ⨾-cong₂ (⨾-cong₂ ⊗-leftUnit-leftUnit⁻¹ ⟨ ≈-trans ⟩ rightId) ⟩
            ⊗-rightUnit⁻¹ {A} ⨾ swap ⨾ ⊗-leftUnit ⨾ ⊗-leftUnit⁻¹
          ≈⟨ ⨾-cong₂ (⨾-assocL ⟨ ≈-trans ⟩ ⨾-cong₁ swap-⊗-leftUnit) ⟩
            ⊗-rightUnit⁻¹ {A} ⨾ ⊗-rightUnit ⨾ ⊗-leftUnit⁻¹
          ≈⟨ ⨾-assocL ⟨ ≈-trans ⟩ ⨾-cong₁ ⊗-rightUnit⁻¹-rightUnit ⟨ ≈-trans ⟩ leftId ⟩
            ⊗-leftUnit⁻¹
          □

⊗-leftUnit⁻¹-swap : {A : Obj} → ⊗-leftUnit⁻¹ {A} ⨾ swap ≈ ⊗-rightUnit⁻¹
⊗-leftUnit⁻¹-swap {A} = ≈-begin
            ⊗-leftUnit⁻¹ {A} ⨾ swap
          ≈˘⟨ ⨾-assocL ⟨ ≈-trans ⟩ ⨾-cong₁ ⊗-rightUnit⁻¹-swap ⟩
            ⊗-rightUnit⁻¹ {A} ⨾ swap ⨾ swap
          ≈⟨ ⨾-cong₂ swap-cancel ⟨ ≈-trans ⟩ rightId ⟩
            ⊗-rightUnit⁻¹
          □


swapId : {A B C : Obj} → Mor ((A ⊗o B) ⊗o C) ((B ⊗o A) ⊗o C)
swapId = swap ⊗m Id

swapId² : {A B C : Obj} → swapId {A} {B} {C} ⨾ swapId {B} {A} {C} ≈ Id {(A ⊗o B) ⊗o C}
swapId² = ≈-begin
          swapId ⨾ swapId
        ≈˘⟨ ⨾-⊗-⨾ ⟩
          (swap ⨾ swap) ⊗m (Id ⨾ Id)
        ≈⟨ ⊗-cong swap-cancel leftId ⟩
          Id ⊗m Id
        ≈⟨ ⊗-Id ⟩
```

```
        Id
      □

idSwap : {A B C : Obj} → Mor (A ⊗o (B ⊗o C)) (A ⊗o (C ⊗o B))
idSwap = Id ⊗m swap

idSwap² : {A B C : Obj} → idSwap {A} {B} {C} ⨾ idSwap {A} {C} {B} ≈ Id {A ⊗o (B ⊗o C)}
idSwap² = ≈-begin
          idSwap ⨾ idSwap
        ≈˘⟨ ⨾-⊗-⨾ ⟩
          (Id ⨾ Id) ⊗m (swap ⨾ swap)
        ≈⟨ ⊗-cong leftId swap-cancel ⟩
          Id ⊗m Id
        ≈⟨ ⊗-Id ⟩
          Id
        □

swapMonLHS-compInv : {A B C : Obj}
  →     ((swap {C} {A} ⊗m Id {B}) ⨾ ⊗-assoc {A} {C} {B} ⨾ (Id {A} ⊗m swap {C} {B}))
        ⨾ ((Id {A} ⊗m swap {B} {C}) ⨾ ⊗-assocL {A} {C} {B} ⨾ (swap {A} {C} ⊗m Id {B}))
        ≈ Id
swapMonLHS-compInv = pairedToIdMor³ swapId² ⊗-assoc-assocL idSwap²

swapMonRHS-compInv : {A B C : Obj}
  →     (⊗-assocL {A} {B} {C} ⨾ swap {A ⊗o B} {C} ⨾ ⊗-assocL {C} {A} {B})
        ⨾ (⊗-assoc {C} {A} {B} ⨾ swap {C} {A ⊗o B} ⨾ ⊗-assoc {A} {B} {C})
        ≈ Id
swapMonRHS-compInv = pairedToIdMor³ ⊗-assocL-assoc swap-cancel ⊗-assocL-assoc

swap-monoidal′ : {A B C : Obj}
             → (swap {C} {A} ⊗m Id {B}) ⨾ ⊗-assoc {A} {C} {B} ⨾ (Id {A} ⊗m swap {C} {B})
               ≈ ⊗-assoc {C} {A} {B} ⨾ swap {C} {A ⊗o B} ⨾ ⊗-assoc {A} {B} {C}
swap-monoidal′ = ≈-begin
         (swap ⊗m Id) ⨾ ⊗-assoc ⨾ (Id ⊗m swap)
       ≈⟨ hasInverse-cong swapMonLHS-compInv swapMonRHS-compInv swap-monoidal ⟩
         ⊗-assoc ⨾ swap ⨾ ⊗-assoc
       □


⊗-transpose₂ : {A B C D : Obj} → Mor ((A ⊗o B) ⊗o (C ⊗o D)) ((A ⊗o C) ⊗o (B ⊗o D))
⊗-transpose₂ = (⊗-assoc ⨾ (Id ⊗m ((⊗-assocL ⨾ (swap ⊗m Id)) ⨾ ⊗-assoc))) ⨾ ⊗-assocL

swapId-⨾ : {A B C D E : Obj} {F : Mor (A ⊗o B) C} {G : Mor D E}
        → swapId ⨾ (F ⊗m G) ≈ (swap ⨾ F) ⊗m G
swapId-⨾ {F = F} {G} = ≈-begin
          swapId ⨾ (F ⊗m G)
        ≈⟨ ≈-refl ⟩
          (swap ⊗m Id) ⨾ (F ⊗m G)
        ≈˘⟨ ⨾-⊗-⨾ ⟩
          (swap ⨾ F) ⊗m (Id ⨾ G)
        ≈⟨ ⊗-cong₂ leftId ⟩
          (swap ⨾ F) ⊗m G
        □

asoclSwapIdAsoc : {A B C : Obj} → Mor (A ⊗o (B ⊗o C)) (B ⊗o (A ⊗o C))
asoclSwapIdAsoc = (⊗-assocL ⨾ (swap ⊗m Id)) ⨾ ⊗-assoc

asoclSwapIdAsoc-⨾ : {A B C D E F : Obj} {G : Mor A B} {H : Mor C D} {K : Mor E F}
                 → asoclSwapIdAsoc ⨾ (G ⊗m (H ⊗m K))
                   ≈ (H ⊗m (G ⊗m K)) ⨾ asoclSwapIdAsoc
asoclSwapIdAsoc-⨾ {G = G} {H} {K} = ≈-begin
          asoclSwapIdAsoc ⨾ (G ⊗m (H ⊗m K))
        ≈⟨ ≈-refl ⟩
          ((⊗-assocL ⨾ swapId) ⨾ ⊗-assoc) ⨾ (G ⊗m (H ⊗m K))
        ≈⟨ ⨾-assoc ⟩
```

```
      (⊗-assocL ⨾ swapId) ⨾ (⊗-assoc ⨾ (G ⊗m (H ⊗m K)))
    ≈˘⟨ ⨾-cong₂ ⊗-assoc-natural ⟩
      (⊗-assocL ⨾ swapId) ⨾ (((G ⊗m H) ⊗m K) ⨾ ⊗-assoc)
    ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ ⨾-assoc ⟩
      (⊗-assocL ⨾ (swapId ⨾ ((G ⊗m H) ⊗m K))) ⨾ ⊗-assoc
    ≈⟨ ⨾-cong₁₂ swapId-⨾ ⟩
      (⊗-assocL ⨾ ((swap ⨾ (G ⊗m H)) ⊗m K)) ⨾ ⊗-assoc
    ≈⟨ ⨾-cong₁₂ (⊗-cong₁ swap-⨾-⊗) ⟩
      (⊗-assocL ⨾ (((H ⊗m G) ⨾ swap) ⊗m K)) ⨾ ⊗-assoc
    ≈˘⟨ ⨾-cong₁₂ (⊗-cong₂ rightId) ⟩
      (⊗-assocL ⨾ (((H ⊗m G) ⨾ swap) ⊗m (K ⨾ Id))) ⨾ ⊗-assoc
    ≈⟨ ⨾-cong₁₂ ⨾-⊗-⨾ ⟩
      (⊗-assocL ⨾ (((H ⊗m G) ⊗m K) ⨾ (swap ⊗m Id))) ⨾ ⊗-assoc
    ≈⟨ ⨾-cong₁ ⨾-assocL ⟨≈≈˘⟩ ⨾-cong₁₁ ⊗-assocL-natural ⟩
      (((H ⊗m (G ⊗m K)) ⨾ ⊗-assocL) ⨾ swapId) ⨾ ⊗-assoc
    ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-assoc ⟨≈≈⟩ (⨾-cong₂ ⨾-assocL) ⟩
      (H ⊗m (G ⊗m K)) ⨾ asoclSwapIdAsoc
    □

asoclSwapIdAsoc² : {A B C : Obj}
                 → asoclSwapIdAsoc {A} {B} {C} ⨾ asoclSwapIdAsoc ≈ Id
asoclSwapIdAsoc² = ≈-begin
        asoclSwapIdAsoc ⨾ asoclSwapIdAsoc
      ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⨾-assocL ⟨≈≈⟩ ⨾-cong₂ (⨾-cong₁ ⨾-assocL) ⟩
        (⊗-assocL ⨾ swapId)
        ⨾ (((⊗-assoc ⨾ ⊗-assocL) ⨾ swapId) ⨾ ⊗-assoc)
      ≈⟨ ⨾-cong₂₁ (⨾-cong₁ ⊗-assoc-assocL ⟨≈≈⟩ leftId) ⟩
        (⊗-assocL ⨾ swapId)
        ⨾ (swapId ⨾ ⊗-assoc)
      ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⨾-assocL ⟩
        ⊗-assocL ⨾ ((swapId ⨾ swapId) ⨾ ⊗-assoc)
      ≈⟨ ⨾-cong₂ (⨾-cong₁ swapId²) ⟩
        ⊗-assocL ⨾ (Id ⨾ ⊗-assoc)
      ≈⟨ ⨾-cong₂ leftId ⟨≈≈⟩ ⊗-assocL-assoc ⟩
        Id
      □

IdAsoclSwapIdAsoc : {A B C D : Obj} → Mor (A ⊗o (B ⊗o (C ⊗o D))) (A ⊗o (C ⊗o (B ⊗o D)))
IdAsoclSwapIdAsoc = Id ⊗m asoclSwapIdAsoc

IdAsoclSwapIdAsoc² : {A B C D : Obj}
                   → IdAsoclSwapIdAsoc {A} {B} {C} {D} ⨾ IdAsoclSwapIdAsoc ≈ Id
IdAsoclSwapIdAsoc² = ≈-begin
        IdAsoclSwapIdAsoc ⨾ IdAsoclSwapIdAsoc
      ≈⟨ ≈-refl ⟩
        (Id ⊗m asoclSwapIdAsoc) ⨾ (Id ⊗m asoclSwapIdAsoc)
      ≈˘⟨ ⨾-⊗-⨾ ⟩
        (Id ⨾ Id) ⊗m (asoclSwapIdAsoc ⨾ asoclSwapIdAsoc)
      ≈⟨ ⊗-cong leftId asoclSwapIdAsoc² ⟩
        Id ⊗m Id
      ≈⟨ ⊗-Id ⟩
        Id
      □

⊗-transpose₂² : {A B C D : Obj} → ⊗-transpose₂ {A} {B} {C} {D} ⨾ ⊗-transpose₂ ≈ Id
⊗-transpose₂² {A} {B} {C} {D} = ≈-begin
        ⊗-transpose₂ ⨾ ⊗-transpose₂
      ≈⟨ ≈-refl ⟩
        ((⊗-assoc ⨾ IdAsoclSwapIdAsoc) ⨾ ⊗-assocL)
        ⨾ ((⊗-assoc ⨾ IdAsoclSwapIdAsoc) ⨾ ⊗-assocL)
      ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⨾-assocL ⟨≈≈⟩ ⨾-cong₂₁ ⨾-assocL ⟩
```

```
                (⊗-assoc ⨾ IdAsoclSwapIdAsoc)
                ⨾ (((⊗-assocL ⨾ ⊗-assoc) ⨾ IdAsoclSwapIdAsoc) ⨾ ⊗-assocL)
              ≈⟨ ⨾-cong₂₁ (⨾-cong₁ ⊗-assocL-assoc ⟨≈≈⟩ leftId) ⟩
                (⊗-assoc ⨾ IdAsoclSwapIdAsoc)
                ⨾ (IdAsoclSwapIdAsoc ⨾ ⊗-assocL)
              ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⨾-assocL ⟩
                ⊗-assoc ⨾ ((IdAsoclSwapIdAsoc ⨾ IdAsoclSwapIdAsoc) ⨾ ⊗-assocL)
              ≈⟨ ⨾-cong₂₁ IdAsoclSwapIdAsoc² ⟩
                ⊗-assoc ⨾ (Id ⨾ ⊗-assocL)
              ≈⟨ ⨾-cong₂ leftId ⟨≈≈⟩ ⊗-assoc-assocL ⟩
                Id
              □

  ⨾-⊗-transpose₂  : {A₁ B₁ C₁ D₁ A₂ B₂ C₂ D₂ : Obj} {F : Mor A₁ A₂}
                     {G : Mor B₁ B₂} {H : Mor C₁ C₂} {K : Mor D₁ D₂}
                → (F ⊗m H) ⊗m (G ⊗m K) ⨾ ⊗-transpose₂ ≈ ⊗-transpose₂ ⨾ (F ⊗m G) ⊗m (H ⊗m K)
  ⨾-⊗-transpose₂ {F = F} {G} {H} {K} = ≈-sym (≈-begin
                ⊗-transpose₂ ⨾ (F ⊗m G) ⊗m (H ⊗m K)
              ≈⟨ ≈-refl ⟩
                ((⊗-assoc ⨾ IdAsoclSwapIdAsoc) ⨾ ⊗-assocL) ⨾ (F ⊗m G) ⊗m (H ⊗m K)
              ≈⟨ ⨾-assoc ⟩
                 (⊗-assoc ⨾ IdAsoclSwapIdAsoc) ⨾ (⊗-assocL ⨾ (F ⊗m G) ⊗m (H ⊗m K))
              ≈˘⟨ ⨾-cong₂ ⊗-assocL-natural ⟩
                (⊗-assoc ⨾ IdAsoclSwapIdAsoc) ⨾ ((F ⊗m (G ⊗m (H ⊗m K))) ⨾ ⊗-assocL)
              ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ ⨾-assoc ⟩
                (⊗-assoc ⨾ ((Id ⊗m asoclSwapIdAsoc) ⨾ (F ⊗m (G ⊗m (H ⊗m K))))) ⨾ ⊗-assocL
              ≈˘⟨ ⨾-cong₁₂ ⨾-⊗-⨾ ⟩
                (⊗-assoc ⨾ ((Id ⨾ F) ⊗m (asoclSwapIdAsoc ⨾ (G ⊗m (H ⊗m K))))) ⨾ ⊗-assocL
              ≈⟨ ⨾-cong₁₂ (⊗-cong (leftId ⟨≈≈˘⟩ rightId) asoclSwapIdAsoc-⨾) ⟩
                (⊗-assoc ⨾ ((F ⨾ Id) ⊗m ((H ⊗m (G ⊗m K)) ⨾ asoclSwapIdAsoc))) ⨾ ⊗-assocL
              ≈⟨ ⨾-cong₁₂ ⨾-⊗-⨾ ⟩
                (⊗-assoc ⨾ ((F ⊗m (H ⊗m (G ⊗m K))) ⨾ (Id ⊗m asoclSwapIdAsoc))) ⨾ ⊗-assocL
              ≈⟨ ⨾-cong₁ ⨾-assocL ⟩
                ((⊗-assoc ⨾ (F ⊗m (H ⊗m (G ⊗m K)))) ⨾ IdAsoclSwapIdAsoc) ⨾ ⊗-assocL
              ≈˘⟨ ⨾-cong₁₁ ⊗-assoc-natural ⟩
                ((((F ⊗m H) ⊗m (G ⊗m K)) ⨾ ⊗-assoc) ⨾ IdAsoclSwapIdAsoc) ⨾ ⊗-assocL
              ≈⟨ ⨾-cong₁ ⨾-assoc ⟨≈≈⟩ ⨾-assoc ⟩
                ((F ⊗m H) ⊗m (G ⊗m K)) ⨾ ((⊗-assoc ⨾ IdAsoclSwapIdAsoc) ⨾ ⊗-assocL)
              ≈⟨ ≈-refl ⟩
                ((F ⊗m H) ⊗m (G ⊗m K)) ⨾ ⊗-transpose₂
              □)
```

7.5 Categoric.MonoidalCategory.GS

The symbol ！ is the Unicode codepoint 0xFF01 (fullwidth exclamation point).

```
record MonCatG {i j k : Level} {Obj : Set i} (monCat : MonoidalCategory j k Obj) : Set (i ⊍ j ⊍ k) where
  open MonoidalCategory monCat using
    (category; ①; _⊗o_; _⊗m_; ⊗-leftUnit)
  open Category category using
    (Mor; _≈_; Id; _⨾_)
  field
    ！           : {A : Obj} → Mor A ①
    ！-unit      : ！ {①} ≈ Id {①}
    ！-monoidal : {A B : Obj} → ！ {A ⊗o B} ≈ ！ {A} ⊗m ！ {B} ⨾ ⊗-leftUnit {①}

record MonCatS {i j k : Level} {Obj : Set i}
               (monCat : MonoidalCategory j k Obj)
```

```
                    (monCatSym : MonCatSym monCat) : Set (i ⊍ j ⊍ k) where
  open MonoidalCategory monCat using
    (category; _⊗o_; _⊗m_; 𝟙
    ; ⊗-leftUnit⁻¹; ⊗-assocL; ⊗-assoc; ⊗-assocL-assoc)
  open Category category using
    (Mor; _≈_; _⨾_; Id
    ; ≈-begin_; _≈˘⟨_⟩_; _≈⟨_⟩_; _□
    ; ⨾-cong₂; ≈-trans; rightId; ⨾-assocL; ⨾-cong₁; ⨾-assoc₃₊₁)
  open MonCatSym monCatSym using (swap)
  field
    ∇           :  {A : Obj} → Mor A (A ⊗o A)
    ∇-unit      :  ∇ {𝟙} ≈ ⊗-leftUnit⁻¹ {𝟙}
    ∇-assoc     :  {A : Obj}
                 → ∇ {A} ⨾ (∇ {A} ⊗m Id {A})
                 ≈  ∇ {A} ⨾ (Id {A} ⊗m ∇ {A}) ⨾ ⊗-assocL {A} {A} {A}
    ∇-⨾-swap    :  {A : Obj} → ∇ {A} ⨾ swap {A} {A} ≈ ∇ {A}
    ∇-monoidal  :  {A B : Obj}
                 → ∇ {A ⊗o B} ⨾ ⊗-assoc {A} {B} {A ⊗o B} ⨾
                   (Id {A} ⊗m (⊗-assocL {B} {A} {B} ⨾ (swap {B} {A} ⊗m Id {B})))
                 ≈  ∇ {A} ⊗m ∇ {B} ⨾ ⊗-assoc {A} {A} {B ⊗o B} ⨾
                   (Id {A} ⊗m ⊗-assocL {A} {B} {B})
  ∇-assocL      :  {A : Obj}
                 → ∇ {A} ⨾ (Id {A} ⊗m ∇ {A})
                 ≈  ∇ {A} ⨾ (∇ {A} ⊗m Id {A}) ⨾ ⊗-assoc {A} {A} {A}
  ∇-assocL {A} = ≈-begin
      ∇ {A} ⨾ (Id {A} ⊗m ∇ {A})
    ≈˘⟨ ⨾-cong₂ (⨾-cong₂ ⊗-assocL-assoc ⟨ ≈-trans ⟩ rightId) ⟩
      ∇ {A} ⨾ (Id {A} ⊗m ∇ {A}) ⨾ ⊗-assocL ⨾ ⊗-assoc
    ≈˘⟨ ⨾-assocL ⟨ ≈-trans ⟩ ⨾-cong₁ ∇-assoc ⟨ ≈-trans ⟩ ⨾-assoc₃₊₁ ⟩
      ∇ {A} ⨾ (∇ {A} ⊗m Id {A}) ⨾ ⊗-assoc {A} {A} {A}
    □
```

We offer the **record** MonCatGS to organise gs-monoidal categories as an extension of symmetric monoidal categories:

```
record MonCatGS {i j k : Level} {Obj : Set i}
                (monCat    : MonoidalCategory j k Obj)
                (monCatSym : MonCatSym monCat)
                : Set (i ⊍ j ⊍ k) where
  field monCatG : MonCatG monCat
        monCatS : MonCatS monCat monCatSym
  open MonoidalCategory monCat using (category; _⊗m_; ⊗-rightUnit⁻¹; ⊗-leftUnit⁻¹)
  open Category category using
    (_⨾_; Id; _≈_
    ; ⨾-assocL₃₊₁; ≈-trans; ⨾-cong₁; ⨾-cong₂; rightId; ⨾-assocL; ⨾-cong
    ; ≈-begin_; _≈˘⟨_⟩_; _≈⟨_⟩_; _□)
  open MonCatSym monCatSym using
    (swap-cancel; swap; swap-⨾-⊗; ⊗-rightUnit⁻¹-swap)
  open MonCatG monCatG using (!)
  open MonCatS monCatS using (∇; ∇-⨾-swap)
  field ∇-rightInv : {A : Obj} → ∇ {A} ⨾ (Id {A} ⊗m ! {A}) ≈ ⊗-rightUnit⁻¹
  ∇-rightInv′ : {A : Obj} → ∇ {A} ⨾ (! {A} ⊗m Id {A}) ≈ ⊗-leftUnit⁻¹
  ∇-rightInv′ {A} = ≈-begin
      ∇ {A} ⨾ (! {A} ⊗m Id {A})
    ≈˘⟨ ⨾-assocL₃₊₁ ⟨ ≈-trans ⟩ ⨾-cong₁ (⨾-cong₂ swap-cancel ⟨ ≈-trans ⟩ rightId) ⟩
      ∇ {A} ⨾ swap ⨾ swap ⨾ (! {A} ⊗m Id {A})
    ≈⟨ ⨾-assocL ⟨ ≈-trans ⟩ ⨾-cong ∇-⨾-swap swap-⨾-⊗ ⟩
      ∇ {A} ⨾ (Id {A} ⊗m ! {A}) ⨾ swap
    ≈⟨ ⨾-assocL ⟨ ≈-trans ⟩ ⨾-cong₁ ∇-rightInv ⟩
```

```
      ⊗-rightUnit⁻¹ ⨾ swap
    ≈⟨ ⊗-rightUnit⁻¹-swap ⟩
      ⊗-leftUnit⁻¹
    □
  open MonCatG  monCatG public
  open MonCatS  monCatS public


record GSMonoidalCategory {i : Level} (j k : Level) (Obj : Set i) : Set (i ⊍ ℓsuc j ⊍ ℓsuc k) where
  field monCat     : MonoidalCategory j k Obj
        monCatSym  : MonCatSym monCat
        monCatGS   : MonCatGS monCat monCatSym
  open MonCatSym monCatSym public
  open MonCatGS  monCatGS  public
```

How useful this particular choice of re-exports is remains to be seen; so this may change in the future.

7.6 Categoric.MonoidalCategory.Coproducts

Every category that has finite sums is a MonoidalCategory:

```
CoproductMonCat : {i j k : Level} {Obj : Set i}
  → (C : Category j k Obj)
  → CatFinColimits.HasCoproducts C
  → CatFinColimits.HasInitialObject C
  → MonoidalCategory j k Obj
CoproductMonCat {Obj = Obj} C hasCoproducts hasInit = let
    open Category C using (semigroupoid; idOp)
    open CatFinColimits C using
      (module HasCoproducts; module HasInitialObject; module HasFiniteCoproducts)
    open HasInitialObject hasInit using (𝟙)
    open HasCoproducts hasCoproducts using
      (⊞-assoc; ⊞-assocL; ⨾-⊞-assoc; ⊞-assoc⨾⊞-assocL; ⊞-assocL⨾⊞-assoc
      ; ⊞-assoc-pentagon
      )
    open HasFiniteCoproducts hasCoproducts hasInit using
      (⊞-leftUnit; ⊞-leftUnit⁻¹; ⊞-leftUnit-naturality
      ; ⊞-leftUnit-leftUnit⁻¹; ⊞-leftUnit⁻¹-leftUnit
      ; ⊞-rightUnit; ⊞-rightUnit⁻¹; ⊞-rightUnit-naturality
      ; ⊞-rightUnit-rightUnit⁻¹; ⊞-rightUnit⁻¹-rightUnit
      ; ⊞-leftUnit-𝟙; ⊞-triangle)
  in record
    {category = C
    ; 𝟙 = 𝟙
    ; ⊗ = CoproductBifunctor C hasCoproducts
    ; ⊗-Assoc = record
      {indmor              = ⊞-assoc
      ; naturality          = ⨾-⊞-assoc
      ; indmor⁻¹            = ⊞-assocL
      ; indmor-⨾-indmor⁻¹  = ⊞-assoc⨾⊞-assocL
      ; indmor⁻¹-⨾-indmor  = ⊞-assocL⨾⊞-assoc
      }
    ; ⊗-assoc-pentagon    = ⊞-assoc-pentagon
    ; ⊗-LeftUnit = record
      {indmor              = ⊞-leftUnit
      ; naturality          = ⊞-leftUnit-naturality
      ; indmor⁻¹            = ⊞-leftUnit⁻¹
```

```
        ; indmor-⨾-indmor⁻¹  = ⊞-leftUnit-leftUnit⁻¹
        ; indmor⁻¹-⨾-indmor  = ⊞-leftUnit⁻¹-leftUnit
        }
      ; ⊗-RightUnit = record
        {indmor              = ⊞-rightUnit
        ; naturality         = ⊞-rightUnit-naturality
        ; indmor⁻¹           = ⊞-rightUnit⁻¹
        ; indmor-⨾-indmor⁻¹  = ⊞-rightUnit-rightUnit⁻¹
        ; indmor⁻¹-⨾-indmor  = ⊞-rightUnit⁻¹-rightUnit
        }
      ; ⊗-leftUnit-①        = ⊞-leftUnit-①
      ; ⊗-triangle           = ⊞-triangle
      }


CoproductMonCatSym : {i j k : Level} {Obj : Set i}
   → (C : Category j k Obj)
   → (hasCoproducts : CatFinColimits.HasCoproducts C)
   → (hasInit : CatFinColimits.HasInitialObject C)
   → MonCatSym (CoproductMonCat C hasCoproducts hasInit)
CoproductMonCatSym {Obj = Obj} C hasCoproducts hasInit = let
      open Category C using (semigroupoid; idOp)
      open CatFinColimits C using
        (module HasCoproducts; module HasFiniteCoproducts)
      open HasCoproducts hasCoproducts using (⊞-swap; ⊕-⨾-⊞-swap; ⊞-swap²; ⊞-swap-monoidal)
      open HasFiniteCoproducts hasCoproducts hasInit using (⊞-swapInit²≈Id; ⊞-swap-leftUnit)
   in record
      {swap              = ⊞-swap
      ; swap-natural     = ⊕-⨾-⊞-swap
      ; swap-cancel      = ⊞-swap²
      ; swap-unit        = ⊞-swapInit²≈Id
      ; swap-monoidal    = ⊞-swap-monoidal
      ; swap-⊗-leftUnit  = ⊞-swap-leftUnit
      }


module SMC-Props {i j k : Level} {Obj : Set i}
   (C : Category j k Obj)
   (hasCoproducts : CatFinColimits.HasCoproducts C)
   (hasInit : CatFinColimits.HasInitialObject C)
   where
   open Category C using
      (semigroupoid; idOp; Mor; _⨾_; _≈_; Id
      ; ≈-begin_; _≈⟨_⟩_; _□
      ; ≈-refl; ⨾-assoc; _⟨≈≈⟩_; ⨾-cong₂; leftId; ⨾-cong₁₂)
   open CatFinColimits C using (module HasCoproducts)
   open HasCoproducts hasCoproducts using
      (_⚠_; _⊕_; ⊞-assocL; ⊞-assoc; ⊞-assocL-⚠⚠; ⊕-⨾-⚠
      ; ⚠-cong; ⚠-cong₂; ⊞-assoc-⚠⚠; ⊞-swap-⨾-⚠; ⊞-transpose₂
      ; ⊞-swap; ⊕-cong₂; ι; κ; _⊕Id; ⚠-⨾; κ⨾⊕Id; ι⨾⊕Id)
   open MonCatSym (CoproductMonCatSym C hasCoproducts hasInit) public
      using (swap)
      renaming (⊗-transpose₂ to ⊕-transpose₂; ⊗-transpose₂² to ⊕-transpose₂²
        ; ⨾-⊗-transpose₂ to ⨾-⊕-transpose₂)


   ⊕-transpose₂-⨾ : {A B C D E : Obj} {F : Mor A E} {G : Mor B E} {H : Mor C E} {K : Mor D E}
                   → ⊕-transpose₂ ⨾ ((F ⚠ G) ⚠ (H ⚠ K)) ≈ (F ⚠ H) ⚠ (G ⚠ K)
   ⊕-transpose₂-⨾ {F = F} {G} {H} {K} = let
```

```
      swap⊕Id = swap ⊕ Id
      asL-swap⊕Id = ⊞-assocL ⨾ swap⊕Id
      asL-swap⊕Id-as = asL-swap⊕Id ⨾ ⊞-assoc
    in ≈-begin
        ⊕-transpose₂ ⨾ ((F ⨹ G) ⨹ (H ⨹ K))
      ≈⟨ ≈-refl ⟩
        ((⊞-assoc ⨾ (Id ⊕ asL-swap⊕Id-as)) ⨾ ⊞-assocL) ⨾ ((F ⨹ G) ⨹ (H ⨹ K))
      ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⊞-assocL-⨹⨹ ⟩
        (⊞-assoc ⨾ (Id ⊕ asL-swap⊕Id-as)) ⨾ (F ⨹ (G ⨹ (H ⨹ K)))
      ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⊕-⨾-⨹ ⟩
        ⊞-assoc ⨾ (Id ⨾ F ⨹ asL-swap⊕Id-as ⨾ (G ⨹ (H ⨹ K)))
      ≈⟨ ⨾-cong₂ (⨹-cong leftId ⨾-assoc) ⟩
        ⊞-assoc ⨾ (F ⨹ asL-swap⊕Id ⨾ (⊞-assoc ⨾ (G ⨹ (H ⨹ K))))
      ≈⟨ ⨾-cong₂ (⨹-cong₂ (⨾-cong₂ ⊞-assoc-⨹⨹ ⟨≈≈⟩ ⨾-assoc)) ⟩
        ⊞-assoc ⨾ (F ⨹ ⊞-assocL ⨾ (swap⊕Id ⨾ ((G ⨹ H) ⨹ K)))
      ≈⟨ ⨾-cong₂ (⨹-cong₂ (⨾-cong₂ ⊕-⨾-⨹)) ⟩
        ⊞-assoc ⨾ (F ⨹ ⊞-assocL ⨾ (swap ⨾ (G ⨹ H) ⨹ Id ⨾ K))
      ≈⟨ ⨾-cong₂ (⨹-cong₂ (⨾-cong₂ (⨹-cong ⊞-swap-⨾-⨹ leftId))) ⟩
        ⊞-assoc ⨾ (F ⨹ ⊞-assocL ⨾ ((H ⨹ G) ⨹ K))
      ≈⟨ ⨾-cong₂ (⨹-cong₂ ⊞-assocL-⨹⨹) ⟩
        ⊞-assoc ⨾ (F ⨹ (H ⨹ (G ⨹ K)))
      ≈⟨ ⊞-assoc-⨹⨹ ⟩
        (F ⨹ H) ⨹ (G ⨹ K)
      □


  ⊕-transpose₂-NF : {A B C D : Obj} → ⊕-transpose₂ ≈ ⊞-transpose₂ {A} {B} {C} {D}
  ⊕-transpose₂-NF = ≈-begin
        ⊕-transpose₂
      ≈⟨ ≈-refl ⟩
        (⊞-assoc ⨾ (Id ⊕ ((⊞-assocL ⨾ (⊞-swap ⊕ Id)) ⨾ ⊞-assoc))) ⨾ ⊞-assocL
      ≈⟨ ⨾-cong₁₂ (⊕-cong₂ ⨾-assoc) ⟩
        (⊞-assoc ⨾ (Id ⊕ (⊞-assocL ⨾ ((⊞-swap ⊕ Id) ⨾ ⊞-assoc)))) ⨾ ⊞-assocL
      ≈⟨ ⨾-cong₁₂ (⊕-cong₂ (⨾-cong₂ ⊕-⨾-⨹)) ⟩
        (⊞-assoc ⨾ (Id ⊕ (⊞-assocL ⨾ (⊞-swap ⨾ (ι ⨹ ι ⨾ κ) ⨹ Id ⨾ (κ ⨾ κ))))) ⨾ ⊞-assocL
      ≈⟨ ⨾-cong₁₂ (⊕-cong₂ (⨾-cong₂ (⨹-cong ⊞-swap-⨾-⨹ leftId))) ⟩
        (⊞-assoc ⨾ (Id ⊕ (⊞-assocL ⨾ ((ι ⨾ κ ⨹ ι) ⨹ κ ⨾ κ)))) ⨾ ⊞-assocL
      ≈⟨ ⨾-cong₁₂ (⊕-cong₂ ⊞-assocL-⨹⨹) ⟩
        (⊞-assoc ⨾ (Id ⊕ (ι ⨾ κ ⨹ (ι ⨹ κ ⨾ κ)))) ⨾ ⊞-assocL
      ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⊕-⨾-⨹ ⟩
        ⊞-assoc ⨾ (Id ⨾ (ι ⨾ ι) ⨹ (ι ⨾ κ ⨹ (ι ⨹ κ ⨾ κ)) ⨾ (κ ⊕Id))
      ≈⟨ ⨾-cong₂ (⨹-cong leftId ⨹-⨾) ⟩
        ⊞-assoc ⨾ (ι ⨾ ι ⨹ ((ι ⨾ κ) ⨾ (κ ⊕Id) ⨹ (ι ⨹ κ ⨾ κ) ⨾ (κ ⊕Id)))
      ≈⟨ ⨾-cong₂ (⨹-cong₂ (⨹-cong (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ κ⨾⊕Id) ⨹-⨾)) ⟩
        ⊞-assoc ⨾ (ι ⨾ ι ⨹ (ι ⨾ κ ⨹ (ι ⨾ (κ ⊕Id) ⨹ (κ ⨾ κ) ⨾ (κ ⊕Id))))
      ≈⟨ ⨾-cong₂ (⨹-cong₂ (⨹-cong₂ (⨹-cong ι⨾⊕Id (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ κ⨾⊕Id)))) ⟩
        ⊞-assoc ⨾ (ι ⨾ ι ⨹ (ι ⨾ κ ⨹ (κ ⨾ ι ⨹ κ ⨾ κ)))
      ≈⟨ ⊞-assoc-⨹⨹ ⟩
        (ι ⊕ ι) ⨹ (κ ⊕ κ)
      ≈⟨ ≈-refl ⟩
        ⊞-transpose₂
      □
```

7.7 Categoric.MonoidalCategory.Products

```
ProductMonCat : {i j k : Level} {Obj : Set i}
  → (C : Category j k Obj)
```

```
  → CatFinLimits.HasProducts C
  → CatFinLimits.HasTerminalObject C
  → MonoidalCategory j k Obj
ProductMonCat {i} {j} {k} {Obj = Obj} C hasProducts hasTerm = let
    open CatFinLimits C using
      (module HasProducts; module HasTerminalObject; module HasFiniteProducts)
    open HasProducts hasProducts using
      (⊠-assoc; ⨾-⊠-assoc
      ; ⊠-assocL; ⊠-assoc⨾⊠-assocL; ⊠-assocL⨾⊠-assoc; ⊠-assoc-pentagon)
    open HasTerminalObject hasTerm using (Ⓣ)
    open HasFiniteProducts hasProducts hasTerm using
      (⊠-leftUnit ; ⊠-leftUnit⁻¹; ⊠-leftUnit-leftUnit⁻¹; ⊠-leftUnit⁻¹-leftUnit
      ; ⊠-rightUnit; ⊠-rightUnit⁻¹; ⊠-rightUnit-rightUnit⁻¹; ⊠-rightUnit⁻¹-rightUnit
      ; ⊠-leftUnit-naturality; ⊠-rightUnit-naturality; ⊠-leftUnit-Ⓣ; ⊠-triangle)
  in record
    {category = C
    ; ① = Ⓣ
    ; ⊗ = ProductBifunctor C hasProducts
    ; ⊗-Assoc = record
      {indmor = ⊠-assoc
      ; naturality = ⨾-⊠-assoc
      ; indmor⁻¹ = ⊠-assocL
      ; indmor-⨾-indmor⁻¹ = ⊠-assoc⨾⊠-assocL
      ; indmor⁻¹-⨾-indmor = ⊠-assocL⨾⊠-assoc
      }
    ; ⊗-assoc-pentagon = ⊠-assoc-pentagon
    ; ⊗-LeftUnit = record
      {indmor            = ⊠-leftUnit
      ; naturality        = λ {A} {B} {F} → ⊠-leftUnit-naturality {B} {A} {F}
      ; indmor⁻¹          = ⊠-leftUnit⁻¹
      ; indmor-⨾-indmor⁻¹ = ⊠-leftUnit-leftUnit⁻¹
      ; indmor⁻¹-⨾-indmor = ⊠-leftUnit⁻¹-leftUnit
      }
    ; ⊗-RightUnit = record
      {indmor            = ⊠-rightUnit
      ; naturality        = λ {A} {B} → ⊠-rightUnit-naturality {B} {A}
      ; indmor⁻¹          = ⊠-rightUnit⁻¹
      ; indmor-⨾-indmor⁻¹ = ⊠-rightUnit-rightUnit⁻¹
      ; indmor⁻¹-⨾-indmor = ⊠-rightUnit⁻¹-rightUnit
      }
    ; ⊗-leftUnit-① = ⊠-leftUnit-Ⓣ
    ; ⊗-triangle = ⊠-triangle
    }

ProductMonCatSym : {i j k : Level} {Obj : Set i}
  → (C : Category j k Obj)
  → (hasProducts : CatFinLimits.HasProducts C)
  → (hasTerm : CatFinLimits.HasTerminalObject C)
  → MonCatSym (ProductMonCat C hasProducts hasTerm)
ProductMonCatSym {Obj = Obj} C hasProducts hasTerm = let
    open CatFinLimits C using (module HasProducts; module HasFiniteProducts)
    open HasProducts hasProducts using (⊠-swap; ⊗-⨾-⊠-swap; ⊠-swap²; ⊠-swap-monoidal)
    open HasFiniteProducts hasProducts hasTerm using (⊠-swapInit²≈Id; ⊠-swap-leftUnit)
  in record
    {swap          = ⊠-swap
    ; swap-natural = ⊗-⨾-⊠-swap
    ; swap-cancel  = ⊠-swap²
    ; swap-unit    = ⊠-swapInit²≈Id
```

```
    ; swap-monoidal   = ⊠-swap-monoidal
    ; swap-⊗-leftUnit = ⊠-swap-leftUnit
    }
```

7.8 Categoric.MonoidalCategory.ProductGS

```
ProductMonCatG : {i j k : Level} {Obj : Set i}
  → (C : Category j k Obj)
  → (hasProducts : CatFinLimits.HasProducts C)
  → (hasTerm : CatFinLimits.HasTerminalObject C)
  → MonCatG (ProductMonCat C hasProducts hasTerm)
ProductMonCatG C hasProducts hasTerm = let
    open Category C using (≈-sym)
    open CatFinLimits C using (module HasFiniteProducts; module HasTerminalObject)
    open HasTerminalObject hasTerm using (ⓣ; ≈ⓣ)
    open HasFiniteProducts hasProducts hasTerm using (ⓣ≈ⓣ⊗ⓣ⨾LU)
  in record
    {!           =      ⓣ
    ; !-unit     =      ≈-sym ≈ⓣ
    ; !-monoidal =  ⓣ≈ⓣ⊗ⓣ⨾LU
    }


ProductMonCatS : {i j k : Level} {Obj : Set i}
  → (C : Category j k Obj)
  → (hasProducts : CatFinLimits.HasProducts C)
  → (hasTerm : CatFinLimits.HasTerminalObject C)
  → MonCatS (ProductMonCat C hasProducts hasTerm) (ProductMonCatSym C hasProducts hasTerm)
ProductMonCatS C hasProducts hasTerm = let
    open Category C using (Id)
    open CatFinLimits C using (module HasProducts; module HasFiniteProducts)
    open HasProducts hasProducts using (_▽_; I▽I-assoc; ▽-⨾-⊠-swap; I▽I-monoidal)
    open HasFiniteProducts hasProducts hasTerm using (Ⓣ ▽Ⓣ ≈LU⁻¹)
  in record
    {▽            = Id ▽ Id
    ; ▽-unit      = Ⓣ ▽Ⓣ ≈LU⁻¹
    ; ▽-assoc     = I▽I-assoc
    ; ▽-⨾-swap    = ▽-⨾-⊠-swap
    ; ▽-monoidal  = I▽I-monoidal
    }


ProductMonCatGS : {i j k : Level} {Obj : Set i}
  → (C : Category j k Obj)
  → (hasProducts : CatFinLimits.HasProducts C)
  → (hasTerm : CatFinLimits.HasTerminalObject C)
  → MonCatGS (ProductMonCat C hasProducts hasTerm) (ProductMonCatSym C hasProducts hasTerm)
ProductMonCatGS C hasProducts hasTerm = let
    open CatFinLimits.HasFiniteProducts C hasProducts hasTerm using (I▽I-⨾-Id⊗ⓣ)
  in record
    {monCatG    = ProductMonCatG C hasProducts hasTerm
    ; monCatS   = ProductMonCatS C hasProducts hasTerm
    ; ▽-rightInv = I▽I-⨾-Id⊗ⓣ
    }


ProductGSMonCat : {i j k : Level} {Obj : Set i}
  → (C : Category j k Obj)
```

```
  → (hasProducts : CatFinLimits.HasProducts C)
  → (hasTerm : CatFinLimits.HasTerminalObject C)
  → GSMonoidalCategory j k Obj
ProductGSMonCat C hasProducts hasTerm = record
    {monCat     = ProductMonCat     C hasProducts hasTerm
    ; monCatSym = ProductMonCatSym C hasProducts hasTerm
    ; monCatGS  = ProductMonCatGS  C hasProducts hasTerm
    }
```

Part II

Categoric Abstractions of Relational Algebras

Chapter 8

Posets and Lattices

8.1 Relation.Binary.Poset.Renamed

The Posets defined in the standard library module Relation.Binary are Setoids, with equivalence relation ≈, with an additional compatible ordering relation ≤. For convenience, we rename the properties of these two relations so that the names refer to the relations, and bring them all into a single scope.

```
module Poset′ {j  k₁ k₂ : Level} (poset : Poset j k₁ k₂) where
  open Poset poset public renaming
    (antisym  to ≤-antisym
    ; refl      to ≤-refl
    ; reflexive to ≤-reflexive
    ; trans     to ≤-trans
    )
  open IsEquivalence isEquivalence public renaming
    (refl      to ≈-refl
    ; sym       to ≈-sym
    ; trans     to ≈-trans
    ; reflexive to ≈-reflexive
    )
```

We also add some derived properties that will be used to abbreviate many proofs.

```
  ≤-reflexive′ : {R S : Carrier} → R ≈ S → S ≤ R
  ≤-reflexive′ eq = ≤-reflexive (≈-sym eq)

  ≤-trans₁ : {Q R S : Carrier} → Q ≤ R → R ≈ S → Q ≤ S
  ≤-trans₁ leq eq = ≤-trans leq (≤-reflexive eq)

  ≤-trans₂ : {Q R S : Carrier} → Q ≈ R → R ≤ S → Q ≤ S
  ≤-trans₂ eq leq = ≤-trans (≤-reflexive eq) leq

  infixl 1 _⟨≈≈⟩_ _⟨≈≈˘⟩_ _⟨≈˘≈⟩_ _⟨≈˘≈˘⟩_ _⟨≤≤⟩_ _⟨≤≈⟩_ _⟨≤≈˘⟩_ _⟨≈≤⟩_ _⟨≈˘≤⟩_
  _⟨≈≈⟩_ : {Q R S : Carrier} → Q ≈ R → R ≈ S → Q ≈ S
  _⟨≈≈⟩_ = ≈-trans

  _⟨≈≈˘⟩_ : {Q R S : Carrier} → Q ≈ R → S ≈ R → Q ≈ S
  _⟨≈≈˘⟩_ x y = ≈-trans x (≈-sym y)

  _⟨≈˘≈⟩_ : {Q R S : Carrier} → R ≈ Q → R ≈ S → Q ≈ S
  _⟨≈˘≈⟩_ x y = ≈-trans (≈-sym x) y

  _⟨≈˘≈˘⟩_ : {Q R S : Carrier} → R ≈ Q → S ≈ R → Q ≈ S
  _⟨≈˘≈˘⟩_ x y = ≈-trans (≈-sym x) (≈-sym y)

  _⟨≤≤⟩_ : {Q R S : Carrier} → Q ≤ R → R ≤ S → Q ≤ S
  _⟨≤≤⟩_ = ≤-trans

  _⟨≤≈⟩_ : {Q R S : Carrier} → Q ≤ R → R ≈ S → Q ≤ S
  _⟨≤≈⟩_ = ≤-trans₁
```

```
  _⟨≤≈˘⟩_ : {Q R S : Carrier} → Q ≤ R → S ≈ R → Q ≤ S
  _⟨≤≈˘⟩_ x y = ≤-trans₁ x (≈-sym y)
  _⟨≈≤⟩_ : {Q R S : Carrier} → Q ≈ R → R ≤ S → Q ≤ S
  _⟨≈≤⟩_ = ≤-trans₂
  _⟨≈˘≤⟩_ : {Q R S : Carrier} → R ≈ Q → R ≤ S → Q ≤ S
  _⟨≈˘≤⟩_ x = ≤-trans₂ (≈-sym x)
```

We also add a convenient alias for Poset.Carrier, while avoiding a name clash with Relation.Binary.Setoid.Utils.⌊_⌋.

```
⌊_≤⌋ : {i j k : Level} → Poset i j k → Set i
⌊_≤⌋ = Poset.Carrier
```

The following renamings do not re-export the Setoid material since in OrderedSemigroupoid, that is obtained separately — this may be organised differently in the future.

```
module Poset-round {j k₁ k₂ : Level} (poset : Poset j k₁ k₂) where
  open Poset′ poset public using () renaming
    (_≤_         to _⊆_          -- : Rel Carrier k₂
    ; ≤-antisym  to ⊆-antisym    -- : {R S : Carrier} → R ⊆ S → S ⊆ R → R ≈ S
    ; ≤-refl     to ⊆-refl       -- : {R : Carrier} → R ⊆ R
    ; ≤-reflexive to ⊆-reflexive -- : {R S : Carrier} → R ≈ S → R ⊆ S
    ; ≤-trans    to ⊆-trans      -- : {Q R S : Carrier} → Q ⊆ R → R ⊆ S → Q ⊆ S
    ; ≤-reflexive′ to ⊆-reflexive′ -- : {R S : Carrier} → R ≈ S → S ⊆ R
    ; ≤-trans₁   to ⊆-trans₁     -- : {Q R S : Carrier} → Q ⊆ R → R ≈ S → Q ⊆ S
    ; ≤-trans₂   to ⊆-trans₂     -- : {Q R S : Carrier} → Q ≈ R → R ⊆ S → Q ⊆ S
    ; _⟨≤≤⟩_     to _⟨⊆⊆⟩_       -- : {Q R S : Carrier} → Q ⊆ R → R ⊆ S → Q ⊆ S
    ; _⟨≤≈⟩_     to _⟨⊆≈⟩_       -- : {Q R S : Carrier} → Q ⊆ R → R ≈ S → Q ⊆ S
    ; _⟨≤≈˘⟩_    to _⟨⊆≈˘⟩_      -- : {Q R S : Carrier} → Q ⊆ R → S ≈ R → Q ⊆ S
    ; _⟨≈≤⟩_     to _⟨≈⊆⟩_       -- : {Q R S : Carrier} → Q ≈ R → R ⊆ S → Q ⊆ S
    ; _⟨≈˘≤⟩_    to _⟨≈˘⊆⟩_      -- : {Q R S : Carrier} → R ≈ Q → R ⊆ S → Q ⊆ S
    )

module Poset-square {j k₁ k₂ : Level} (poset : Poset j k₁ k₂) where
  open Poset′ poset public using () renaming
    (_≤_         to _⊑_          -- : Rel Carrier k₂
    ; ≤-antisym  to ⊑-antisym    -- : {R S : Carrier} → R ⊑ S → S ⊑ R → R ≈ S
    ; ≤-refl     to ⊑-refl       -- : {R : Carrier} → R ⊑ R
    ; ≤-reflexive to ⊑-reflexive -- : {R S : Carrier} → R ≈ S → R ⊑ S
    ; ≤-trans    to ⊑-trans      -- : {Q R S : Carrier} → Q ⊑ R → R ⊑ S → Q ⊑ S
    ; ≤-reflexive′ to ⊑-reflexive′ -- : {R S : Carrier} → R ≈ S → S ⊑ R
    ; ≤-trans₁   to ⊑-trans₁     -- : {Q R S : Carrier} → Q ⊑ R → R ≈ S → Q ⊑ S
    ; ≤-trans₂   to ⊑-trans₂     -- : {Q R S : Carrier} → Q ≈ R → R ⊑ S → Q ⊑ S
    ; _⟨≤≤⟩_     to _⟨⊑⊑⟩_       -- : {Q R S : Carrier} → Q ⊑ R → R ⊑ S → Q ⊑ S
    ; _⟨≤≈⟩_     to _⟨⊑≈⟩_       -- : {Q R S : Carrier} → Q ⊑ R → R ≈ S → Q ⊑ S
    ; _⟨≤≈˘⟩_    to _⟨⊑≈˘⟩_      -- : {Q R S : Carrier} → Q ⊑ R → S ≈ R → Q ⊑ S
    ; _⟨≈≤⟩_     to _⟨≈⊑⟩_       -- : {Q R S : Carrier} → Q ≈ R → R ⊑ S → Q ⊑ S
    ; _⟨≈˘≤⟩_    to _⟨≈˘⊑⟩_      -- : {Q R S : Carrier} → R ≈ Q → R ⊑ S → Q ⊑ S
    )
```

8.2 Relation.Binary.Poset.Dual

The module Relation.Binary.Poset.Dual provides dualisations for the Posets of the standard library module Relation.Binary, and for all relevant concepts used to define Posets. This is therefore one of the few places in the current work that are directly concerned with the internal structure of these standard library concepts.

```
dualIsPreorder : {a ℓ₁ ℓ₂ : Level} {A : Set a} {_≈_ : Rel A ℓ₁} {_≤_ : Rel A ℓ₂}
                 → IsPreorder _≈_ _≤_ → IsPreorder _≈_ (λ x y → y ≤ x)
dualIsPreorder base = let open IsPreorder base in record
  {isEquivalence = isEquivalence
  ; reflexive    = λ eq → reflexive (IsEquivalence.sym isEquivalence eq)
  ; trans        = λ xy yz → trans yz xy
  }
dualPreorder : {c ℓ₁ ℓ₂ : Level} → Preorder c ℓ₁ ℓ₂ → Preorder c ℓ₁ ℓ₂
dualPreorder base = let open Preorder base in record
  {Carrier    = Carrier
  ; _≈_       = _≈_
  ; _∼_       = λ x y → y ∼ x
  ; isPreorder = dualIsPreorder isPreorder
  }
dualIsPartialOrder : {a ℓ₁ ℓ₂ : Level} {A : Set a} {_≈_ : Rel A ℓ₁} {_≤_ : Rel A ℓ₂}
                     → IsPartialOrder _≈_ _≤_ → IsPartialOrder _≈_ (λ x y → y ≤ x)
dualIsPartialOrder isPO = let open IsPartialOrder isPO in record
  {isPreorder = dualIsPreorder isPreorder
  ; antisym   = λ xy yx → antisym yx xy
  }
dualPoset : {c ℓ₁ ℓ₂ : Level} → Poset c ℓ₁ ℓ₂ → Poset c ℓ₁ ℓ₂
dualPoset base = let open Poset base in record
  {Carrier         = Carrier
  ; _≈_            = _≈_
  ; _≤_            = λ x y → y ≤ x
  ; isPartialOrder = dualIsPartialOrder isPartialOrder
  }
```

8.3 Relation.Binary.Poset.Calc

We introduce a generalisation of the standard library's Relation.Binary.PreorderReasoning that works in the context of a given Poset both for equational reasoning in the underlying setoid and for preorder reasoning. As long as the poset relation levels k_1 and k_2 cannot be assumed to be equal, we cannot use a single begin to extract both kinds of proof, so we use separate versions for equality proofs and inclusion proofs. However, we can share the other symbols between the two kinds of proof by distinguishing them by a Boolean parameter.

```
module PosetCalc {j k₁ k₂ : Level} (P : Poset j k₁ k₂) where
  open Poset′ P renaming (Carrier to S)
  infix 2 _□
  infixr 2 _≤⟨_⟩_ _≈⟨_⟩_ _≈˘⟨_⟩_ _≈≡⟨_⟩_ _≈≡˘⟨_⟩_
  infix 1 ≈-begin_ ≤-begin_
  private
    data IsRelatedTo (x y : S) : Bool → Set (k₁ ⊍ k₂) where
      eqTo : (x≈y : x ≈ y) → IsRelatedTo x y true
      leqTo : (x≤y : x ≤ y) → IsRelatedTo x y false
  ≈-begin_ : {x y : S} → IsRelatedTo x y true → x ≈ y
  ≈-begin (eqTo x≈y) = x≈y
  ≤-begin_ : {b : Bool} {x y : S} → IsRelatedTo x y b → x ≤ y
  ≤-begin (eqTo x≈y) = ≤-reflexive x≈y
  ≤-begin (leqTo x≤y) = x≤y
  _≤⟨_⟩_ : {b : Bool} (x : S) {y z : S} → x ≤ y → IsRelatedTo y z b → IsRelatedTo x z false
  _ ≤⟨ x≤y ⟩ eqTo y≈z = leqTo (≤-trans₁ x≤y y≈z)
  _ ≤⟨ x≤y ⟩ leqTo y≤z = leqTo (≤-trans x≤y y≤z)
  _≈⟨_⟩_ : {b : Bool} (x : S) {y z : S} → x ≈ y → IsRelatedTo y z b → IsRelatedTo x z b
```

```
  _ ≈⟨ x≈y ⟩ eqTo y≈z  =  eqTo (≈-trans x≈y y≈z)
  _ ≈⟨ x≈y ⟩ leqTo y≤z  =  leqTo (≤-trans₂ x≈y y≤z)
  _≈˘⟨_⟩_ : {b : Bool} (x : S) {y z : S} → y ≈ x → IsRelatedTo y z b → IsRelatedTo x z b
  x ≈˘⟨ y≈x ⟩ relTo  =  x ≈⟨ ≈-sym y≈x ⟩ relTo
  _≈≡⟨_⟩_ : {b : Bool} (x : S) {y z : S} → x ≡ y → IsRelatedTo y z b → IsRelatedTo x z b
  x ≈≡⟨ x≡y ⟩ relTo  =  x ≈⟨ ≈-reflexive x≡y ⟩ relTo
  _≈≡˘⟨_⟩_ : {b : Bool} (x : S) {y z : S} → y ≡ x → IsRelatedTo y z b → IsRelatedTo x z b
  x ≈≡˘⟨ y≡x ⟩ relTo  =  x ≈≡⟨ ≡-sym y≡x ⟩ relTo
  _□ : (x : S) → IsRelatedTo x x true
  _□ _  =  eqTo ≈-refl


module PosetCalc-round {j k₁ k₂ : Level} (P : Poset j k₁ k₂) where
  open PosetCalc P public renaming (≤-begin_ to ⊆-begin_; _≤⟨_⟩_ to _⊆⟨_⟩_)


module PosetCalc-square {j k₁ k₂ : Level} (P : Poset j k₁ k₂) where
  open PosetCalc P public renaming (≤-begin_ to ⊑-begin_; _≤⟨_⟩_ to _⊑⟨_⟩_)
```

8.4 Relation.Binary.Poset

The module Relation.Binary.Poset provides auxiliary poset-related material.

```
posetBy : {p₁ p₂ p₃ : Level} (P : Poset p₁ p₂ p₃)
        → {s : Level} {S : Set s} (value : S → Poset.Carrier P) → Poset s p₂ p₃
posetBy P {S = S} value  =  let open Poset′ P in record
  {Carrier  =  S
  ; _≈_    =  λ b c → value b ≈ value c
  ; _≤_    =  λ b c → value b ≤ value c
  ; isPartialOrder  =  record
    {isPreorder  =  record
      {isEquivalence  =  record
        {refl   =  λ {b} → ≈-refl
        ; sym   =  ≈-sym
        ; trans  =  ≈-trans
        }
      ; reflexive       =  ≤-reflexive
      ; trans           =  ≤-trans
      }
    ; antisym  =  ≤-antisym
    }
  }


module LowerBounds {p₁ p₂ p₃ : Level} (P : Poset p₁ p₂ p₃) where
  open Poset′ P
```

The greatest element of a set is the greatest lower bound of the empty subset of that set, which justifies the choice to include isGreatestElem here, and not isLeastElem. This choice also has as a consequence that, when defining meets as greatest lower bounds in Sect. 8.5, we will only need material from the current module LowerBounds, and nothing from the dual module UpperBounds.

```
  isGreatestElem : Carrier → Set (p₁ ⊍ p₃)
  isGreatestElem t  =  (x : Carrier) → x ≤ t
```

Due to their importance for the definition of lattices, we define specialised concepts for binary lower bounds:

```
record IsLowerBound₂ (x y b : Carrier) : Set (p₁ ⊍ p₃) where
  field
    bound₁ : b ≤ x
    bound₂ : b ≤ y

record LowerBound₂ (x y : Carrier) : Set (p₁ ⊍ p₃) where
  field
    value : Carrier
    proof : IsLowerBound₂ x y value
  open IsLowerBound₂ proof public

LowerBound₂Poset : (x y : Carrier) → Poset (p₁ ⊍ p₃) p₂ p₃
LowerBound₂Poset x y = posetBy P {S = LowerBound₂ x y} LowerBound₂.value
```

Analogously, we define lower bounds of indexed sets:

```
isLowerBoundI : {i : Level} {I : Set i} (f : I → Carrier) (b : Carrier) → Set (i ⊍ p₃)
isLowerBoundI {_} {I} f b = (x : I) → b ≤ f x

record LowerBoundI {i : Level} {I : Set i} (f : I → Carrier) : Set (i ⊍ p₁ ⊍ p₃) where
  field
    value : Carrier
    proof : isLowerBoundI f value

LowerBoundIPoset : {i : Level} {I : Set i} (f : I → Carrier) → Poset (i ⊍ p₁ ⊍ p₃) p₂ p₃
LowerBoundIPoset f = posetBy P {S = LowerBoundI f} LowerBoundI.value
```

```
module UpperBounds {p₁ p₂ p₃ : Level} (P : Poset p₁ p₂ p₃) where
  open LowerBounds (dualPoset P) public renaming
    (isGreatestElem          to isLeastElem
    ; IsLowerBound₂          to IsUpperBound₂
    ; module IsLowerBound₂   to IsUpperBound₂
    ; LowerBound₂            to UpperBound₂
    ; module LowerBound₂     to UpperBound₂
    ; LowerBound₂Poset       to UpperBound₂Poset
    ; isLowerBoundI          to isUpperBoundI
    ; LowerBoundI            to UpperBoundI
    ; module LowerBoundI     to UpperBoundI
    ; LowerBoundIPoset       to UpperBoundIPoset
    )
```

Module Bounds re-exports both LowerBounds and UpperBounds; if a renaming were missing in the latter, that would lead to a duplicate definition error here.

```
module Bounds {p₁ p₂ p₃ : Level} (P : Poset p₁ p₂ p₃) where
  open LowerBounds P public
  open UpperBounds P public
```

8.5 Relation.Binary.Poset.Lattice

We introduce everything related with meets (greatest lower bounds) inside a separate module, which we will dualise below for joins (least upper bounds).

```
module PosetMeet {j k₁ k₂ : Level} (𝒫 : Poset j k₁ k₂) where
  open Poset′ 𝒫
```

The field names of the two records IsMeet and Meet have been chosen so that they still make sense after renaming of the records to IsJoin and Join. (Field names apparently cannot be renamed.) We formalise meets in a context where they do not need to exist; a value of type Meet R S documents (constructively) that R and S have a meet.

```
record IsMeet (R S M : Carrier) : Set (j ⊍ k₂) where
  field
    bound₁ : M ≤ R
    bound₂ : M ≤ S
    universal : {X : Carrier} → X ≤ R → X ≤ S → X ≤ M
record Meet (R S : Carrier) : Set (j ⊍ k₂) where
  field
    value : Carrier
    proof : IsMeet R S value
  open IsMeet proof public
```

Meets are greatest lower bounds:

```
open LowerBounds
isMeet-IsLowerBound₂ : {R S M : Carrier} → IsMeet R S M → IsLowerBound₂ 𝒫 R S M
isMeet-IsLowerBound₂ m = let open IsMeet m in record
  {bound₁ = bound₁
  ;bound₂ = bound₂
  }
isMeet-LowerBound₂ : {R S M : Carrier} → IsMeet R S M → LowerBound₂ 𝒫 R S
isMeet-LowerBound₂ {M = M} m = record {value = M; proof = isMeet-IsLowerBound₂ m}
isMeet-isGLB : {R S M : Carrier} → (m : IsMeet R S M)
             → isGreatestElem (LowerBound₂Poset 𝒫 R S) (isMeet-LowerBound₂ m)
isMeet-isGLB m = λ x → let open LowerBound₂ 𝒫 x in IsMeet.universal m bound₁ bound₂
isGLB-isMeet : {R S : Carrier} {m : LowerBound₂ 𝒫 R S}
             → isGreatestElem (LowerBound₂Poset 𝒫 R S) m
             → IsMeet R S (LowerBound₂.value m)
isGLB-isMeet {R} {S} {m} g = record
  {bound₁ = LowerBound₂.bound₁ m
  ;bound₂ = LowerBound₂.bound₂ m
  ;universal = λ {X} X≤R X≤S → let p = record {bound₁ = X≤R; bound₂ = X≤S}
                                in g (record {value = X; proof = p})
  }
```

We derive some standard properties of meets, first idempotency:

```
IsMeet-idempotent : {R M : Carrier} → IsMeet R R M → M ≈ R
IsMeet-idempotent m = ≤-antisym (IsMeet.bound₁ m) (IsMeet.universal m ≤-refl ≤-refl)
idempotentMeet : {R : Carrier} → Meet R R
idempotentMeet {R} = record
  {value = R
  ;proof = record
      {bound₁ = ≤-refl
      ;bound₂ = ≤-refl
      ;universal = λ {_} X≤R _ → X≤R
      }
  }
```

Symmetry of meet:

```
IsMeet-sym : {R S M : Carrier} → IsMeet R S M → IsMeet S R M
IsMeet-sym m = record
      {bound₁ = IsMeet.bound₂ m
      ;bound₂ = IsMeet.bound₁ m
      ;universal = λ {_} X≤R X≤S → IsMeet.universal m X≤S X≤R
      }
commuteMeet : {R S : Carrier} → Meet R S → Meet S R
```

```
commuteMeet R⊓S = record
  {value = Meet.value R⊓S
  ; proof = IsMeet-sym (Meet.proof R⊓S)
  }
```

Monotonicity properties with respect to ≤:

```
IsMeet-monotone : {R₁ R₂ S₁ S₂ M₁ M₂ : Carrier}
                → IsMeet R₁ S₁ M₁ → IsMeet R₂ S₂ M₂ → R₁ ≤ R₂ → S₁ ≤ S₂ → M₁ ≤ M₂
IsMeet-monotone m₁ m₂ R₁≤R₂ S₁≤S₂ = IsMeet.universal m₂ (≤-trans (IsMeet.bound₁ m₁) R₁≤R₂)
                                                         (≤-trans (IsMeet.bound₂ m₁) S₁≤S₂)
Meet-monotone : {R₁ R₂ S₁ S₂ : Carrier} → (R₁⊓S₁ : Meet R₁ S₁) → (R₂⊓S₂ : Meet R₂ S₂)
              → R₁ ≤ R₂ → S₁ ≤ S₂ → Meet.value R₁⊓S₁ ≤ Meet.value R₂⊓S₂
Meet-monotone R₁⊓S₁ R₂⊓S₂ = IsMeet-monotone (Meet.proof R₁⊓S₁) (Meet.proof R₂⊓S₂)
Meet-monotone₁ : {R₁ R₂ S : Carrier} → (R₁⊓S : Meet R₁ S) → (R₂⊓S : Meet R₂ S)
               → R₁ ≤ R₂ → Meet.value R₁⊓S ≤ Meet.value R₂⊓S
Meet-monotone₁ R₁⊓S R₂⊓S R₁≤R₂ = Meet-monotone R₁⊓S R₂⊓S R₁≤R₂ ≤-refl
Meet-monotone₂ : {R S₁ S₂ : Carrier} → (R⊓S₁ : Meet R S₁) → (R⊓S₂ : Meet R S₂)
               → S₁ ≤ S₂ → Meet.value R⊓S₁ ≤ Meet.value R⊓S₂
Meet-monotone₂ R⊓S₁ R⊓S₂ S₁≤S₂ = Meet-monotone R⊓S₁ R⊓S₂ ≤-refl S₁≤S₂
```

Congruence properties with respect to ≈ follow from monotonicity:

```
IsMeet-cong : {R₁ R₂ S₁ S₂ M₁ M₂ : Carrier}
            → IsMeet R₁ S₁ M₁ → IsMeet R₂ S₂ M₂ → R₁ ≈ R₂ → S₁ ≈ S₂ → M₁ ≈ M₂
IsMeet-cong m₁ m₂ R₁≈R₂ S₁≈S₂ = ≤-antisym
  (IsMeet-monotone m₁ m₂ (≤-reflexive  R₁≈R₂) (≤-reflexive  S₁≈S₂))
  (IsMeet-monotone m₂ m₁ (≤-reflexive′ R₁≈R₂) (≤-reflexive′ S₁≈S₂))
Meet-cong : {R₁ R₂ S₁ S₂ : Carrier} → (R₁⊓S₁ : Meet R₁ S₁) → (R₂⊓S₂ : Meet R₂ S₂)
          → R₁ ≈ R₂ → S₁ ≈ S₂ → Meet.value R₁⊓S₁ ≈ Meet.value R₂⊓S₂
Meet-cong R₁⊓S₁ R₂⊓S₂ = IsMeet-cong (Meet.proof R₁⊓S₁) (Meet.proof R₂⊓S₂)
Meet-cong₁ : {R₁ R₂ S : Carrier} → (R₁⊓S : Meet R₁ S) → (R₂⊓S : Meet R₂ S)
           → R₁ ≈ R₂ → Meet.value R₁⊓S ≈ Meet.value R₂⊓S
Meet-cong₁ R₁⊓S R₂⊓S R₁≈R₂ = Meet-cong R₁⊓S R₂⊓S R₁≈R₂ ≈-refl
Meet-cong₂ : {R S₁ S₂ : Carrier} → (R⊓S₁ : Meet R S₁) → (R⊓S₂ : Meet R S₂)
           → S₁ ≈ S₂ → Meet.value R⊓S₁ ≈ Meet.value R⊓S₂
Meet-cong₂ R⊓S₁ R⊓S₂ S₁≈S₂ = Meet-cong R⊓S₁ R⊓S₂ ≈-refl S₁≈S₂
cong-IsMeet : {R₁ R₂ S₁ S₂ M : Carrier}
            → R₁ ≈ R₂ → S₁ ≈ S₂ → IsMeet R₁ S₁ M → IsMeet R₂ S₂ M
cong-IsMeet R₁≈R₂ S₁≈S₂ m = record
  {bound₁ = ≤-trans₁ (IsMeet.bound₁ m) R₁≈R₂
  ; bound₂ = ≤-trans₁ (IsMeet.bound₂ m) S₁≈S₂
  ; universal = λ {X} X≤R₂ X≤S₂ → IsMeet.universal m (≤-trans₁ X≤R₂ (≈-sym R₁≈R₂))
                                                    (≤-trans₁ X≤S₂ (≈-sym S₁≈S₂))
  }
```

The congruence properties allow us to show that meets are uniquely determined up to ≈:

```
IsMeet-unique : {R S M₁ M₂ : Carrier} → IsMeet R S M₁ → IsMeet R S M₂ → M₁ ≈ M₂
IsMeet-unique m₁ m₂ = IsMeet-cong m₁ m₂ ≈-refl ≈-refl
```

Where a meet is equivalent to one of the two arguments, these arguments are comparable:

```
≤-from-IsMeet₁ : {R S M : Carrier} → IsMeet R S M → M ≈ R → R ≤ S
≤-from-IsMeet₁ m eq = ≤-trans₂ (≈-sym eq) (IsMeet.bound₂ m)
≤-from-IsMeet₂ : {R S M : Carrier} → IsMeet R S M → M ≈ S → S ≤ R
≤-from-IsMeet₂ m eq = ≤-trans₂ (≈-sym eq) (IsMeet.bound₁ m)
```

```
≤-from-Meet₁ : {R S : Carrier} → (R⊓S : Meet R S) → Meet.value R⊓S ≈ R → R ≤ S
≤-from-Meet₁ R⊓S = ≤-from-IsMeet₁ (Meet.proof R⊓S)
≤-from-Meet₂ : {R S : Carrier} → (R⊓S : Meet R S) → Meet.value R⊓S ≈ S → S ≤ R
≤-from-Meet₂ R⊓S = ≤-from-IsMeet₂ (Meet.proof R⊓S)
```

Vice versa, comparable arguments contain their meet:

```
≤-to-IsMeet₁ : {R S : Carrier} → R ≤ S → IsMeet R S R
≤-to-IsMeet₁ leq = record
  {bound₁ = ≤-refl
  ;bound₂ = leq
  ;universal = λ {X} X≤R X≤S → X≤R
  }
≤-to-IsMeet₂ : {R S : Carrier} → S ≤ R → IsMeet R S S
≤-to-IsMeet₂ leq = record
  {bound₁ = leq
  ;bound₂ = ≤-refl
  ;universal = λ {X} X≤R X≤S → X≤S
  }
≤-to-IsMeet₁≈ : {R S M : Carrier} → IsMeet R S M → R ≤ S → M ≈ R
≤-to-IsMeet₁≈ m leq = IsMeet-unique m (≤-to-IsMeet₁ leq)
≤-to-IsMeet₂≈ : {R S M : Carrier} → IsMeet R S M → S ≤ R → M ≈ S
≤-to-IsMeet₂≈ m leq = IsMeet-unique m (≤-to-IsMeet₂ leq)
≤-to-Meet₁ : {R S : Carrier} → R ≤ S → Meet R S
≤-to-Meet₁ {R} {S} leq = record
  {value = R
  ;proof = ≤-to-IsMeet₁ leq
  }
≤-to-Meet₂ : {R S : Carrier} → S ≤ R → Meet R S
≤-to-Meet₂ {R} {S} leq = record
  {value = S
  ;proof = ≤-to-IsMeet₂ leq
  }
```

Where sufficient meets exist, these are associative:

```
IsMeet-assoc : {Q R S Q∧R R∧S M : Carrier} → IsMeet Q R Q∧R → IsMeet Q∧R S M
                                           → IsMeet R S R∧S  → IsMeet Q R∧S M
IsMeet-assoc Q-R Q∧R-S R-S = let open IsMeet in record
  {bound₁ = ≤-trans (bound₁ Q∧R-S) (bound₁ Q-R)
  ;bound₂ = universal R-S (≤-trans (bound₁ Q∧R-S) (bound₂ Q-R)) (bound₂ Q∧R-S)
  ;universal = λ {X} X≤Q X≤R∧S → universal Q∧R-S
        (universal Q-R X≤Q (≤-trans X≤R∧S (bound₁ R-S)))
        (≤-trans X≤R∧S (bound₂ R-S))
  }


Meet-assoc : {Q R S : Carrier} → (Q∧R : Meet Q R) → (M : Meet (Meet.value Q∧R) S)
  → (R∧S                             : Meet R S) → (N : Meet Q (Meet.value R∧S))
  → Meet.value M ≈ Meet.value N
Meet-assoc Q∧R M R∧S N = let open Meet in
  IsMeet-unique (IsMeet-assoc (proof Q∧R) (proof M) (proof R∧S)) (proof N)


IsMeet-distrR    : {Q R S R∧S M Q∧R Q∧S : Carrier}
                 → IsMeet R S R∧S  → IsMeet Q R∧S M
                 → IsMeet Q R Q∧R → IsMeet Q S Q∧S → IsMeet Q∧R Q∧S M
IsMeet-distrR R-S Q-R∧S Q-R Q-S   = let open IsMeet in record
```

```
    {bound₁  =  universal Q-R (bound₁ Q-R∧S) (≤-trans (bound₂ Q-R∧S) (bound₁ R-S))
    ; bound₂  =  universal Q-S (bound₁ Q-R∧S) (≤-trans (bound₂ Q-R∧S) (bound₂ R-S))
    ; universal  =  λ {X} X≤Q∧R X≤Q∧S → universal Q-R∧S
          (≤-trans X≤Q∧R (bound₁ Q-R))
          (universal R-S (≤-trans X≤Q∧R (bound₂ Q-R)) (≤-trans X≤Q∧S (bound₂ Q-S)))
    }

  Meet-distrR  : {Q R S : Carrier}
               → (R∧S  : Meet R S) → (M : Meet Q (Meet.value R∧S))
               → (Q∧R : Meet Q R) → (Q∧S : Meet Q S)
               → (N : Meet (Meet.value Q∧R) (Meet.value Q∧S))
               → Meet.value M ≈ Meet.value N
  Meet-distrR R∧S M Q∧R Q∧S N  =  let open Meet in
    IsMeet-unique (IsMeet-distrR (proof R∧S) (proof M) (proof Q∧R) (proof Q∧S)) (proof N)
```

Joins are obtained by renaming all members of the Meet module for the dualised ordering.

```
module PosetJoin {j k₁ k₂ : Level} (𝒫 : Poset j k₁ k₂) where
  open PosetMeet (dualPoset 𝒫) public renaming
    (IsMeet                  to IsJoin
    ; module IsMeet          to IsJoin
    ; Meet                   to Join
    ; module Meet            to Join
    ; isMeet-IsLowerBound₂   to isJoin-IsUpperBound₂
    ; isMeet-LowerBound₂     to isJoin-UpperBound₂
    ; isMeet-isGLB           to isJoin-isLUB
    ; isGLB-isMeet           to isLUB-isJoin
    ; IsMeet-idempotent      to IsJoin-idempotent
    ; idempotentMeet         to idempotentJoin
    ; IsMeet-sym             to IsJoin-sym
    ; commuteMeet            to commuteJoin
    ; IsMeet-monotone        to IsJoin-monotone
    ; Meet-monotone          to Join-monotone
    ; Meet-monotone₁         to Join-monotone₁
    ; Meet-monotone₂         to Join-monotone₂
    ; IsMeet-cong            to IsJoin-cong
    ; Meet-cong              to Join-cong
    ; Meet-cong₁             to Join-cong₁
    ; Meet-cong₂             to Join-cong₂
    ; cong-IsMeet            to cong-IsJoin
    ; IsMeet-unique          to IsJoin-unique
    ; ≤-from-IsMeet₁         to ≤-from-IsJoin₁
    ; ≤-from-IsMeet₂         to ≤-from-IsJoin₂
    ; ≤-from-Meet₁           to ≤-from-Join₁
    ; ≤-from-Meet₂           to ≤-from-Join₂
    ; ≤-to-IsMeet₁           to ≤-to-IsJoin₁
    ; ≤-to-IsMeet₂           to ≤-to-IsJoin₂
    ; ≤-to-IsMeet₁≈          to ≤-to-IsJoin₁≈
    ; ≤-to-IsMeet₂≈          to ≤-to-IsJoin₂≈
    ; ≤-to-Meet₁             to ≤-to-Join₁
    ; ≤-to-Meet₂             to ≤-to-Join₂
    ; IsMeet-assoc           to IsJoin-assoc
    ; Meet-assoc             to Join-assoc
    ; IsMeet-distrR          to IsJoin-distrR
    ; Meet-distrR            to Join-distrR
    )
```

A proof that all meets exist comes as a function meet that takes two carrier elements R and S to a Meet R S, and we can use this to define a meet operator and obtain simpler statements of the meet properties.

```
module LowerSemilattice {j k₁ k₂ : Level} (𝒫 : Poset j k₁ k₂)
                        (meet : (R S : Poset.Carrier 𝒫) → PosetMeet.Meet 𝒫 R S) where
  open Poset′ 𝒫
  open PosetMeet 𝒫

  infixr 7 _∧_
  _∧_ : Carrier → Carrier → Carrier
  R ∧ S = Meet.value (meet R S)
  ∧-IsMeet : {R S : Carrier} → IsMeet R S (R ∧ S)
  ∧-IsMeet {R} {S} = Meet.proof (meet R S)
  ∧-lower₁ : {R S : Carrier} → R ∧ S ≤ R
  ∧-lower₁ {R} {S} = Meet.bound₁ (meet R S)
  ∧-lower₂ : {R S : Carrier} → R ∧ S ≤ S
  ∧-lower₂ {R} {S} = Meet.bound₂ (meet R S)
  ∧-universal : {R S X : Carrier} → X ≤ R → X ≤ S → X ≤ R ∧ S
  ∧-universal {R} {S} = Meet.universal (meet R S)
  ∧-idempotent : {R : Carrier} → R ∧ R ≈ R
  ∧-idempotent = IsMeet-idempotent ∧-IsMeet
  ∧-monotone : {R₁ R₂ S₁ S₂ : Carrier} → R₁ ≤ R₂ → S₁ ≤ S₂ → R₁ ∧ S₁ ≤ R₂ ∧ S₂
  ∧-monotone {R₁} {R₂} {S₁} {S₂} = Meet-monotone (meet R₁ S₁) (meet R₂ S₂)
  ∧-monotone₁ : {R₁ R₂ S : Carrier} → R₁ ≤ R₂ → R₁ ∧ S ≤ R₂ ∧ S
  ∧-monotone₁ {R₁} {R₂} {S} = Meet-monotone₁ (meet R₁ S) (meet R₂ S)
  ∧-monotone₂ : {R S₁ S₂ : Carrier} → S₁ ≤ S₂ → R ∧ S₁ ≤ R ∧ S₂
  ∧-monotone₂ {R} {S₁} {S₂} = Meet-monotone₂ (meet R S₁) (meet R S₂)
  ∧-monotone₁₁ : {R₁ R₂ S T : Carrier} → R₁ ≤ R₂ → (R₁ ∧ S) ∧ T ≤ (R₂ ∧ S) ∧ T
  ∧-monotone₁₁ e = ∧-monotone₁ (∧-monotone₁ e)
  ∧-monotone₁₂ : {R S₁ S₂ T : Carrier} → S₁ ≤ S₂ → (R ∧ S₁) ∧ T ≤ (R ∧ S₂) ∧ T
  ∧-monotone₁₂ e = ∧-monotone₁ (∧-monotone₂ e)
  ∧-monotone₂₁ : {Q R₁ R₂ S : Carrier} → R₁ ≤ R₂ → Q ∧ (R₁ ∧ S) ≤ Q ∧ (R₂ ∧ S)
  ∧-monotone₂₁ e = ∧-monotone₂ (∧-monotone₁ e)
  ∧-monotone₂₂ : {Q R S₁ S₂ : Carrier} → S₁ ≤ S₂ → Q ∧ (R ∧ S₁) ≤ Q ∧ (R ∧ S₂)
  ∧-monotone₂₂ e = ∧-monotone₂ (∧-monotone₂ e)
  ∧-cong : {R₁ R₂ S₁ S₂ : Carrier} → R₁ ≈ R₂ → S₁ ≈ S₂ → R₁ ∧ S₁ ≈ R₂ ∧ S₂
  ∧-cong {R₁} {R₂} {S₁} {S₂} = Meet-cong (meet R₁ S₁) (meet R₂ S₂)
  ∧-cong₁ : {R₁ R₂ S : Carrier} → R₁ ≈ R₂ → R₁ ∧ S ≈ R₂ ∧ S
  ∧-cong₁ {R₁} {R₂} {S} = Meet-cong₁ (meet R₁ S) (meet R₂ S)
  ∧-cong₂ : {R S₁ S₂ : Carrier} → S₁ ≈ S₂ → R ∧ S₁ ≈ R ∧ S₂
  ∧-cong₂ {R} {S₁} {S₂} = Meet-cong₂ (meet R S₁) (meet R S₂)
  ∧-cong₁₁ : {R₁ R₂ S T : Carrier} → R₁ ≈ R₂ → (R₁ ∧ S) ∧ T ≈ (R₂ ∧ S) ∧ T
  ∧-cong₁₁ e = ∧-cong₁ (∧-cong₁ e)
  ∧-cong₁₂ : {R S₁ S₂ T : Carrier} → S₁ ≈ S₂ → (R ∧ S₁) ∧ T ≈ (R ∧ S₂) ∧ T
  ∧-cong₁₂ e = ∧-cong₁ (∧-cong₂ e)
  ∧-cong₂₁ : {Q R₁ R₂ S : Carrier} → R₁ ≈ R₂ → Q ∧ (R₁ ∧ S) ≈ Q ∧ (R₂ ∧ S)
  ∧-cong₂₁ e = ∧-cong₂ (∧-cong₁ e)
  ∧-cong₂₂ : {Q R S₁ S₂ : Carrier} → S₁ ≈ S₂ → Q ∧ (R ∧ S₁) ≈ Q ∧ (R ∧ S₂)
  ∧-cong₂₂ e = ∧-cong₂ (∧-cong₂ e)
  ≤-from-∧₁ : {R S : Carrier} → R ∧ S ≈ R → R ≤ S
  ≤-from-∧₁ {R} {S} = ≤-from-Meet₁ (meet R S)
  ≤-from-∧₂ : {R S : Carrier} → R ∧ S ≈ S → S ≤ R
  ≤-from-∧₂ {R} {S} = ≤-from-Meet₂ (meet R S)
  ≤-from-≤∧₁ : {R S : Carrier} → R ≤ R ∧ S → R ≤ S
  ≤-from-≤∧₁ {R} {S} R≤R∧S = ≤-trans R≤R∧S ∧-lower₂
  ≤-from-≤∧₂ : {R S : Carrier} → S ≤ R ∧ S → S ≤ R
```

```
≤-from-≤∧₂ {R} {S} S≤R∧S = ≤-trans S≤R∧S ∧-lower₁
≤-to-∧₁ : {R S : Carrier} → R ≤ S → R ∧ S ≈ R
≤-to-∧₁ = ≤-to-IsMeet₁≈ ∧-IsMeet
≤-to-∧₂ : {R S : Carrier} → S ≤ R → R ∧ S ≈ S
≤-to-∧₂ = ≤-to-IsMeet₂≈ ∧-IsMeet
∧-commutative : {R S : Carrier} → R ∧ S ≈ S ∧ R
∧-commutative = IsMeet-unique ∧-IsMeet (IsMeet-sym ∧-IsMeet)
∧-assoc : {Q R S : Carrier} → (Q ∧ R) ∧ S ≈ Q ∧ (R ∧ S)
∧-assoc {Q} {R} {S} = let Q∧R = meet Q R; R∧S = meet R S
  in Meet-assoc Q∧R (meet (Meet.value Q∧R) S) R∧S (meet Q (Meet.value R∧S))
```

The proof of ∧-assoc above relies on Meet_assoc; a proof that only uses properties of ∧ directly is in essence quite similar to the proof of IsMeet-assoc:

```
private
  ∧-assoc' : {Q R S : Carrier} → (Q ∧ R) ∧ S ≈ Q ∧ (R ∧ S)
  ∧-assoc' = ≤-antisym
    (∧-universal (≤-trans ∧-lower₁ ∧-lower₁) (∧-monotone₁      ∧-lower₂))
    (∧-universal (∧-monotone₂      ∧-lower₁) (≤-trans ∧-lower₂ ∧-lower₂))


∧-assocL : {Q R S : Carrier} → Q ∧ (R ∧ S) ≈ (Q ∧ R) ∧ S
∧-assocL = ≈-sym ∧-assoc
∧-assoc₃₊₁  : {f g h j : Carrier} → (f ∧ g ∧ h) ∧ j ≈ f ∧ (g ∧ (h ∧ j))
∧-assoc₃₊₁  = ∧-assoc ⟨≈≈⟩ ∧-cong₂ ∧-assoc
∧-assocL₃₊₁ : {f g h j : Carrier} → f ∧ (g ∧ (h ∧ j)) ≈ (f ∧ g ∧ h) ∧ j
∧-assocL₃₊₁ = ∧-cong₂ ∧-assocL ⟨≈≈⟩ ∧-assocL
∧-transpose₂  : {f g h j : Carrier} → (f ∧ g) ∧ (h ∧ j) ≈ (f ∧ h) ∧ (g ∧ j)
∧-transpose₂  = ∧-assoc ⟨≈≈⟩ ∧-cong₂ (∧-assocL ⟨≈≈⟩ ∧-cong₁ ∧-commutative ⟨≈≈⟩ ∧-assoc)
                ⟨≈≈⟩ ∧-assocL


∧-distrR-∧ : {Q R S : Carrier} → Q ∧ (R ∧ S) ≈ (Q ∧ R) ∧ (Q ∧ S)
∧-distrR-∧ {Q} {R} {S} = let R∧S = meet R S; Q∧R = meet Q R; Q∧S = meet Q S
  in Meet-distrR R∧S (meet Q (Meet.value R∧S))
                 Q∧R Q∧S (meet (Meet.value Q∧R) (Meet.value Q∧S))
∧-distrL-∧ : {Q R S : Carrier} → (Q ∧ R) ∧ S ≈ (Q ∧ S) ∧ (R ∧ S)
∧-distrL-∧ {Q} {R} {S} = let open EqR (posetSetoid 𝒫) in ≈-begin
      (Q ∧ R) ∧ S
    ≈⟨ ∧-commutative ⟩
      S ∧ (Q ∧ R)
    ≈⟨ ∧-distrR-∧ ⟩
      (S ∧ Q) ∧ (S ∧ R)
    ≈⟨ ∧-cong ∧-commutative ∧-commutative ⟩
      (Q ∧ S) ∧ (R ∧ S)
    ■
```

For items that don't include the symbol ∧ in their name we create a module that does:

```
module ∧-SL where


    -- X = (Q ⇒ R) in sublattice below top
  record IsRelativePseudocomplement (top Q R X : Carrier) : Set (j ⊍ k₂) where
    field
      bounded : X ≤ top
      sat : Q ∧ X ≤ R
      universal : {Y : Carrier} → Y ≤ top → Q ∧ Y ≤ R → Y ≤ X
```

Relative pseudo-complements, where they exist, are monotone in the `top` restriction and in the second argument (`R`), and antitone in the first argument (`Q`).

```
IsRelativePseudocomplement-monotone₀ : {top₁ top₂ Q R X₁ X₂ : Carrier}
  → IsRelativePseudocomplement top₁ Q R X₁
  → IsRelativePseudocomplement top₂ Q R X₂
  → top₁ ≤ top₂ → X₁ ≤ X₂
IsRelativePseudocomplement-monotone₀ irpc₁ irpc₂ top₁≤top₂ =
  let open IsRelativePseudocomplement
  in universal irpc₂ (≤-trans (bounded irpc₁) top₁≤top₂) (sat irpc₁)
IsRelativePseudocomplement-antitone₁ : {top Q₁ Q₂ R X₁ X₂ : Carrier}
  → IsRelativePseudocomplement top Q₁ R X₁
  → IsRelativePseudocomplement top Q₂ R X₂
  → Q₁ ≤ Q₂ → X₂ ≤ X₁
IsRelativePseudocomplement-antitone₁ irpc₁ irpc₂ Q₁≤Q₂ =
  let open IsRelativePseudocomplement
  in universal irpc₁ (bounded irpc₂) (≤-trans (∧-monotone₁ Q₁≤Q₂) (sat irpc₂))
IsRelativePseudocomplement-monotone₂ : {top Q R₁ R₂ X₁ X₂ : Carrier}
  → IsRelativePseudocomplement top Q R₁ X₁
  → IsRelativePseudocomplement top Q R₂ X₂
  → R₁ ≤ R₂ → X₁ ≤ X₂
IsRelativePseudocomplement-monotone₂ irpc₁ irpc₂ R₁≤R₂ =
  let open IsRelativePseudocomplement
  in universal irpc₂ (bounded irpc₁) (≤-trans (sat irpc₁) R₁≤R₂)
IsRelativePseudocomplement-cong₀ : {top₁ top₂ Q R X₁ X₂ : Carrier}
  → IsRelativePseudocomplement top₁ Q R X₁
  → IsRelativePseudocomplement top₂ Q R X₂
  → top₁ ≈ top₂ → X₁ ≈ X₂
IsRelativePseudocomplement-cong₀ irpc₁ irpc₂ eq = ≤-antisym
  (IsRelativePseudocomplement-monotone₀ irpc₁ irpc₂ (≤-reflexive eq))
  (IsRelativePseudocomplement-monotone₀ irpc₂ irpc₁ (≤-reflexive′ eq))
IsRelativePseudocomplement-cong₁ : {top Q₁ Q₂ R X₁ X₂ : Carrier}
  → IsRelativePseudocomplement top Q₁ R X₁
  → IsRelativePseudocomplement top Q₂ R X₂
  → Q₁ ≈ Q₂ → X₁ ≈ X₂
IsRelativePseudocomplement-cong₁ irpc₁ irpc₂ eq = ≤-antisym
  (IsRelativePseudocomplement-antitone₁ irpc₂ irpc₁ (≤-reflexive′ eq))
  (IsRelativePseudocomplement-antitone₁ irpc₁ irpc₂ (≤-reflexive eq))
IsRelativePseudocomplement-cong₂ : {top Q R₁ R₂ X₁ X₂ : Carrier}
  → IsRelativePseudocomplement top Q R₁ X₁
  → IsRelativePseudocomplement top Q R₂ X₂
  → R₁ ≈ R₂ → X₁ ≈ X₂
IsRelativePseudocomplement-cong₂ irpc₁ irpc₂ eq = ≤-antisym
  (IsRelativePseudocomplement-monotone₂ irpc₁ irpc₂ (≤-reflexive eq))
  (IsRelativePseudocomplement-monotone₂ irpc₂ irpc₁ (≤-reflexive′ eq))
```

Relative pseudo-complements, where they exist, also distribute over meets in the `top` restriction and in the second argument (`R`).

```
IsRelativePseudocomplement-meet₀ : {top₁ top₂ Q R X₁ X₂ : Carrier}
  → IsRelativePseudocomplement top₁ Q R X₁
  → IsRelativePseudocomplement top₂ Q R X₂
  → IsRelativePseudocomplement (top₁ ∧ top₂) Q R (X₁ ∧ X₂)
IsRelativePseudocomplement-meet₀ irpc₁ irpc₂ = let open IsRelativePseudocomplement in record
  {bounded = ∧-monotone (bounded irpc₁) (bounded irpc₂)
  ;sat = ≤-trans₂ ∧-assocL (≤-trans ∧-lower₁ (sat irpc₁))
  ;universal = λ {Y} Y≤top Q∧Y≤R → ∧-universal
    (universal irpc₁ (≤-trans Y≤top ∧-lower₁) Q∧Y≤R)
```

```
      (universal irpc₂ (≤-trans Y≤top ∧-lower₂) Q∧Y≤R)
    }
  IsRelativePseudocomplement-meet₂ : {top Q R₁ R₂ X₁ X₂ : Carrier}
    → IsRelativePseudocomplement top Q R₁ X₁
    → IsRelativePseudocomplement top Q R₂ X₂
    → IsRelativePseudocomplement top Q (R₁ ∧ R₂) (X₁ ∧ X₂)
  IsRelativePseudocomplement-meet₂ irpc₁ irpc₂ = let open IsRelativePseudocomplement in record
    {bounded = ≤-trans ∧-lower₁ (bounded irpc₁)
    ; sat = ≤-trans₂ ∧-distrR-∧ (∧-monotone (sat irpc₁) (sat irpc₂))
    ; universal = λ {Y} Y≤top Q∧Y≤R → ∧-universal
      (universal irpc₁ Y≤top (≤-trans Q∧Y≤R ∧-lower₁))
      (universal irpc₂ Y≤top (≤-trans Q∧Y≤R ∧-lower₂))
    }
```

In the sublattice below top, we have (Q ⇒ Q) ≈ top.

```
  selfIsPseudoComplement : {top Q : Carrier} → IsRelativePseudocomplement top Q Q top
  selfIsPseudoComplement {top} {Q} = record
    {bounded = ≤-refl
    ; sat = ∧-lower₁
    ; universal = λ {Y} Y≤top Q∧Y≤Q → Y≤top
    }
```

If X ≈ (Q ⇒ R) in the sublattice below top, then X ≈ (Q ⇒ (Q ∧ R)).

```
  propagateIsPseudoComplementAntecedent : {top Q R X : Carrier}
    → IsRelativePseudocomplement top Q R X
    → IsRelativePseudocomplement top Q (Q ∧ R) X
  propagateIsPseudoComplementAntecedent X=Q⇒R =
    let open IsRelativePseudocomplement X=Q⇒R in record
      {bounded = bounded
      ; sat = ∧-universal ∧-lower₁ sat
      ; universal = λ {Y} Y≤top Q∧Y≤Q∧R → universal Y≤top (≤-trans Q∧Y≤Q∧R ∧-lower₂)
      }


  record RelativePseudocomplement (top Q R : Carrier) : Set (j ⊍ k₂) where
    field
      value : Carrier
      proof : IsRelativePseudocomplement top Q R value
```

Difference based on pseudo-complement is defined as Q - R = (Q ∧ (R ⇒ bot)).

```
  IsPseudoDifference : (bot Q R X : Carrier) → Set (j ⊍ k₂)
  IsPseudoDifference bot Q R X = IsRelativePseudocomplement Q R bot X
  PseudoDifference : (bot Q R : Carrier) → Set (j ⊍ k₂)
  PseudoDifference bot Q R = RelativePseudocomplement Q R bot
```

For joins, we could of course just repeat the development of **module** LowerSemilattice, since it does not contain much substance of its own, but we prefer to perform another renaming, since that is guaranteed to preserve duality in all details.

```
module UpperSemilattice {j k₁ k₂ : Level} (𝒫 : Poset j k₁ k₂)
                        (join : (R S : Poset.Carrier 𝒫) → PosetJoin.Join 𝒫 R S)
  where
    open Poset 𝒫
    open LowerSemilattice (dualPoset 𝒫) join public using () renaming
      (_∧_           to _∨_
```

```
      ; ∧-IsMeet        to ∨-IsJoin
      ; ∧-lower₁        to ∨-upper₁
      ; ∧-lower₂        to ∨-upper₂
      ; ∧-universal     to ∨-universal
      ; ∧-idempotent    to ∨-idempotent
      ; ∧-monotone      to ∨-monotone
      ; ∧-monotone₁     to ∨-monotone₁
      ; ∧-monotone₂     to ∨-monotone₂
      ; ∧-monotone₁₁    to ∨-monotone₁₁
      ; ∧-monotone₁₂    to ∨-monotone₁₂
      ; ∧-monotone₂₁    to ∨-monotone₂₁
      ; ∧-monotone₂₂    to ∨-monotone₂₂
      ; ∧-cong          to ∨-cong
      ; ∧-cong₁         to ∨-cong₁
      ; ∧-cong₂         to ∨-cong₂
      ; ∧-cong₁₁        to ∨-cong₁₁
      ; ∧-cong₁₂        to ∨-cong₁₂
      ; ∧-cong₂₁        to ∨-cong₂₁
      ; ∧-cong₂₂        to ∨-cong₂₂
      ; ≤-from-∧₁       to ≤-from-∨₁     -- R ∨ S ≈ R → S ≤ R
      ; ≤-from-∧₂       to ≤-from-∨₂     -- R ∨ S ≈ S → R ≤ S
      ; ≤-from-≤∧₁      to ≤-from-∨≤₁    -- R ∨ S ≤ R → S ≤ R
      ; ≤-from-≤∧₂      to ≤-from-∨≤₂    -- R ∨ S ≤ S → R ≤ S
      ; ≤-to-∧₁         to ≤-to-∨₁       -- S ≤ R → R ∨ S ≈ R
      ; ≤-to-∧₂         to ≤-to-∨₂       -- R ≤ S → R ∨ S ≈ S
      ; ∧-commutative to ∨-commutative
      ; ∧-assoc         to ∨-assoc
      ; ∧-assoc₃₊₁      to ∨-assoc₃₊₁
      ; ∧-assocL₃₊₁     to ∨-assocL₃₊₁
      ; ∧-transpose₂    to ∨-transpose₂
      ; ∧-assocL        to ∨-assocL
      ; ∧-distrR-∧      to ∨-distrR-∨
      ; ∧-distrL-∧      to ∨-distrL-∨
      )
  module ∨-SL where
    open LowerSemilattice.∧-SL (dualPoset 𝒫) join public using () renaming
      (IsRelativePseudocomplement to IsRelativeSemicomplement
      ; module IsRelativePseudocomplement to IsRelativeSemicomplement
      ; IsRelativePseudocomplement-monotone₀ to IsRelativeSemicomplement-monotone₀
      ; IsRelativePseudocomplement-antitone₁ to IsRelativeSemicomplement-antitone₁
      ; IsRelativePseudocomplement-monotone₂ to IsRelativeSemicomplement-monotone₂
      ; IsRelativePseudocomplement-cong₀ to IsRelativeSemicomplement-cong₀
      ; IsRelativePseudocomplement-cong₁ to IsRelativeSemicomplement-cong₁
      ; IsRelativePseudocomplement-cong₂ to IsRelativeSemicomplement-cong₂
      ; IsRelativePseudocomplement-meet₀ to IsRelativeSemicomplement-join₀
      ; IsRelativePseudocomplement-meet₂ to IsRelativeSemicomplement-join₂
      ; selfIsPseudoComplement to selfIsSemiComplement
          -- {bot Q : Carrier} → IsRelativeSemicomplement bot Q Q bot
      ; propagateIsPseudoComplementAntecedent to propagateIsSemiComplementAntecedent
          -- {bot Q R X : Carrier} → IsRelativePseudocomplement bot Q R X
          -- → IsRelativePseudocomplement bot Q (Q ∨ R) X
      ; RelativePseudocomplement to RelativeSemicomplement
      ; module RelativePseudocomplement to RelativeSemicomplement
      )
```

IsRelativeSemicomplement bot Q R X means X = Q ⇨ R over bot, that is, X is least such that bot ≤ X and R ≤ Q ∨ X.

```
      -- Q - R = bot ∨ (R ⇨ Q)
    IsSemiDifference : (bot Q R X : Carrier) → Set (j ⊍ k₂)
```

```
    IsSemiDifference bot Q R X  =  IsRelativeSemicomplement bot R Q X
    SemiDifference : (bot Q R : Carrier) → Set (j ⊍ k₂)
    SemiDifference bot Q R  =  RelativeSemicomplement bot R Q


module LatticeProps {j k₁ k₂ : Level} (𝒫 : Poset j k₁ k₂)
                    (meet : (R S : Poset.Carrier 𝒫) → PosetMeet.Meet 𝒫 R S)
                    (join : (R S : Poset.Carrier 𝒫) → PosetJoin.Join 𝒫 R S) where
  open Poset′ 𝒫
  open LowerSemilattice 𝒫 meet
  open UpperSemilattice 𝒫 join
  ∧-∨-absorbR₁ : {Q R : Carrier} → (Q ∧ (Q ∨ R)) ≈ Q
  ∧-∨-absorbR₁  =  ≤-antisym ∧-lower₁ (∧-universal ≤-refl ∨-upper₁)
  ∧-∨-absorbR₂ : {Q R : Carrier} → (Q ∧ (R ∨ Q)) ≈ Q
  ∧-∨-absorbR₂  =  ≤-antisym ∧-lower₁ (∧-universal ≤-refl ∨-upper₂)
  ∧-∨-absorbL₁ : {Q R : Carrier} → ((Q ∨ R) ∧ Q) ≈ Q
  ∧-∨-absorbL₁  =  ≤-antisym ∧-lower₂ (∧-universal ∨-upper₁ ≤-refl)
  ∧-∨-absorbL₂ : {Q R : Carrier} → ((R ∨ Q) ∧ Q) ≈ Q
  ∧-∨-absorbL₂  =  ≤-antisym ∧-lower₂ (∧-universal ∨-upper₂ ≤-refl)
  ∨-∧-absorbR₁ : {Q R : Carrier} → (Q ∨ (Q ∧ R)) ≈ Q
  ∨-∧-absorbR₁  =  ≤-antisym (∨-universal ≤-refl ∧-lower₁) ∨-upper₁
  ∨-∧-absorbR₂ : {Q R : Carrier} → (Q ∨ (R ∧ Q)) ≈ Q
  ∨-∧-absorbR₂  =  ≤-antisym (∨-universal ≤-refl ∧-lower₂) ∨-upper₁
  ∨-∧-absorbL₁ : {Q R : Carrier} → ((Q ∧ R) ∨ Q) ≈ Q
  ∨-∧-absorbL₁  =  ≤-antisym (∨-universal ∧-lower₁ ≤-refl) ∨-upper₂
  ∨-∧-absorbL₂ : {Q R : Carrier} → ((R ∧ Q) ∨ Q) ≈ Q
  ∨-∧-absorbL₂  =  ≤-antisym (∨-universal ∧-lower₂ ≤-refl) ∨-upper₂
  ∧-∨-supdistribL : {Q R S : Carrier} → (Q ∧ S) ∨ (R ∧ S) ≤ ((Q ∨ R) ∧ S)
  ∧-∨-supdistribL  =  ∨-universal (∧-monotone₁ ∨-upper₁) (∧-monotone₁ ∨-upper₂)
  ∧-∨-supdistribR : {Q R S : Carrier} → (Q ∧ R) ∨ (Q ∧ S) ≤ (Q ∧ (R ∨ S))
  ∧-∨-supdistribR  =  ∨-universal (∧-monotone₂ ∨-upper₁) (∧-monotone₂ ∨-upper₂)
  ∨-∧-subdistribL : {Q R S : Carrier} → ((Q ∧ R) ∨ S) ≤ (Q ∨ S) ∧ (R ∨ S)
  ∨-∧-subdistribL  =  ∧-universal (∨-monotone₁ ∧-lower₁) (∨-monotone₁ ∧-lower₂)
  ∨-∧-subdistribR : {Q R S : Carrier} → (Q ∨ (R ∧ S)) ≤ (Q ∨ R) ∧ (Q ∨ S)
  ∨-∧-subdistribR  =  ∧-universal (∨-monotone₂ ∧-lower₁) (∨-monotone₂ ∧-lower₂)


module Lattice {j k₁ k₂ : Level} (𝒫 : Poset j k₁ k₂)
               (meet : (R S : Poset.Carrier 𝒫) → PosetMeet.Meet 𝒫 R S)
               (join : (R S : Poset.Carrier 𝒫) → PosetJoin.Join 𝒫 R S) where
  open Poset′            𝒫           public
  open LowerSemilattice 𝒫 meet      public
  open UpperSemilattice 𝒫 join      public
  open LatticeProps      𝒫 meet join public
```

Distributive lattices arising from right-subdistributivity of ∧ over ∨:

```
module RDistrLattice {j k₁ k₂ : Level} (𝒫 : Poset j k₁ k₂)
                     (meet : (R S : Poset.Carrier 𝒫) → PosetMeet.Meet 𝒫 R S)
                     (join : (R S : Poset.Carrier 𝒫) → PosetJoin.Join 𝒫 R S)
                     (∧-∨-subdistribR : let open Lattice 𝒫 meet join in
                              {Q R S : Carrier} → (Q ∧ (R ∨ S)) ≤ (Q ∧ R) ∨ (Q ∧ S)) where
  open Lattice 𝒫 meet join
  ∧-∨-distribR : {Q R S : Carrier} → (Q ∧ (R ∨ S)) ≈ (Q ∧ R) ∨ (Q ∧ S)
  ∧-∨-distribR {Q} {R} {S}  =  ≤-antisym ∧-∨-subdistribR ∧-∨-supdistribR
  ∧-∨-distribL : {Q R S : Carrier} → ((Q ∨ R) ∧ S) ≈ (Q ∧ S) ∨ (R ∧ S)
```

```
∧-∨-distribL = ∧-commutative ⟨≈≈⟩ ∧-∨-distribR
                             ⟨≈≈⟩ ∨-cong ∧-commutative ∧-commutative
∨-∧-distribR : {Q R S : Carrier} → (Q ∨ (R ∧ S)) ≈ (Q ∨ R) ∧ (Q ∨ S)
∨-∧-distribR {Q} {R} {S} = ≈-sym (let open EqR (posetSetoid 𝒫) in ≈-begin
    (Q ∨ R) ∧ (Q ∨ S)
  ≈⟨ ∧-∨-distribR ⟩
    ((Q ∨ R) ∧ Q) ∨ ((Q ∨ R) ∧ S)
  ≈⟨ ∨-cong₁ ∧-∨-absorbL₁ ⟩
    Q ∨ ((Q ∨ R) ∧ S)
  ≈⟨ ∨-cong₂ ∧-∨-distribL ⟩
    Q ∨ ((Q ∧ S) ∨ (R ∧ S))
  ≈⟨ ∨-assocL ⟩
    (Q ∨ (Q ∧ S)) ∨ (R ∧ S)
  ≈⟨ ∨-cong₁ ∨-∧-absorbR₁ ⟩
    Q ∨ (R ∧ S)
  ■)
∨-∧-distribL : {Q R S : Carrier}
             → ((Q ∧ R) ∨ S) ≈ (Q ∨ S) ∧ (R ∨ S)
∨-∧-distribL = ∨-commutative ⟨≈≈⟩ ∨-∧-distribR
                             ⟨≈≈⟩ ∧-cong ∨-commutative ∨-commutative
```

Arbitrary meets are formalised as meets of all elements in the range of a function:

```
module PosetMeetI {j k₁ k₂ : Level} (𝒫 : Poset j k₁ k₂) where
  open Poset′ 𝒫
  open LowerBounds

  record IsMeetI {i : Level} {I : Set i} (f : I → Carrier) (m : Carrier) : Set (i ⊍ j ⊍ k₂) where
    field
      bound : isLowerBoundI 𝒫 f m
      universal : {y : Carrier} → isLowerBoundI 𝒫 f y → y ≤ m
  record MeetI {i : Level} {I : Set i} (f : I → Carrier) : Set (i ⊍ j ⊍ k₂) where
    field
      value : Carrier
      proof : IsMeetI f value
    open IsMeetI proof public
```

Meets are greatest lower bounds:

```
  isMeetI-LowerBoundI : {i : Level} {I : Set i} {f : I → Carrier} {M : Carrier}
                      → IsMeetI f M → LowerBoundI 𝒫 f
  isMeetI-LowerBoundI {M = M} m = record {value = M; proof = IsMeetI.bound m}
  isMeetI-isGLB : {i : Level} {I : Set i} {f : I → Carrier} {M : Carrier} → (m : IsMeetI f M)
    → isGreatestElem (LowerBoundIPoset 𝒫 f) (isMeetI-LowerBoundI m)
  isMeetI-isGLB m = λ x → IsMeetI.universal m (LowerBoundI.proof x)
  isGLB-isMeetI : {i : Level} {I : Set i} {f : I → Carrier} {m : LowerBoundI 𝒫 f}
    → isGreatestElem (LowerBoundIPoset 𝒫 f) m
    → IsMeetI f (LowerBoundI.value m)
  isGLB-isMeetI {f = f} {m} g = record
    {bound = LowerBoundI.proof m
    ; universal = λ {y} ∀x→y≤x → g (record {value = y; proof = ∀x→y≤x})
    }
```

Renamed Interfaces for Different Ordering Symbols

```
module LowerSemilattice-round {j k₁ k₂ : Level} (𝒫 : Poset j k₁ k₂)
                              (meet : (R S : Poset.Carrier 𝒫) → PosetMeet.Meet 𝒫 R S) where
```

```
  open LowerSemilattice 𝒫 meet public using () renaming
    (_∧_             to _∩_             -- : Carrier → Carrier → Carrier
    ; ∧-IsMeet       to ∩-IsMeet        -- : {R S : _} → IsMeet R S (R ∩ S)
    ; ∧-lower₁       to ∩-lower₁        -- : {R S : _} → R ∩ S ⊆ R
    ; ∧-lower₂       to ∩-lower₂        -- : {R S : _} → R ∩ S ⊆ S
    ; ∧-universal    to ∩-universal     -- : {R S X : _} → X ⊆ R → X ⊆ S → X ⊆ R ∩ S
    ; ∧-idempotent   to ∩-idempotent    -- : {R : _} → R ∩ R ≈ R
    ; ∧-monotone     to ∩-monotone      -- : {R₁ R₂ S₁ S₂ : _}
                                        -- → R₁ ⊆ R₂ → S₁ ⊆ S₂ → R₁ ∩ S₁ ⊆ R₂ ∩ S₂
    ; ∧-monotone₁    to ∩-monotone₁     -- : {R₁ R₂ S : _} → R₁ ⊆ R₂ → R₁ ∩ S ⊆ R₂ ∩ S
    ; ∧-monotone₂    to ∩-monotone₂     -- : {R S₁ S₂ : _} → S₁ ⊆ S₂ → R ∩ S₁ ⊆ R ∩ S₂
    ; ∧-monotone₁₁   to ∩-monotone₁₁    -- : {R₁ R₂ S T : _} → R₁ ⊆ R₂ → (R₁ ∩ S) ∩ T ⊆ (R₂ ∩ S) ∩ T
    ; ∧-monotone₁₂   to ∩-monotone₁₂    -- : {R S₁ S₂ T : _} → S₁ ⊆ S₂ → (R ∩ S₁) ∩ T ⊆ (R ∩ S₂) ∩ T
    ; ∧-monotone₂₁   to ∩-monotone₂₁    -- : {Q R₁ R₂ S : _} → R₁ ⊆ R₂ → Q ∩ (R₁ ∩ S) ⊆ Q ∩ (R₂ ∩ S)
    ; ∧-monotone₂₂   to ∩-monotone₂₂    -- : {Q R S₁ S₂ : _} → S₁ ⊆ S₂ → Q ∩ (R ∩ S₁) ⊆ Q ∩ (R ∩ S₂)
    ; ∧-cong         to ∩-cong          -- : {R₁ R₂ S₁ S₂ : _}
                                        -- → R₁ ≈ R₂ → S₁ ≈ S₂ → R₁ ∩ S₁ ≈ R₂ ∩ S₂
    ; ∧-cong₁        to ∩-cong₁         -- : {R₁ R₂ S : _} → R₁ ≈ R₂ → R₁ ∩ S ≈ R₂ ∩ S
    ; ∧-cong₂        to ∩-cong₂         -- : {R S₁ S₂ : _} → S₁ ≈ S₂ → R ∩ S₁ ≈ R ∩ S₂
    ; ∧-cong₁₁       to ∩-cong₁₁        -- : {R₁ R₂ S T : _} → R₁ ≈ R₂ → (R₁ ∩ S) ∩ T ≈ (R₂ ∩ S) ∩ T
    ; ∧-cong₁₂       to ∩-cong₁₂        -- : {R S₁ S₂ T : _} → S₁ ≈ S₂ → (R ∩ S₁) ∩ T ≈ (R ∩ S₂) ∩ T
    ; ∧-cong₂₁       to ∩-cong₂₁        -- : {Q R₁ R₂ S : _} → R₁ ≈ R₂ → Q ∩ (R₁ ∩ S) ≈ Q ∩ (R₂ ∩ S)
    ; ∧-cong₂₂       to ∩-cong₂₂        -- : {Q R S₁ S₂ : _} → S₁ ≈ S₂ → Q ∩ (R ∩ S₁) ≈ Q ∩ (R ∩ S₂)
    ; ≤-from-∧₁      to ⊆-from-∩₁       -- : {R S : _} → R ∩ S ≈ R → R ⊆ S
    ; ≤-from-∧₂      to ⊆-from-∩₂       -- : {R S : _} → R ∩ S ≈ S → S ⊆ R
    ; ≤-from-≤∧₁     to ⊆-from-⊆∩₁      -- : {R S : _} → R ⊆ R ∩ S → R ⊆ S
    ; ≤-from-≤∧₂     to ⊆-from-⊆∩₂      -- : {R S : _} → S ⊆ R ∩ S → S ⊆ R
    ; ≤-to-∧₁        to ⊆-to-∩₁         -- : {R S : _} → R ⊆ S → R ∩ S ≈ R
    ; ≤-to-∧₂        to ⊆-to-∩₂         -- : {R S : _} → S ⊆ R → R ∩ S ≈ S
    ; ∧-commutative  to ∩-commutative   -- : {R S : _} → R ∩ S ≈ S ∩ R
    ; ∧-assoc        to ∩-assoc         -- : {Q R S : _} → (Q ∩ R) ∩ S ≈ Q ∩ (R ∩ S)
    ; ∧-assocL       to ∩-assocL        -- : {Q R S : _} → Q ∩ (R ∩ S) ≈ (Q ∩ R) ∩ S
    ; ∧-assoc₃₊₁     to ∩-assoc₃₊₁      -- : {P Q R S : _} → (P ∩ Q ∩ R) ∩ S ≈ P ∩ (Q ∩ R ∩ S)
    ; ∧-assocL₃₊₁    to ∩-assocL₃₊₁     -- : {P Q R S : _} → P ∩ (Q ∩ R ∩ S) ≈ (P ∩ Q ∩ R) ∩ S
    ; ∧-transpose₂ to ∩-transpose₂  -- : {P Q R S : _} → (P ∩ Q) ∩ (R ∩ S) ≈ (P ∩ R) ∩ (Q ∩ S)
    ; ∧-distrR-∧     to ∩-distrR-∩      -- : {Q R S : _} → Q ∩ (R ∩ S) ≈ (Q ∩ R) ∩ (Q ∩ S)
    ; ∧-distrL-∧     to ∩-distrL-∩      -- : {Q R S : _} → (Q ∩ R) ∩ S ≈ (Q ∩ S) ∩ (R ∩ S)
    ; module ∧-SL    to ∩-SL
    )

module UpperSemilattice-round {j k₁ k₂ : Level} (𝒫 : Poset j k₁ k₂)
                              (join : (R S : Poset.Carrier 𝒫) → PosetJoin.Join 𝒫 R S) where
  open UpperSemilattice 𝒫 join public using () renaming
    (_∨_             to _∪_             -- : Carrier → Carrier → Carrier
    ; ∨-IsJoin       to ∪-IsJoin        -- : {R S : _} → IsJoin R S (R ∪ S)
    ; ∨-upper₁       to ∪-upper₁        -- : {R S : _} → R ⊆ R ∪ S
    ; ∨-upper₂       to ∪-upper₂        -- : {R S : _} → S ⊆ R ∪ S
    ; ∨-universal    to ∪-universal     -- : {R S X : _} → R ⊆ X → S ⊆ X → R ∪ S ⊆ X
    ; ∨-idempotent   to ∪-idempotent    -- : {R : _} → R ∪ R ≈ R
    ; ∨-monotone     to ∪-monotone      -- : {R₁ R₂ S₁ S₂ : _}
                                        -- → R₁ ⊆ R₂ → S₁ ⊆ S₂ → R₁ ∪ S₁ ⊆ R₂ ∪ S₂
    ; ∨-monotone₁    to ∪-monotone₁     -- : {R₁ R₂ S : _} → R₁ ⊆ R₂ → R₁ ∪ S ⊆ R₂ ∪ S
    ; ∨-monotone₂    to ∪-monotone₂     -- : {R S₁ S₂ : _} → S₁ ⊆ S₂ → R ∪ S₁ ⊆ R ∪ S₂
    ; ∨-monotone₁₁   to ∪-monotone₁₁    -- : {R₁ R₂ S T : _} → R₁ ⊆ R₂ → (R₁ ∪ S) ∪ T ⊆ (R₂ ∪ S) ∪ T
    ; ∨-monotone₁₂   to ∪-monotone₁₂    -- : {R S₁ S₂ T : _} → S₁ ⊆ S₂ → (R ∪ S₁) ∪ T ⊆ (R ∪ S₂) ∪ T
    ; ∨-monotone₂₁   to ∪-monotone₂₁    -- : {Q R₁ R₂ S : _} → R₁ ⊆ R₂ → Q ∪ (R₁ ∪ S) ⊆ Q ∪ (R₂ ∪ S)
    ; ∨-monotone₂₂   to ∪-monotone₂₂    -- : {Q R S₁ S₂ : _} → S₁ ⊆ S₂ → Q ∪ (R ∪ S₁) ⊆ Q ∪ (R ∪ S₂)
    ; ∨-cong         to ∪-cong          -- : {R₁ R₂ S₁ S₂ : _}
                                        -- → R₁ ≈ R₂ → S₁ ≈ S₂ → R₁ ∪ S₁ ≈ R₂ ∪ S₂
```

```
    ; ∨-cong₁           to ∪-cong₁          -- : {R₁ R₂ S : _} → R₁ ≈ R₂ → R₁ ∪ S ≈ R₂ ∪ S
    ; ∨-cong₂           to ∪-cong₂          -- : {R S₁ S₂ : _} → S₁ ≈ S₂ → R ∪ S₁ ≈ R ∪ S₂
    ; ∨-cong₁₁          to ∪-cong₁₁         -- : {R₁ R₂ S T : _} → R₁ ≈ R₂ → (R₁ ∪ S) ∪ T ≈ (R₂ ∪ S) ∪ T
    ; ∨-cong₁₂          to ∪-cong₁₂         -- : {R S₁ S₂ T : _} → S₁ ≈ S₂ → (R ∪ S₁) ∪ T ≈ (R ∪ S₂) ∪ T
    ; ∨-cong₂₁          to ∪-cong₂₁         -- : {Q R₁ R₂ S : _} → R₁ ≈ R₂ → Q ∪ (R₁ ∪ S) ≈ Q ∪ (R₂ ∪ S)
    ; ∨-cong₂₂          to ∪-cong₂₂         -- : {Q R S₁ S₂ : _} → S₁ ≈ S₂ → Q ∪ (R ∪ S₁) ≈ Q ∪ (R ∪ S₂)
    ; ≤-from-∨₁         to ⊆-from-∪₁        -- : {R S : _} → R ∪ S ≈ R → S ⊆ R
    ; ≤-from-∨₂         to ⊆-from-∪₂        -- : {R S : _} → R ∪ S ≈ S → R ⊆ S
    ; ≤-from-∨≤₁        to ⊆-from-∪⊆₁       -- : {R S : _} → R ∪ S ⊆ R → S ⊆ R
    ; ≤-from-∨≤₂        to ⊆-from-∪⊆₂       -- : {R S : _} → R ∪ S ⊆ S → R ⊆ S
    ; ≤-to-∨₁           to ⊆-to-∪₁          -- : {R S : _} → S ⊆ R → R ∪ S ≈ R
    ; ≤-to-∨₂           to ⊆-to-∪₂          -- : {R S : _} → R ⊆ S → R ∪ S ≈ S
    ; ∨-commutative     to ∪-commutative    -- : {R S : _} → R ∪ S ≈ S ∪ R
    ; ∨-assoc           to ∪-assoc          -- : {Q R S : _} → (Q ∪ R) ∪ S ≈ Q ∪ (R ∪ S)
    ; ∨-assocL          to ∪-assocL         -- : {Q R S : _} → Q ∪ (R ∪ S) ≈ (Q ∪ R) ∪ S
    ; ∨-assoc₃₊₁        to ∪-assoc₃₊₁       -- : {P Q R S : _} → (P ∪ Q ∪ R) ∪ S ≈ P ∪ (Q ∪ R ∪ S)
    ; ∨-assocL₃₊₁       to ∪-assocL₃₊₁      -- : {P Q R S : _} → P ∪ (Q ∪ R ∪ S) ≈ (P ∪ Q ∪ R) ∪ S
    ; ∨-transpose₂ to ∪-transpose₂ -- : {P Q R S : _} → (P ∪ Q) ∪ (R ∪ S) ≈ (P ∪ R) ∪ (Q ∪ S)
    ; ∨-distrR-∨        to ∪-distrR-∪       -- : {Q R S : _} → Q ∪ (R ∪ S) ≈ (Q ∪ R) ∪ (Q ∪ S)
    ; ∨-distrL-∨        to ∪-distrL-∪       -- : {Q R S : _} → (Q ∪ R) ∪ S ≈ (Q ∪ S) ∪ (R ∪ S)
    ; module ∨-SL       to ∪-SL
    )

module LatticeProps-round {j k₁ k₂ : Level} (𝒫 : Poset j k₁ k₂)
                          (meet : (R S : Poset.Carrier 𝒫) → PosetMeet.Meet 𝒫 R S)
                          (join : (R S : Poset.Carrier 𝒫) → PosetJoin.Join 𝒫 R S) where
  open LatticeProps 𝒫 meet join public renaming
    (∧-∨-absorbR₁    to ∩-∪-absorbR₁     -- : {Q R : Carrier} → (Q ∩ (Q ∪ R)) ≈ Q
    ; ∧-∨-absorbR₂   to ∩-∪-absorbR₂     -- : {Q R : Carrier} → (Q ∩ (R ∪ Q)) ≈ Q
    ; ∧-∨-absorbL₁   to ∩-∪-absorbL₁     -- : {Q R : Carrier} → ((Q ∪ R) ∩ Q) ≈ Q
    ; ∧-∨-absorbL₂   to ∩-∪-absorbL₂     -- : {Q R : Carrier} → ((R ∪ Q) ∩ Q) ≈ Q
    ; ∨-∧-absorbR₁   to ∪-∩-absorbR₁     -- : {Q R : Carrier} → (Q ∪ (Q ∩ R)) ≈ Q
    ; ∨-∧-absorbR₂   to ∪-∩-absorbR₂     -- : {Q R : Carrier} → (Q ∪ (R ∩ Q)) ≈ Q
    ; ∨-∧-absorbL₁   to ∪-∩-absorbL₁     -- : {Q R : Carrier} → ((Q ∩ R) ∪ Q) ≈ Q
    ; ∨-∧-absorbL₂   to ∪-∩-absorbL₂     -- : {Q R : Carrier} → ((R ∩ Q) ∪ Q) ≈ Q
    ; ∧-∨-supdistribL to ∩-∪-supdistribL -- : {Q R S : Carrier} → (Q ∩ S) ∪ (R ∩ S) ⊆ ((Q ∪ R) ∩ S)
    ; ∧-∨-supdistribR to ∩-∪-supdistribR -- : {Q R S : Carrier} → (Q ∩ R) ∪ (Q ∩ S) ⊆ (Q ∩ (R ∪ S))
    ; ∨-∧-subdistribL to ∪-∩-supdistribL -- : {Q R S : Carrier} → ((Q ∩ R) ∪ S) ⊆ (Q ∪ S) ∩ (R ∪ S)
    ; ∨-∧-subdistribR to ∪-∩-supdistribR -- : {Q R S : Carrier} → (Q ∪ (R ∩ S)) ⊆ (Q ∪ R) ∩ (Q ∪ S)
    )

module RDistrLattice-round {j k₁ k₂ : Level} (𝒫 : Poset j k₁ k₂)
                           (meet : (R S : Poset.Carrier 𝒫) → PosetMeet.Meet 𝒫 R S)
                           (join : (R S : Poset.Carrier 𝒫) → PosetJoin.Join 𝒫 R S)
                           (∧-∨-subdistribR : let open Lattice 𝒫 meet join in
                                     {Q R S : Carrier} → (Q ∧ (R ∨ S)) ≤ (Q ∧ R) ∨ (Q ∧ S))
  where
  open RDistrLattice 𝒫 meet join ∧-∨-subdistribR using () renaming
    (∧-∨-distribR to ∩-∪-distribR  -- : {Q R S : Carrier} → (Q ∩ (R ∪ S)) ≈ (Q ∩ R) ∪ (Q ∩ S)
    ; ∧-∨-distribL to ∩-∪-distribL -- : {Q R S : Carrier} → ((Q ∪ R) ∩ S) ≈ (Q ∩ S) ∪ (R ∩ S)
    ; ∨-∧-distribR to ∪-∩-distribR -- : {Q R S : Carrier} → (Q ∪ (R ∩ S)) ≈ (Q ∪ R) ∩ (Q ∪ S)
    ; ∨-∧-distribL to ∪-∩-distribL -- : {Q R S : Carrier} → ((Q ∩ R) ∪ S) ≈ (Q ∪ S) ∩ (R ∪ S)
    )

module LowerSemilattice-square {j k₁ k₂ : Level} (𝒫 : Poset j k₁ k₂)
                               (meet : (R S : Poset.Carrier 𝒫) → PosetMeet.Meet 𝒫 R S) where
  open LowerSemilattice 𝒫 meet public using () renaming
```

```
    ( _∧_              to _⊓_               -- : Carrier → Carrier → Carrier
    ; ∧-IsMeet         to ⊓-IsMeet          -- : {R S : _} → IsMeet R S (R ⊓ S)
    ; ∧-lower₁         to ⊓-lower₁          -- : {R S : _} → R ⊓ S ⊑ R
    ; ∧-lower₂         to ⊓-lower₂          -- : {R S : _} → R ⊓ S ⊑ S
    ; ∧-universal      to ⊓-universal       -- : {R S X : _} → X ⊑ R → X ⊑ S → X ⊑ R ⊓ S
    ; ∧-idempotent     to ⊓-idempotent      -- : {R : _} → R ⊓ R ≈ R
    ; ∧-monotone       to ⊓-monotone        -- : {R₁ R₂ S₁ S₂ : _}
                                            -- → R₁ ⊑ R₂ → S₁ ⊑ S₂ → R₁ ⊓ S₁ ⊑ R₂ ⊓ S₂
    ; ∧-monotone₁      to ⊓-monotone₁       -- : {R₁ R₂ S : _} → R₁ ⊑ R₂ → R₁ ⊓ S ⊑ R₂ ⊓ S
    ; ∧-monotone₂      to ⊓-monotone₂       -- : {R S₁ S₂ : _} → S₁ ⊑ S₂ → R ⊓ S₁ ⊑ R ⊓ S₂
    ; ∧-monotone₁₁     to ⊓-monotone₁₁     -- : {R₁ R₂ S T : _} → R₁ ⊑ R₂ → (R₁ ⊓ S) ⊓ T ⊑ (R₂ ⊓ S) ⊓ T
    ; ∧-monotone₁₂     to ⊓-monotone₁₂     -- : {R S₁ S₂ T : _} → S₁ ⊑ S₂ → (R ⊓ S₁) ⊓ T ⊑ (R ⊓ S₂) ⊓ T
    ; ∧-monotone₂₁     to ⊓-monotone₂₁     -- : {Q R₁ R₂ S : _} → R₁ ⊑ R₂ → Q ⊓ (R₁ ⊓ S) ⊑ Q ⊓ (R₂ ⊓ S)
    ; ∧-monotone₂₂     to ⊓-monotone₂₂     -- : {Q R S₁ S₂ : _} → S₁ ⊑ S₂ → Q ⊓ (R ⊓ S₁) ⊑ Q ⊓ (R ⊓ S₂)
    ; ∧-cong           to ⊓-cong            -- : {R₁ R₂ S₁ S₂ : _}
                                            -- → R₁ ≈ R₂ → S₁ ≈ S₂ → R₁ ⊓ S₁ ≈ R₂ ⊓ S₂
    ; ∧-cong₁          to ⊓-cong₁           -- : {R₁ R₂ S : _} → R₁ ≈ R₂ → R₁ ⊓ S ≈ R₂ ⊓ S
    ; ∧-cong₂          to ⊓-cong₂           -- : {R S₁ S₂ : _} → S₁ ≈ S₂ → R ⊓ S₁ ≈ R ⊓ S₂
    ; ∧-cong₁₁         to ⊓-cong₁₁         -- : {R₁ R₂ S T : _} → R₁ ≈ R₂ → (R₁ ⊓ S) ⊓ T ≈ (R₂ ⊓ S) ⊓ T
    ; ∧-cong₁₂         to ⊓-cong₁₂         -- : {R S₁ S₂ T : _} → S₁ ≈ S₂ → (R ⊓ S₁) ⊓ T ≈ (R ⊓ S₂) ⊓ T
    ; ∧-cong₂₁         to ⊓-cong₂₁         -- : {Q R₁ R₂ S : _} → R₁ ≈ R₂ → Q ⊓ (R₁ ⊓ S) ≈ Q ⊓ (R₂ ⊓ S)
    ; ∧-cong₂₂         to ⊓-cong₂₂         -- : {Q R S₁ S₂ : _} → S₁ ≈ S₂ → Q ⊓ (R ⊓ S₁) ≈ Q ⊓ (R ⊓ S₂)
    ; ≤-from-∧₁        to ⊑-from-⊓₁         -- : {R S : _} → R ⊓ S ≈ R → R ⊑ S
    ; ≤-from-∧₂        to ⊑-from-⊓₂         -- : {R S : _} → R ⊓ S ≈ S → S ⊑ R
    ; ≤-from-≤∧₁       to ⊑-from-⊑⊓₁        -- : {R S : _} → R ⊑ R ⊓ S → R ⊑ S
    ; ≤-from-≤∧₂       to ⊑-from-⊑⊓₂        -- : {R S : _} → S ⊑ R ⊓ S → S ⊑ R
    ; ≤-to-∧₁          to ⊑-to-⊓₁           -- : {R S : _} → R ⊑ S → R ⊓ S ≈ R
    ; ≤-to-∧₂          to ⊑-to-⊓₂           -- : {R S : _} → S ⊑ R → R ⊓ S ≈ S
    ; ∧-commutative to ⊓-commutative        -- : {R S : _} → R ⊓ S ≈ S ⊓ R
    ; ∧-assoc          to ⊓-assoc           -- : {Q R S : _} → (Q ⊓ R) ⊓ S ≈ Q ⊓ (R ⊓ S)
    ; ∧-assocL         to ⊓-assocL          -- : {Q R S : _} → Q ⊓ (R ⊓ S) ≈ (Q ⊓ R) ⊓ S
    ; ∧-assoc₃₊₁       to ⊓-assoc₃₊₁        -- : {P Q R S : _} → (P ⊓ Q ⊓ R) ⊓ S ≈ P ⊓ (Q ⊓ R ⊓ S)
    ; ∧-assocL₃₊₁      to ⊓-assocL₃₊₁       -- : {P Q R S : _} → P ⊓ (Q ⊓ R ⊓ S) ≈ (P ⊓ Q ⊓ R) ⊓ S
    ; ∧-transpose₂ to ⊓-transpose₂    -- : {P Q R S : _} → (P ⊓ Q) ⊓ (R ⊓ S) ≈ (P ⊓ R) ⊓ (Q ⊓ S)
    ; ∧-distrR-∧       to ⊓-distrR-⊓        -- : {Q R S : _} → Q ⊓ (R ⊓ S) ≈ (Q ⊓ R) ⊓ (Q ⊓ S)
    ; ∧-distrL-∧       to ⊓-distrL-⊓        -- : {Q R S : _} → (Q ⊓ R) ⊓ S ≈ (Q ⊓ S) ⊓ (R ⊓ S)
    ; module ∧-SL      to ⊓-SL
    )

module UpperSemilattice-square {j k₁ k₂ : Level} (𝒫 : Poset j k₁ k₂)
                                 (join : (R S : Poset.Carrier 𝒫) → PosetJoin.Join 𝒫 R S) where
  open UpperSemilattice 𝒫 join public using () renaming
    ( _∨_              to _⊔_               -- : Carrier → Carrier → Carrier
    ; ∨-IsJoin         to ⊔-IsJoin          -- : {R S : _} → IsJoin R S (R ⊔ S)
    ; ∨-upper₁         to ⊔-upper₁          -- : {R S : _} → R ⊑ R ⊔ S
    ; ∨-upper₂         to ⊔-upper₂          -- : {R S : _} → S ⊑ R ⊔ S
    ; ∨-universal      to ⊔-universal       -- : {R S X : _} → R ⊑ X → S ⊑ X → R ⊔ S ⊑ X
    ; ∨-idempotent     to ⊔-idempotent      -- : {R : _} → R ⊔ R ≈ R
    ; ∨-monotone       to ⊔-monotone        -- : {R₁ R₂ S₁ S₂ : _}
                                            -- → R₁ ⊑ R₂ → S₁ ⊑ S₂ → R₁ ⊔ S₁ ⊑ R₂ ⊔ S₂
    ; ∨-monotone₁      to ⊔-monotone₁       -- : {R₁ R₂ S : _} → R₁ ⊑ R₂ → R₁ ⊔ S ⊑ R₂ ⊔ S
    ; ∨-monotone₂      to ⊔-monotone₂       -- : {R S₁ S₂ : _} → S₁ ⊑ S₂ → R ⊔ S₁ ⊑ R ⊔ S₂
    ; ∨-monotone₁₁     to ⊔-monotone₁₁     -- : {R₁ R₂ S T : _} → R₁ ⊑ R₂ → (R₁ ⊔ S) ⊔ T ⊑ (R₂ ⊔ S) ⊔ T
    ; ∨-monotone₁₂     to ⊔-monotone₁₂     -- : {R S₁ S₂ T : _} → S₁ ⊑ S₂ → (R ⊔ S₁) ⊔ T ⊑ (R ⊔ S₂) ⊔ T
    ; ∨-monotone₂₁     to ⊔-monotone₂₁     -- : {Q R₁ R₂ S : _} → R₁ ⊑ R₂ → Q ⊔ (R₁ ⊔ S) ⊑ Q ⊔ (R₂ ⊔ S)
    ; ∨-monotone₂₂     to ⊔-monotone₂₂     -- : {Q R S₁ S₂ : _} → S₁ ⊑ S₂ → Q ⊔ (R ⊔ S₁) ⊑ Q ⊔ (R ⊔ S₂)
    ; ∨-cong           to ⊔-cong            -- : {R₁ R₂ S₁ S₂ : _}
                                            -- → R₁ ≈ R₂ → S₁ ≈ S₂ → R₁ ⊔ S₁ ≈ R₂ ⊔ S₂
    ; ∨-cong₁          to ⊔-cong₁           -- : {R₁ R₂ S : _} → R₁ ≈ R₂ → R₁ ⊔ S ≈ R₂ ⊔ S
```

```
    ; ∨-cong₂           to ⊔-cong₂            -- : {R S₁ S₂ : _} → S₁ ≈ S₂ → R ⊔ S₁ ≈ R ⊔ S₂
    ; ∨-cong₁₁          to ⊔-cong₁₁           -- : {R₁ R₂ S T : _} → R₁ ≈ R₂ → (R₁ ⊔ S) ⊔ T ≈ (R₂ ⊔ S) ⊔ T
    ; ∨-cong₁₂          to ⊔-cong₁₂           -- : {R S₁ S₂ T : _} → S₁ ≈ S₂ → (R ⊔ S₁) ⊔ T ≈ (R ⊔ S₂) ⊔ T
    ; ∨-cong₂₁          to ⊔-cong₂₁           -- : {Q R₁ R₂ S : _} → R₁ ≈ R₂ → Q ⊔ (R₁ ⊔ S) ≈ Q ⊔ (R₂ ⊔ S)
    ; ∨-cong₂₂          to ⊔-cong₂₂           -- : {Q R S₁ S₂ : _} → S₁ ≈ S₂ → Q ⊔ (R ⊔ S₁) ≈ Q ⊔ (R ⊔ S₂)
    ; ≤-from-∨₁         to ⊑-from-⊔₁          -- : {R S : _} → R ⊔ S ≈ R → S ⊑ R
    ; ≤-from-∨₂         to ⊑-from-⊔₂          -- : {R S : _} → R ⊔ S ≈ S → R ⊑ S
    ; ≤-from-∨≤₁        to ⊑-from-⊔⊑₁         -- : {R S : _} → R ⊔ S ⊑ R → S ⊑ R
    ; ≤-from-∨≤₂        to ⊑-from-⊔⊑₂         -- : {R S : _} → R ⊔ S ⊑ S → R ⊑ S
    ; ≤-to-∨₁           to ⊑-to-⊔₁            -- : {R S : _} → S ⊑ R → R ⊔ S ≈ R
    ; ≤-to-∨₂           to ⊑-to-⊔₂            -- : {R S : _} → R ⊑ S → R ⊔ S ≈ S
    ; ∨-commutative to ⊔-commutative          -- : {R S : _} → R ⊔ S ≈ S ⊔ R
    ; ∨-assoc           to ⊔-assoc            -- : {Q R S : _} → (Q ⊔ R) ⊔ S ≈ Q ⊔ (R ⊔ S)
    ; ∨-assocL          to ⊔-assocL           -- : {Q R S : _} → Q ⊔ (R ⊔ S) ≈ (Q ⊔ R) ⊔ S
    ; ∨-assoc₃₊₁        to ⊔-assoc₃₊₁         -- : {P Q R S : _} → (P ⊔ Q ⊔ R) ⊔ S ≈ P ⊔ (Q ⊔ R ⊔ S)
    ; ∨-assocL₃₊₁       to ⊔-assocL₃₊₁        -- : {P Q R S : _} → P ⊔ (Q ⊔ R ⊔ S) ≈ (P ⊔ Q ⊔ R) ⊔ S
    ; ∨-transpose₂ to ⊔-transpose₂   -- : {P Q R S : _} → (P ⊔ Q) ⊔ (R ⊔ S) ≈ (P ⊔ R) ⊔ (Q ⊔ S)
    ; ∨-distrR-∨        to ⊔-distrR-⊔         -- : {Q R S : _} → Q ⊔ (R ⊔ S) ≈ (Q ⊔ R) ⊔ (Q ⊔ S)
    ; ∨-distrL-∨        to ⊔-distrL-⊔         -- : {Q R S : _} → (Q ⊔ R) ⊔ S ≈ (Q ⊔ S) ⊔ (R ⊔ S)
    ; module ∨-SL       to ⊔-SL
    )

module LatticeProps-square {j k₁ k₂ : Level} (𝒫 : Poset j k₁ k₂)
                              (meet : (R S : Poset.Carrier 𝒫) → PosetMeet.Meet 𝒫 R S)
                              (join : (R S : Poset.Carrier 𝒫) → PosetJoin.Join 𝒫 R S) where
  open LatticeProps 𝒫 meet join public renaming
    (∧-∨-absorbR₁    to ⊓-⊔-absorbR₁      -- : {Q R : Carrier} → (Q ⊓ (Q ⊔ R)) ≈ Q
    ; ∧-∨-absorbR₂   to ⊓-⊔-absorbR₂      -- : {Q R : Carrier} → (Q ⊓ (R ⊔ Q)) ≈ Q
    ; ∧-∨-absorbL₁   to ⊓-⊔-absorbL₁      -- : {Q R : Carrier} → ((Q ⊔ R) ⊓ Q) ≈ Q
    ; ∧-∨-absorbL₂   to ⊓-⊔-absorbL₂      -- : {Q R : Carrier} → ((R ⊔ Q) ⊓ Q) ≈ Q
    ; ∨-∧-absorbR₁   to ⊔-⊓-absorbR₁      -- : {Q R : Carrier} → (Q ⊔ (Q ⊓ R)) ≈ Q
    ; ∨-∧-absorbR₂   to ⊔-⊓-absorbR₂      -- : {Q R : Carrier} → (Q ⊔ (R ⊓ Q)) ≈ Q
    ; ∨-∧-absorbL₁   to ⊔-⊓-absorbL₁      -- : {Q R : Carrier} → ((Q ⊓ R) ⊔ Q) ≈ Q
    ; ∨-∧-absorbL₂   to ⊔-⊓-absorbL₂      -- : {Q R : Carrier} → ((R ⊓ Q) ⊔ Q) ≈ Q
    ; ∧-∨-supdistribL to ⊓-⊔-supdistribL  -- : {Q R S : Carrier} → (Q ⊓ S) ⊔ (R ⊓ S) ⊑ ((Q ⊔ R) ⊓ S)
    ; ∧-∨-supdistribR to ⊓-⊔-supdistribR  -- : {Q R S : Carrier} → (Q ⊓ R) ⊔ (Q ⊓ S) ⊑ (Q ⊓ (R ⊔ S))
    ; ∨-∧-subdistribL to ⊔-⊓-supdistribL  -- : {Q R S : Carrier} → ((Q ⊓ R) ⊔ S) ⊑ (Q ⊔ S) ⊓ (R ⊔ S)
    ; ∨-∧-subdistribR to ⊔-⊓-supdistribR  -- : {Q R S : Carrier} → (Q ⊔ (R ⊓ S)) ⊑ (Q ⊔ R) ⊓ (Q ⊔ S)
    )

module RDistrLattice-square {j k₁ k₂ : Level} (𝒫 : Poset j k₁ k₂)
                               (meet : (R S : Poset.Carrier 𝒫) → PosetMeet.Meet 𝒫 R S)
                               (join : (R S : Poset.Carrier 𝒫) → PosetJoin.Join 𝒫 R S)
                               (∧-∨-subdistribR : let open Lattice 𝒫 meet join in
                                         {Q R S : Carrier} → (Q ∧ (R ∨ S)) ≤ (Q ∧ R) ∨ (Q ∧ S))
  where
  open RDistrLattice 𝒫 meet join ∧-∨-subdistribR using () renaming
    (∧-∨-distribR to ⊓-⊔-distribR   -- : {Q R S : Carrier} → (Q ⊓ (R ⊔ S)) ≈ (Q ⊓ R) ⊔ (Q ⊓ S)
    ; ∧-∨-distribL to ⊓-⊔-distribL  -- : {Q R S : Carrier} → ((Q ⊔ R) ⊓ S) ≈ (Q ⊓ S) ⊔ (R ⊓ S)
    ; ∨-∧-distribR to ⊔-⊓-distribR  -- : {Q R S : Carrier} → (Q ⊔ (R ⊓ S)) ≈ (Q ⊔ R) ⊓ (Q ⊔ S)
    ; ∨-∧-distribL to ⊔-⊓-distribL  -- : {Q R S : Carrier} → ((Q ⊓ R) ⊔ S) ≈ (Q ⊔ S) ⊓ (R ⊔ S)
    )
```

Chapter 9

Locally Ordered Semigroupoids and Categories

In this chapter, we add order structure to the individual hom-setoids. As already seen in Sect. 8.5, we consider the ordering as primitive and define joins and meets in terms of the ordering. Our definition of locally ordered semigroupoids already contains definitions and properties of meets that apply also in the case where not all meets exist. The corresponding properties for joins are obtained via switching to the dual ordering, as defined in Sect. 8.2.

We also include definitions for (semi-)lattice semigroupoids and categories.

9.1 Categoric.OrderedSemigroupoid

Since Poset from the standard library already has the equivalence relation and the ordering relation at potentially different Levels, we do not see any reason to disallow this difference for the local ordering in our semigroupoids.

Instead of with hom-setoids, we now deal with hom-posets; modules LocOrdBase and LocOrdCalc collect definitions and properties that involve only a single poset of morphisms.

In module LocOrd, we add monotony of composition and therewith relate the orderings in adjacent homsets.

```
module LocOrdBase {i j k₁ k₂ : Level} {Obj : Set i} (Hom : Obj → Obj → Poset j k₁ k₂) where
  open LocalHomSetoid (posetSetoid ∘₂ Hom)

  Hom≈ : Obj → Obj → Setoid j k₁
  Hom≈ = posetSetoid ∘₂ Hom

  Hom⊑ : Obj → Obj → Preorder j k₁ k₂
  Hom⊑ = Poset.preorder ∘₂ Hom

  module LocalHomPoset₀ {A B : Obj} where
    open Poset-square (Hom A B) public
  open LocalHomPoset₀ public


  isTop : {A B : Obj} → (f : Mor A B) → Set (j ⊍ k₂)
  isTop {A} {B} f = ∀ {g : Mor A B} → g ⊑ f

  isBottom : {A B : Obj} → (f : Mor A B) → Set (j ⊍ k₂)
  isBottom {A} {B} f = ∀ {g : Mor A B} → f ⊑ g

  ⊑-implies : {A B : Obj} {R S : Mor A B} → ({Q : Mor A B} → Q ⊑ R → Q ⊑ S) → R ⊑ S
  ⊑-implies {R = R} {S} F = F ⊑-refl
```

To save having to write **open** Pre (Hom A B) or similar for each calculational proof, we add A and B as implicit arguments to the calculation combinators of Relation.Binary.Poset.Calc.

Since LocOrdCalc exports a superset of the symbols of LocalSetoidCalc, all structures inheriting LocOrdCalc from OrderedSemigroupoid need to avoid inheriting also LocalSetoidCalc from LESGraph.

```
module LocOrdCalc {i j k₁ k₂ : Level} {Obj : Set i} (Hom : Obj → Obj → Poset j k₁ k₂) where
  module LocalLocOrdCalc {A B : Obj} where
```

```
    open PosetCalc-square (Hom A B) public
  open LocalLocOrdCalc public


record LocOrd {i j k₁ k₂ : Level} {Obj : Set i}
              (Hom : Obj → Obj → Poset j k₁ k₂)
              (compOp : CompOp (posetSetoid ∘₂ Hom))
              : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  open LocalHomSetoid    (posetSetoid ∘₂ Hom)
  open LocOrdBase    Hom
  open CompOpProps compOp
  field
    ⨾-monotone : {A B C : Obj} {f f' : Mor A B} {g g' : Mor B C}
               → f ⊑ f' → g ⊑ g' →   (f ⨾ g) ⊑ (f' ⨾ g')
  open LocOrdCalc Hom public
```

The following special cases of monotony make proofs shorter and therewith serve to improve readability in particular of calculational proofs.

```
  ⨾-monotone₁ : {A B C : Obj} {f f' : Mor A B} {g : Mor B C}
              → f ⊑ f' → (f ⨾ g) ⊑ (f' ⨾ g)
  ⨾-monotone₁ {A} {B} {C} inclF = ⨾-monotone inclF (Poset.refl (Hom B C))
  ⨾-monotone₂ : {A B C : Obj} {f : Mor A B} {g g' : Mor B C}
              → g ⊑ g' → (f ⨾ g) ⊑ (f ⨾ g')
  ⨾-monotone₂ {A} {B} {C} = ⨾-monotone (Poset.refl (Hom A B))
  ⨾-monotone₁₁ : {A B C D : Obj} {f f' : Mor A B} {g : Mor B C} {h : Mor C D}
                → f ⊑ f' →  ((f ⨾ g) ⨾ h) ⊑ ((f' ⨾ g) ⨾ h)
  ⨾-monotone₁₁ leq = ⨾-monotone₁ (⨾-monotone₁ leq)
  ⨾-monotone₁₂ : {A B C D : Obj} {f : Mor A B} {g g' : Mor B C} {h : Mor C D}
                → g ⊑ g' → ((f ⨾ g) ⨾ h) ⊑ ((f ⨾ g') ⨾ h)
  ⨾-monotone₁₂ leq = ⨾-monotone₁ (⨾-monotone₂ leq)
  ⨾-monotone₂₁ : {A B C D : Obj} {f : Mor A B} {g g' : Mor B C} {h : Mor C D}
                → g ⊑ g' → (f ⨾ g ⨾ h) ⊑ (f ⨾ (g' ⨾ h))
  ⨾-monotone₂₁ leq = ⨾-monotone₂ (⨾-monotone₁ leq)
  ⨾-monotone₂₂ : {A B C D : Obj} {f : Mor A B} {g : Mor B C} {h h' : Mor C D}
                → h ⊑ h' → (f ⨾ (g ⨾ h)) ⊑ (f ⨾ (g ⨾ h'))
  ⨾-monotone₂₂ leq = ⨾-monotone₂ (⨾-monotone₂ leq)
```

We can now define (co-)transitivity and subidentities and prove their basic properties:

```
  isTransitive : {A : Obj} → Mor A A → Set k₂
  isTransitive R = R ⨾ R ⊑ R
  isCotransitive : {A : Obj} → Mor A A → Set k₂
  isCotransitive R = R ⊑ R ⨾ R
  isLeftSubidentity : {A : Obj} → (p : Mor A A) → Set (i ⊍ j ⊍ k₂)
  isLeftSubidentity {A} p = {B : Obj} {R : Mor A B} → p ⨾ R ⊑ R
  isRightSubidentity : {A : Obj} → (p : Mor A A) → Set (i ⊍ j ⊍ k₂)
  isRightSubidentity {A} p = {B : Obj} {S : Mor B A} → S ⨾ p ⊑ S
  isSubidentity : {A : Obj} → (p : Mor A A) → Set (i ⊍ j ⊍ k₂)
  isSubidentity p = isLeftSubidentity p × isRightSubidentity p
  isLeftSuperidentity : {A : Obj} → (p : Mor A A) → Set (i ⊍ j ⊍ k₂)
  isLeftSuperidentity {A} p = {B : Obj} {R : Mor A B} → R ⊑ p ⨾ R
  isRightSuperidentity : {A : Obj} → (p : Mor A A) → Set (i ⊍ j ⊍ k₂)
  isRightSuperidentity {A} p = {B : Obj} {S : Mor B A} → S ⊑ S ⨾ p
  isSuperidentity : {A : Obj} → (p : Mor A A) → Set (i ⊍ j ⊍ k₂)
  isSuperidentity p = isLeftSuperidentity p × isRightSuperidentity p
```

```
⊑⊒-isLeftIdentity : {A : Obj} {q : Mor A A} → isLeftSubidentity q
                                             → isLeftSuperidentity q → isLeftIdentity q
⊑⊒-isLeftIdentity isLeftSubid isLeftSupid = ⊑-antisym isLeftSubid isLeftSupid
⊑⊒-isRightIdentity : {A : Obj} {q : Mor A A}  → isRightSubidentity q
                                               → isRightSuperidentity q → isRightIdentity q
⊑⊒-isRightIdentity isRightSubid isRightSupid = ⊑-antisym isRightSubid isRightSupid
⊑⊒-isIdentity : {A : Obj} {q : Mor A A} → isSubidentity q
                                         → isSuperidentity q → isIdentity q
⊑⊒-isIdentity (isLeftSubid, isRightSubid) (isLeftSupid, isRightSupid)
    = (λ {B} {R} → ⊑⊒-isLeftIdentity isLeftSubid isLeftSupid)
    , (λ {B} {R} → ⊑⊒-isRightIdentity isRightSubid isRightSupid)
isLeftIdentity-sub : {A : Obj} {q : Mor A A} → isLeftIdentity q → isLeftSubidentity q
isLeftIdentity-sub {A} {q} leftId {B} {R} = ⊑-reflexive leftId
isRightIdentity-sub : {A : Obj} {q : Mor A A} → isRightIdentity q → isRightSubidentity q
isRightIdentity-sub {A} {q} rightId {B} {R} = ⊑-reflexive rightId
isIdentity-sub : {A : Obj} {q : Mor A A} → isIdentity q → isSubidentity q
isIdentity-sub (L, R) = (λ {_} {_} → isLeftIdentity-sub L), (λ {_} {_} → isRightIdentity-sub R)
isLeftIdentity-super : {A : Obj} {q : Mor A A} → isLeftIdentity q → isLeftSuperidentity q
isLeftIdentity-super {A} {q} leftId {B} {R} = ⊑-reflexive′ leftId
isRightIdentity-super : {A : Obj} {q : Mor A A} → isRightIdentity q → isRightSuperidentity q
isRightIdentity-super {A} {q} rightId {B} {R} = ⊑-reflexive′ rightId
isIdentity-super : {A : Obj} {q : Mor A A} → isIdentity q → isSuperidentity q
isIdentity-super (L, R) = (λ {_} {_} → isLeftIdentity-super L), (λ {_} {_} → isRightIdentity-super R)
⨾-isLeftSubidentity  : {A : Obj} → {p q : Mor A A}
                     → isLeftSubidentity p → isLeftSubidentity q → isLeftSubidentity (p ⨾ q)
⨾-isLeftSubidentity {A} {p} {q} leftP leftQ {B} {R} = ⊑-begin
      (p ⨾ q) ⨾ R      ≈⟨ ⨾-assoc ⟩ p ⨾ (q ⨾ R)
                   ⊑⟨ leftP ⟩       q ⨾ R
                   ⊑⟨ leftQ ⟩       R □
⨾-isRightSubidentity     : {A : Obj} → {p q : Mor A A}
                         → isRightSubidentity p → isRightSubidentity q → isRightSubidentity (p ⨾ q)
⨾-isRightSubidentity {A} {p} {q} rightP rightQ {B} {S} = ⊑-begin
      S ⨾ (p ⨾ q)      ≈⟨ ⨾-assocL ⟩  (S ⨾ p) ⨾ q
                         ⊑⟨ rightQ ⟩ S ⨾ p
                         ⊑⟨ rightP ⟩ S □
⨾-isSubidentity    : {A : Obj} → {p q : Mor A A}
                   → isSubidentity p → isSubidentity q → isSubidentity (p ⨾ q)
⨾-isSubidentity {A} {p} {q} (leftP, rightP) (leftQ, rightQ)
   = (λ {B} {R} → ⨾-isLeftSubidentity {A} {p} {q} leftP leftQ {B} {R})
   , (λ {B} {S} → ⨾-isRightSubidentity {A} {p} {q} rightP rightQ {B} {S})
```

The two auxiliary functions swapFromSubid and swapToSubid will be used in particular in Sect. 10.2 to demonstrate that range defines a domain operator in the opposite ordered semigroupoid.

```
swapFromSubid : {A : Obj} {p : Mor A A}
              → isSubidentity p → isRightSubidentity p × isLeftSubidentity p
swapFromSubid (l, r) = (λ {_} {_} → r), (λ {_} {_} → l)
swapToSubid   : {A : Obj} {p : Mor A A}
              → isRightSubidentity p × isLeftSubidentity p → isSubidentity p
swapToSubid (r, l)  = (λ {_} {_} → l), (λ {_} {_} → r)
```

For superidentities, we choose to replicate the development instead of invoking duality:

```
⨾-isLeftSuperidentity       : {A : Obj} → {p q : Mor A A}
                            → isLeftSuperidentity p → isLeftSuperidentity q → isLeftSuperidentity (p ⨾ q)
⨾-isLeftSuperidentity {A} {p} {q} leftP leftQ {B} {R} = ⊑-begin
```

```
        R       ⊑⟨ leftQ ⟩      q ⨾ R
                ⊑⟨ leftP ⟩      p ⨾ (q ⨾ R)
                ≈⟨ ⨾-assocL ⟩ (p ⨾ q) ⨾ R □

  ⨾-isRightSuperidentity  : {A : Obj} → {p q : Mor A A}
              → isRightSuperidentity p → isRightSuperidentity q → isRightSuperidentity (p ⨾ q)
  ⨾-isRightSuperidentity {A} {p} {q} rightP rightQ {B} {S}  =  ⊑-begin
        S               ⊑⟨ rightP ⟩      S ⨾ p
                        ⊑⟨ rightQ ⟩      (S ⨾ p) ⨾ q
                        ≈⟨ ⨾-assoc ⟩     S ⨾ (p ⨾ q) □

  ⨾-isSuperidentity  : {A : Obj} → {p q : Mor A A}
              → isSuperidentity p → isSuperidentity q → isSuperidentity (p ⨾ q)
  ⨾-isSuperidentity (leftP, rightP) (leftQ, rightQ)
    =  (λ {B} {R} → ⨾-isLeftSuperidentity leftP leftQ {B} {R})
    , (λ {B} {S} → ⨾-isRightSuperidentity rightP rightQ {B} {S})

  ⊑-isLeftSubidentity : {A : Obj} → {p q : Mor A A}
              → p ⊑ q → isLeftSubidentity q → isLeftSubidentity p
  ⊑-isLeftSubidentity {A} {p} {q} pq left {B} {R}  =  ⊑-begin
        p ⨾ R       ⊑⟨ ⨾-monotone₁ pq ⟩          q ⨾ R
                    ⊑⟨ left ⟩                    R □

  ⊑-isRightSubidentity     : {A : Obj} → {p q : Mor A A}
              → p ⊑ q → isRightSubidentity q → isRightSubidentity p
  ⊑-isRightSubidentity {A} {p} {q} pq right {B} {S}  =  ⊑-begin
        S ⨾ p           ⊑⟨ ⨾-monotone₂ pq ⟩      S ⨾ q
                        ⊑⟨ right ⟩               S    □

  ⊑-isSubidentity : {A : Obj} → {p q : Mor A A} → p ⊑ q → isSubidentity q → isSubidentity p
  ⊑-isSubidentity pq (left, right)  =  (λ {B} {R} → ⊑-isLeftSubidentity pq left {B} {R})
                                     , (λ {B} {S} → ⊑-isRightSubidentity pq right {B} {S})

  ≈-isLeftSubidentity : {A : Obj} → {p q : Mor A A}
              → q ≈ p → isLeftSubidentity p → isLeftSubidentity q
  ≈-isLeftSubidentity qp  =  ⊑-isLeftSubidentity (⊑-reflexive qp)

  ≈-isRightSubidentity     : {A : Obj} → {p q : Mor A A}
              → q ≈ p → isRightSubidentity p → isRightSubidentity q
  ≈-isRightSubidentity qp  =  ⊑-isRightSubidentity (⊑-reflexive qp)

  ≈-isSubidentity : {A : Obj} → {p q : Mor A A} → q ≈ p → isSubidentity p → isSubidentity q
  ≈-isSubidentity qp  =  ⊑-isSubidentity (⊑-reflexive qp)

  ⊑-isLeftSuperidentity        : {A : Obj} → {p q : Mor A A}
              → p ⊑ q → isLeftSuperidentity p → isLeftSuperidentity q
  ⊑-isLeftSuperidentity {A} {p} {q} pq left {B} {R}  =  ⊑-begin
        R               ⊑⟨ left ⟩                p ⨾ R
                        ⊑⟨ ⨾-monotone₁ pq ⟩     q ⨾ R □

  ⊑-isRightSuperidentity : {A : Obj} → {p q : Mor A A}
              → p ⊑ q → isRightSuperidentity p → isRightSuperidentity q
  ⊑-isRightSuperidentity {A} {p} {q} pq right {B} {S}  =  ⊑-begin
        S               ⊑⟨ right ⟩               S ⨾ p
                        ⊑⟨ ⨾-monotone₂ pq ⟩ S ⨾ q □

  ⊑-isSuperidentity : {A : Obj} → {p q : Mor A A}
              → p ⊑ q → isSuperidentity p → isSuperidentity q
  ⊑-isSuperidentity pq (left, right)
    =  (λ {B} {R} → ⊑-isLeftSuperidentity pq left {B} {R})
    , (λ {B} {S} → ⊑-isRightSuperidentity pq right {B} {S})

  ≈-isLeftSuperidentity        : {A : Obj} → {p q : Mor A A}
              → q ≈ p → isLeftSuperidentity p → isLeftSuperidentity q
  ≈-isLeftSuperidentity qp  =  ⊑-isLeftSuperidentity (⊑-reflexive (≈-sym qp))

  ≈-isRightSuperidentity : {A : Obj} → {p q : Mor A A}
              → q ≈ p → isRightSuperidentity p → isRightSuperidentity q
  ≈-isRightSuperidentity qp  =  ⊑-isRightSuperidentity (⊑-reflexive (≈-sym qp))
```

```
≈-isSuperidentity : {A : Obj} → {p q : Mor A A}
                        → q ≈ p → isSuperidentity p → isSuperidentity q
≈-isSuperidentity qp = ⊑-isSuperidentity (⊑-reflexive (≈-sym qp))
```

Idempotent sub-identities play a special rôle as domain elements (see Sect. 10.2), so we define a separate type for them:

```
record isISId {A : Obj} (p : Mor A A) : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  field
    subid : isSubidentity p
    idempot : isIdempotent p
  leftSubid : isLeftSubidentity p
  leftSubid = proj₁ subid
  rightSubid : isRightSubidentity p
  rightSubid = proj₂ subid
  transitive : isTransitive p
  transitive = ⊑-reflexive idempot
  cotransitive : isCotransitive p
  cotransitive = ⊑-reflexive′ idempot
record ISId {A : Obj} : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  field
    mor : Mor A A
    proof : isISId mor
  open isISId proof public
ISId-poset : {A : Obj} → Poset (i ⊍ j ⊍ k₁ ⊍ k₂) k₁ k₂
ISId-poset {A} = record
  {Carrier = ISId {A}
  ; _≈_ = λ p q → ISId.mor p ≈ ISId.mor q
  ; _≤_ = λ p q → ISId.mor p ⊑ ISId.mor q
  ; isPartialOrder = record
    {isPreorder = record
      {isEquivalence = record
            {refl    = ≈-refl
            ; sym    = ≈-sym
            ; trans  = ≈-trans
            }
      ; reflexive      = ⊑-reflexive
      ; trans          = ⊑-trans
      }
    ; antisym = ⊑-antisym
    }
  }
```

The interfaces for calculational reasoning:

```
ISId⊑ : {A : Obj} → Preorder (i ⊍ j ⊍ k₁ ⊍ k₂) k₁ k₂
ISId⊑ {A} = Poset.preorder (ISId-poset {A})
ISId≈ : {A : Obj} → Setoid (i ⊍ j ⊍ k₁ ⊍ k₂) k₁
ISId≈ {A} = posetSetoid (ISId-poset {A})
```

The composition of commuting dempotent sub-identities is an idempotent sub-identity again:

```
commuting-isISId-⨾ : {A : Obj} {p q : Mor A A}
                         → p ⨾ q ≈ q ⨾ p → isISId p → isISId q → isISId (p ⨾ q)
commuting-isISId-⨾ {_} {p} {q} p⨾q≈q⨾p pISId qISId = record
  {subid = ⨾-isSubidentity (isISId.subid pISId) (isISId.subid qISId)
  ; idempot = ≈-begin
      (p ⨾ q) ⨾ (p ⨾ q)
```

```
      ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⨾-assocL ⟩
        p ⨾ (q ⨾ p) ⨾ q
      ≈˘⟨ ⨾-cong₂₁ p⨾q≈q⨾p ⟩
        p ⨾ (p ⨾ q) ⨾ q
      ≈⟨ ⨾-cong₂ (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ (isISId.idempot qISId)) ⟩
        p ⨾ (p ⨾ q)
      ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (isISId.idempot pISId) ⟩
        p ⨾ q
      □
    }
  commuting-ISId-⨾     : {A : Obj} (P Q : ISId {A})
                        → let p = ISId.mor P; q = ISId.mor Q
                        in p ⨾ q ≈ q ⨾ p
                        → ISId {A}
  commuting-ISId-⨾ {A} P Q p⨾q≈q⨾p = record
    {mor = ISId.mor P ⨾ ISId.mor Q
    ; proof = commuting-isISId-⨾ p⨾q≈q⨾p (ISId.proof P) (ISId.proof Q)
    }

  open LocOrdBase Hom public

retractLocOrd : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
              → (F : Obj₂ → Obj₁)
              → {Hom : Obj₁ → Obj₁ → Poset j k₁ k₂}
                {compOp : CompOp (posetSetoid ∘₂ Hom)}
              → LocOrd Hom compOp → LocOrd (λ A B → Hom (F A) (F B)) (retractCompOp F compOp)
retractLocOrd F locOrd = let open LocOrd locOrd in record {⨾-monotone = ⨾-monotone}

Attach-⊑ : {i j k₁ k₂ : Level} {Obj : Set i} (Hom : Obj → Obj → Poset j k₁ k₂)
         → {x y : Obj} →   Rel (Attach (posetSetoid ∘₂ Hom) x y) k₂
Attach-⊑ Hom {x} {y} a₁ a₂ = Poset._≤_ (Hom x y) (edgeᵃ a₁) (edgeᵃ a₂)

attachLocalPoset : {i j k₁ k₂ : Level} {Obj : Set i}
                 → (Obj → Obj → Poset j k₁ k₂)
                 → (Obj → Obj → Poset (i ⊍ j) k₁ k₂)
attachLocalPoset Hom x y = let LES = posetSetoid ∘₂ Hom; open LocOrdBase Hom in record
  {Carrier = Attach LES x y
  ; _≈_    = Attach-≈
  ; _≤_    = Attach-⊑ Hom
  ; isPartialOrder = record
    {isPreorder = record
      {isEquivalence = Setoid.isEquivalence (attachLES LES x y)
      ; reflexive = ⊑-reflexive
      ; trans = ⊑-trans
      }
    ; antisym = ⊑-antisym
    }
  }

attachLocOrd : {i j k₁ k₂ : Level} {Obj : Set i} {Hom : Obj → Obj → Poset j k₁ k₂}
               {compOp : CompOp (posetSetoid ∘₂ Hom)}
  →            LocOrd Hom compOp → LocOrd (attachLocalPoset Hom) (attachCompOp compOp)
attachLocOrd locOrd = let open LocOrd locOrd in record
  {⨾-monotone = ⨾-monotone
  }
```

In locally ordered semigroupoids, both categoric duality (opposite-) and order duality (dual-) apply:

```
oppositeLocOrd : {i j k₁ k₂ : Level} {Obj : Set i} {Hom : Obj → Obj → Poset j k₁ k₂}
                 {compOp : CompOp (posetSetoid ∘₂ Hom)}
  →              LocOrd Hom compOp → LocOrd (λ A B → Hom B A) (oppositeCompOp compOp)
oppositeLocOrd locOrd = let open LocOrd locOrd in record
  {⨾-monotone = λ f⊑f' g⊑g' → ⨾-monotone g⊑g' f⊑f'
  }
```

```
dualLocOrd : {i j k₁ k₂ : Level} {Obj : Set i} {Hom : Obj → Obj → Poset j k₁ k₂}
             {compOp : CompOp (posetSetoid ∘₂ Hom)}
           → LocOrd Hom compOp → LocOrd (dualPoset ∘₂ Hom) compOp
dualLocOrd locOrd = let open LocOrd locOrd in record {⨾-monotone = ⨾-monotone}
```

```
record OrderedSemigroupoid {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field
    Hom : Obj → Obj → Poset j k₁ k₂
    compOp : CompOp (posetSetoid ∘₂ Hom)
    locOrd : LocOrd Hom compOp
  open LocOrd locOrd   public
  semigroupoid : Semigroupoid j k₁ Obj
  semigroupoid = record
    {Hom = Hom≈
    ;compOp = compOp
    }
  open LocalHomSetoid Hom≈   public
  open CompOpProps   compOp public
```

```
retractOrderedSemigroupoid : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                           → (F : Obj₂ → Obj₁)
                           → OrderedSemigroupoid j k₁ k₂ Obj₁
                           → OrderedSemigroupoid j k₁ k₂ Obj₂
retractOrderedSemigroupoid F osg = let open OrderedSemigroupoid osg in record
  {Hom    = λ A B → Hom (F A) (F B)
  ;compOp = retractCompOp F compOp
  ;locOrd = retractLocOrd F locOrd
  }
```

```
retractIsSubidentity : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                     → (F : Obj₂ → Obj₁)
                     → (OSG₁ : OrderedSemigroupoid j k₁ k₂ Obj₁)
                     → let OSG₂ = retractOrderedSemigroupoid F OSG₁ in
                        {A : Obj₂} {R : OrderedSemigroupoid.Mor OSG₁ (F A) (F A)}
                     → OrderedSemigroupoid.isSubidentity OSG₁ {F A} R
                     → OrderedSemigroupoid.isSubidentity OSG₂ {A} R
retractIsSubidentity F base (left, right) = (λ {Z} {S} → left), (λ {Z} {Q} → right)
```

A function reflectIsSubidentity for the opposite direction would need an inverse of F.

```
attachOrderedSemigroupoid : {i j k₁ k₂ : Level} {Obj : Set i}
                          → OrderedSemigroupoid j k₁ k₂ Obj
                          → OrderedSemigroupoid (i ⊍ j) k₁ k₂ Obj
attachOrderedSemigroupoid osg = let open OrderedSemigroupoid osg in record
  {Hom    = attachLocalPoset Hom
  ;compOp = attachCompOp compOp
```

```
  ; locOrd   = attachLocOrd locOrd
  }


oppositeOrderedSemigroupoid : {i j k₁ k₂ : Level} {Obj : Set i}
                            → OrderedSemigroupoid j k₁ k₂ Obj
                            → OrderedSemigroupoid j k₁ k₂ Obj
oppositeOrderedSemigroupoid osg = let open OrderedSemigroupoid osg in record
  {Hom      = λ A B → Hom B A
  ; compOp  = oppositeCompOp compOp
  ; locOrd  = oppositeLocOrd locOrd
  }


dualOrderedSemigroupoid : {i j k₁ k₂ : Level} {Obj : Set i}
                        → OrderedSemigroupoid j k₁ k₂ Obj
                        → OrderedSemigroupoid j k₁ k₂ Obj
dualOrderedSemigroupoid osg = let open OrderedSemigroupoid osg in record
  {Hom      = dualPoset ∘₂ Hom
  ; compOp  = compOp
  ; locOrd  = dualLocOrd locOrd
  }
```

9.2 Categoric.OrderedCategory

Since ordered categories are directly based on ordered semigroupoids, we have to explicityly invoke CategoryProps again to inherit also the complete Category theory.

For the new properties here, we define the module OC-Props inside OrderedCategory, instead of outside as we did with CategoryProps — this has the advantage that it saves the hassles of properly extracting the necessary arguments, and the disadvantage that a full OrderedCategory needs to be defined before its OC-Props becomes accessible. We do not yet see a clear-cut reason to prefer one approach over the other, and therefore experiment with both.

```
record OrderedCategory {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field orderedSemigroupoid : OrderedSemigroupoid j k₁ k₂ Obj
  open OrderedSemigroupoid orderedSemigroupoid
  field idOp : IdOp (posetSetoid ∘₂ Hom) _⨾_
  category : Category j k₁ Obj
  category = record
    {semigroupoid = semigroupoid
    ; idOp = idOp
    }
  open IdOp                       idOp public
  open CategoryProps semigroupoid idOp public
  open OppositeIsos   semigroupoid idOp public


  module OC-Props where    -- sub-module for selective open


    isCoreflexive : {A : Obj} → Mor A A → Set k₂
    isCoreflexive {A} P = P ⊑ Id
    ⊑-isCoreflexive :    {A : Obj} → {P Q : Mor A A} → Q ⊑ P → isCoreflexive P → isCoreflexive Q
    ⊑-isCoreflexive = ⊑-trans
    ≈-isCoreflexive :    {A : Obj} → {P Q : Mor A A} → Q ≈ P → isCoreflexive P → isCoreflexive Q
    ≈-isCoreflexive qp = ⊑-isCoreflexive (⊑-reflexive qp)
```

```
    coreflexiveIsSubidentity : {A : Obj} → {P : Mor A A} → isCoreflexive P → isSubidentity P
    coreflexiveIsSubidentity {A} {P} cor
      = (λ {B} {R} → ⊑-trans (⨾-monotone₁ cor) (⊑-reflexive leftId))
      , (λ {B} {S} → ⊑-trans (⨾-monotone₂ cor) (⊑-reflexive rightId))

    subidentityIsCoreflexive : {A : Obj} → {P : Mor A A} → isSubidentity P → isCoreflexive P
    subidentityIsCoreflexive {A} {P} (left, _) = ⊑-trans (⊑-reflexive (≈-sym rightId)) left

    isReflexive : {A : Obj} → Mor A A → Set k₂
    isReflexive {A} R = Id ⊑ R

    ⊑-isReflexive : {A : Obj} → {P Q : Mor A A} → P ⊑ Q → isReflexive P → isReflexive Q
    ⊑-isReflexive pq refP = ⊑-trans refP pq

    ≈-isReflexive : {A : Obj} → {P Q : Mor A A} → Q ≈ P → isReflexive P → isReflexive Q
    ≈-isReflexive qp = ⊑-isReflexive (⊑-reflexive (≈-sym qp))

    reflexiveIsSuperidentity : {A : Obj} → {P : Mor A A} → isReflexive P → isSuperidentity P
    reflexiveIsSuperidentity {A} {P} ref
      = (λ {B} {R} → ⊑-trans (⊑-reflexive (≈-sym leftId)) (⨾-monotone₁ ref))
      , (λ {B} {S} → ⊑-trans (⊑-reflexive (≈-sym rightId)) (⨾-monotone₂ ref))

    superidentityIsReflexive : {A : Obj} → {P : Mor A A} → isSuperidentity P → isReflexive P
    superidentityIsReflexive {A} {P} (left, _) = ⊑-trans left (⊑-reflexive rightId)

  open OC-Props public
  open OrderedSemigroupoid orderedSemigroupoid public

retractOrderedCategory : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                       → (F : Obj₂ → Obj₁)
                       → OrderedCategory j k₁ k₂ Obj₁ → OrderedCategory j k₁ k₂ Obj₂
retractOrderedCategory F oc = let open OrderedCategory oc in record
  {orderedSemigroupoid = retractOrderedSemigroupoid F orderedSemigroupoid
  ; idOp = retractIdOp F idOp
  }

oppositeOrderedCategory : {i j k₁ k₂ : Level} {Obj : Set i}
                        → OrderedCategory j k₁ k₂ Obj → OrderedCategory j k₁ k₂ Obj
oppositeOrderedCategory oc = let open OrderedCategory oc in record
  {orderedSemigroupoid = oppositeOrderedSemigroupoid orderedSemigroupoid
  ; idOp = oppositeIdOp idOp
  }

dualOrderedCategory : {i j k₁ k₂ : Level} {Obj : Set i}
                    → OrderedCategory j k₁ k₂ Obj → OrderedCategory j k₁ k₂ Obj
dualOrderedCategory oc = let open OrderedCategory oc in record
  {orderedSemigroupoid = dualOrderedSemigroupoid orderedSemigroupoid
  ; idOp = idOp
  }
```

9.3 Categoric.OrderedSemigroupoid.Lattice

Here we add modules LocOrdMeet and LocOrdJoin in the context of an ordered semigroupoid, where in general not all meets and joins exist.

```
module LocOrdMeet {i j k₁ k₂ : Level} {Obj : Set i} (Hom : Obj → Obj → Poset j k₁ k₂) where
  open LocalHomSetoid (posetSetoid ∘₂ Hom)
  open LocOrdBase Hom
```

```
private
  module LocOrdMeet′ {A B : Obj} where
    open PosetMeet (Hom A B) public renaming
      (≤-from-IsMeet₁ to ⊑-from-IsMeet₁   -- {R S M : Mor A B} → IsMeet R S M → M ≈ R → R ⊑ S
      ; ≤-from-IsMeet₂ to ⊑-from-IsMeet₂  -- {R S M : Mor A B} → IsMeet R S M → M ≈ S → S ⊑ R
      ; ≤-from-Meet₁   to ⊑-from-Meet₁    -- {R S : Mor A B} → (R⊓S : Meet R S)
                                          -- → Meet.mor R⊓S ≈ R → R ⊑ S
      ; ≤-from-Meet₂   to ⊑-from-Meet₂    -- {R S : Mor A B} → (R⊓S : Meet R S)
                                          -- → Meet.mor R⊓S ≈ S → S ⊑ R
      ; ≤-to-IsMeet₁   to ⊑-to-IsMeet₁    -- {R S : Mor A B} → R ⊑ S → IsMeet R S R
      ; ≤-to-IsMeet₂   to ⊑-to-IsMeet₂    -- {R S : Mor A B} → S ⊑ R → IsMeet R S S
      ; ≤-to-IsMeet₁≈  to ⊑-to-IsMeet₁≈   -- {R S M : Mor A B} → IsMeet R S M → R ⊑ S → M ≈ R
      ; ≤-to-IsMeet₂≈  to ⊑-to-IsMeet₂≈   -- {R S M : Mor A B} → IsMeet R S M → S ⊑ R → M ≈ S
      ; ≤-to-Meet₁     to ⊑-to-Meet₁      -- {R S : Mor A B} → R ⊑ S → Meet R S
      ; ≤-to-Meet₂     to ⊑-to-Meet₂      -- {R S : Mor A B} → S ⊑ R → Meet R S
      )
open LocOrdMeet′ public
```

We defines joins and their properties as the duals of meets and their properties — although this saves almost two pages of explicitly dualised definitions, it also means that the join properties are not explicitly stated, and require either translation while reading the corresponding meet properties, or tool support to display the resulting types of the members of the LocOrdJoin module. (If we forget to rename an item defined in LocOrdMeet, Agda will complain about duplicate definitions below in the definition of LocOrd where both LocOrdMeet and LocOrdJoin are **open**ed.)

```
module LocOrdJoin {i j k₁ k₂ : Level} {Obj : Set i} (Hom : Obj → Obj → Poset j k₁ k₂) where
  private
    module LocOrdJoin′ {A B : Obj} where
      open PosetJoin (Hom A B) public renaming
        (≤-from-IsJoin₁ to ⊑-from-IsJoin₁   -- {R S M : Mor A B} → IsJoin R S M → M ≈ R → S ⊑ R
        ; ≤-from-IsJoin₂ to ⊑-from-IsJoin₂  -- {R S M : Mor A B} → IsJoin R S M → M ≈ S → R ⊑ S
        ; ≤-from-Join₁   to ⊑-from-Join₁    -- {R S : Mor A B} → (R⊔S : Join R S)
                                            -- → Join.mor R⊔S ≈ R → S ⊑ R
        ; ≤-from-Join₂   to ⊑-from-Join₂    -- {R S : Mor A B} → (R⊔S : Join R S)
                                            -- → Join.mor R⊔S ≈ S → R ⊑ S
        ; ≤-to-IsJoin₁   to ⊑-to-IsJoin₁    -- {R S : Mor A B} → S ⊑ R → IsJoin R S R
        ; ≤-to-IsJoin₂   to ⊑-to-IsJoin₂    -- {R S : Mor A B} → R ⊑ S → IsJoin R S S
        ; ≤-to-IsJoin₁≈  to ⊑-to-IsJoin₁≈   -- {R S M : Mor A B} → IsJoin R S M → S ⊑ R → M ≈ R
        ; ≤-to-IsJoin₂≈  to ⊑-to-IsJoin₂≈   -- {R S M : Mor A B} → IsJoin R S M → R ⊑ S → M ≈ S
        ; ≤-to-Join₁     to ⊑-to-Join₁      -- {R S : Mor A B} → S ⊑ R → Join R S
        ; ≤-to-Join₂     to ⊑-to-Join₂      -- {R S : Mor A B} → R ⊑ S → Join R S
        )
  open LocOrdJoin′ public
```

In the poset of ISIds (but not in general in Hom A A), composition of commuting ISIds produces meets:

```
module ISId-Meet {i j k₁ k₂ : Level} {Obj : Set i}
                 (OSG : OrderedSemigroupoid j k₁ k₂ Obj) where
  open OrderedSemigroupoid OSG
  commuting-ISId-meet : {A : Obj} (P Q : ISId {A})
                      → let p = ISId.mor P; q = ISId.mor Q
                        in (p⨾q≈q⨾p : p ⨾ q ≈ q ⨾ p)
                      → PosetMeet.IsMeet (ISId-poset {A}) P Q (commuting-ISId-⨾ P Q p⨾q≈q⨾p)
  commuting-ISId-meet {A} P Q p⨾q≈q⨾p = let open ISId in record
    {bound₁ = rightSubid Q
    ; bound₂ = leftSubid P
    ; universal = λ {X} X⊑P X⊑Q → ⊑-begin
        mor X
```

```
    ≈˘⟨ idempot X ⟩
      mor X ⨾ mor X
    ⊑⟨ ⨾-monotone X⊑P X⊑Q ⟩
      mor P ⨾ mor Q
    □
  }
```

9.4 Categoric.LSLSemigroupoid

In **lower-semilattice semigroupoids**, meets are defined on each hom-poset. We derive the meet operator _⊓_ from a function meet that produces complete Meet records.

```
record MeetOp {i j k₁ k₂ : Level} {Obj : Set i}
              (OSG : OrderedSemigroupoid j k₁ k₂ Obj)
              : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  open OrderedSemigroupoid OSG
  open LocOrdMeet Hom
  field
    meet : {A B : Obj} → (R S : Mor A B) → Meet R S
  module HomLowerSemilattice {A B : Obj} where
    open LowerSemilattice-square (Hom A B) meet public
  open HomLowerSemilattice public
  ⨾-⊓-subdistribL : {A B C : Obj} {R₁ R₂ : Mor A B} {S : Mor B C}
    → ((R₁ ⊓ R₂) ⨾ S) ⊑ (R₁ ⨾ S) ⊓ (R₂ ⨾ S)
  ⨾-⊓-subdistribL = ⊓-universal (⨾-monotone₁ ⊓-lower₁) (⨾-monotone₁ ⊓-lower₂)
  ⨾-⊓-subdistribR : {A B C : Obj} {R : Mor A B} {S₁ S₂ : Mor B C}
    → (R ⨾ (S₁ ⊓ S₂)) ⊑ (R ⨾ S₁) ⊓ (R ⨾ S₂)
  ⨾-⊓-subdistribR = ⊓-universal (⨾-monotone₂ ⊓-lower₁) (⨾-monotone₂ ⊓-lower₂)
```

```
retractMeetOp : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
              → (F : Obj₂ → Obj₁)
              → {OSG : OrderedSemigroupoid j k₁ k₂ Obj₁}
              → MeetOp OSG → MeetOp (retractOrderedSemigroupoid F OSG)
retractMeetOp F meetOp = record {meet = MeetOp.meet meetOp}
```

```
record LSLSemigroupoid {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field orderedSemigroupoid : OrderedSemigroupoid j k₁ k₂ Obj
  field meetOp              : MeetOp orderedSemigroupoid
  open OrderedSemigroupoid orderedSemigroupoid public
  open LocOrdMeet       Hom          public
  open MeetOp           meetOp       public
```

```
retractLSLSemigroupoid : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                       → (F : Obj₂ → Obj₁)
                       → LSLSemigroupoid j k₁ k₂ Obj₁ → LSLSemigroupoid j k₁ k₂ Obj₂
retractLSLSemigroupoid F base = let open LSLSemigroupoid base in record
  {orderedSemigroupoid = retractOrderedSemigroupoid F orderedSemigroupoid
  ;meetOp = retractMeetOp F meetOp
  }
```

9.5 Categoric.USLSemigroupoid

In **upper-semilattice semigroupoids**, all joins exist in each hom-poset. For readability, we do not invoke duality, but replicate the development of MeetOp.

Recently, there has been increased interest in joins over which composition does not distribute from both sides; we define different theories accordingly.

We use "⊔" as the join operator on morphisms of locally ordered semigroupoids; we first rename the upper semilattice material accordingly and add lemmas and definitions that relate join with composition without further assumptions.

```
record JoinOp {i j k₁ k₂ : Level} {Obj : Set i}
              (OSG : OrderedSemigroupoid j k₁ k₂ Obj)
              : Set (i ⊍ j ⊍ k₁ ⊍ k₂)
  where
  open OrderedSemigroupoid OSG
  open LocOrdJoin Hom
  field
    join : {A B : Obj} → (R S : Mor A B) → Join R S


  module HomUpperSemilattice {A B : Obj} where
    open UpperSemilattice-square (Hom A B) join public
  open HomUpperSemilattice public


  ⨾-⊔-supdistribL : {A B C : Obj} {R₁ R₂ : Mor A B} {S : Mor B C}
                  → (R₁ ⨾ S) ⊔ (R₂ ⨾ S) ⊑ ((R₁ ⊔ R₂) ⨾ S)
  ⨾-⊔-supdistribL = ⊔-universal (⨾-monotone₁ ⊔-upper₁) (⨾-monotone₁ ⊔-upper₂)
  ⨾-⊔-supdistribR : {A B C : Obj} {R : Mor A B} {S₁ S₂ : Mor B C}
                  → (R ⨾ S₁) ⊔ (R ⨾ S₂) ⊑ (R ⨾ (S₁ ⊔ S₂))
  ⨾-⊔-supdistribR = ⊔-universal (⨾-monotone₂ ⊔-upper₁) (⨾-monotone₂ ⊔-upper₂)


  ⊔-isLeftSuperidentity : {A : Obj} {p q : Mor A A}
    → isLeftSuperidentity p → isLeftSuperidentity q → isLeftSuperidentity (p ⊔ q)
  ⊔-isLeftSuperidentity {A} {p} {q} pL qL {B} {R} = ⊑-begin
      R
    ≈˘⟨ ⊔-idempotent ⟩
      R ⊔ R
    ⊑⟨ ⊔-monotone pL qL ⟩
      p ⨾ R ⊔ q ⨾ R
    ⊑⟨ ⨾-⊔-supdistribL ⟩
      (p ⊔ q) ⨾ R
    □
  ⊔-isRightSuperidentity : {A : Obj} {p q : Mor A A}
    → isRightSuperidentity p → isRightSuperidentity q → isRightSuperidentity (p ⊔ q)
  ⊔-isRightSuperidentity {A} {p} {q} pR qR {B} {R} = ⊑-begin
      R
    ≈˘⟨ ⊔-idempotent ⟩
      R ⊔ R
    ⊑⟨ ⊔-monotone pR qR ⟩
      R ⨾ p ⊔ R ⨾ q
    ⊑⟨ ⨾-⊔-supdistribR ⟩
      R ⨾ (p ⊔ q)
    □
  ⊔-isSuperidentity : {A : Obj} {p q : Mor A A}
    → isSuperidentity p → isSuperidentity q → isSuperidentity (p ⊔ q)
```

```agda
  ⊔-isSuperidentity (pL , pR) (qL , qR) = (λ {_} {_} → ⊔-isLeftSuperidentity pL qL)
                                         , (λ {_} {_} → ⊔-isRightSuperidentity pR qR)

  ⊔-⨾-⊔-par : {A B C : Obj} {R₁ R₂ : Mor A B} {S₁ S₂ : Mor B C}
    → (R₁ ⨾ S₁) ⊔ (R₂ ⨾ S₂) ⊑ ((R₁ ⊔ R₂) ⨾ (S₁ ⊔ S₂))
  ⊔-⨾-⊔-par {_} {_} {_} {R₁} {R₂} {S₁} {S₂} = ⊑-begin
      (R₁ ⨾ S₁) ⊔ (R₂ ⨾ S₂)
    ⊑⟨ ⊔-monotone (⨾-monotone₁ ⊔-upper₁) (⨾-monotone₁ ⊔-upper₂) ⟩
      (R₁ ⊔ R₂) ⨾ S₁ ⊔ (R₁ ⊔ R₂) ⨾ S₂
    ⊑⟨ ⨾-⊔-supdistribR ⟩
      (R₁ ⊔ R₂) ⨾ (S₁ ⊔ S₂)
    □

  _isLeftSubidUpto_ : {A : Obj} (I : Mor A A) (W : Mor A A) → Set (i ⊍ j ⊍ k₂)
  _isLeftSubidUpto_ {A} I W = {B : Obj} {R : Mor A B} → I ⨾ R ⊑ R ⊔ W ⨾ R
  _isRightSubidUpto_ : {A : Obj} (I : Mor A A) (W : Mor A A) → Set (i ⊍ j ⊍ k₂)
  _isRightSubidUpto_ {A} I W = {Z : Obj} {S : Mor Z A} → S ⨾ I ⊑ S ⊔ S ⨾ W
  _isSubidUpto_ : {A : Obj} (I : Mor A A) (W : Mor A A) → Set (i ⊍ j ⊍ k₂)
  I isSubidUpto W = (I isLeftSubidUpto W) × (I isRightSubidUpto W)

retractJoinOp : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
              → (F : Obj₂ → Obj₁)
              → {OSG : OrderedSemigroupoid j k₁ k₂ Obj₁}
              → JoinOp OSG → JoinOp (retractOrderedSemigroupoid F OSG)
retractJoinOp F joinOp = record {join = JoinOp.join joinOp}

record RawUSLSemigroupoid {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field orderedSemigroupoid : OrderedSemigroupoid j k₁ k₂ Obj
  field joinOp : JoinOp orderedSemigroupoid
  open OrderedSemigroupoid orderedSemigroupoid public
  open LocOrdJoin          Hom                 public
  open JoinOp              joinOp              public

retractRawUSLSemigroupoid : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                          → (F : Obj₂ → Obj₁)
                          → RawUSLSemigroupoid j k₁ k₂ Obj₁ → RawUSLSemigroupoid j k₁ k₂ Obj₂
retractRawUSLSemigroupoid F base = let open RawUSLSemigroupoid base in record
  {orderedSemigroupoid = retractOrderedSemigroupoid F orderedSemigroupoid
  ;joinOp = retractJoinOp F joinOp
  }

record JoinCompDistrL {i j k₁ k₂ : Level} {Obj : Set i}
                      {OSG : OrderedSemigroupoid j k₁ k₂ Obj}
                      (joinOp : JoinOp OSG)
                      : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  open OrderedSemigroupoid OSG
  open JoinOp joinOp
  field
    ⨾-⊔-subdistribL : {A B C : Obj} {R₁ R₂ : Mor A B} {S : Mor B C}
                    → ((R₁ ⊔ R₂) ⨾ S) ⊑ (R₁ ⨾ S) ⊔ (R₂ ⨾ S)
  ⨾-⊔-distribL : {A B C : Obj} {R₁ R₂ : Mor A B} {S : Mor B C}
               → ((R₁ ⊔ R₂) ⨾ S) ≈ (R₁ ⨾ S) ⊔ (R₂ ⨾ S)
  ⨾-⊔-distribL = ⊑-antisym ⨾-⊔-subdistribL ⨾-⊔-supdistribL
```

```
  ⊔-isLeftSubidentity : {A : Obj} {p q : Mor A A}
    → isLeftSubidentity p → isLeftSubidentity q → isLeftSubidentity (p ⊔ q)
  ⊔-isLeftSubidentity {A} {p} {q} pL qL {B} {R} = ⊑-begin
       (p ⊔ q) ⨾ R
    ⊑⟨ ⨾-⊔-subdistribL ⟩
       p ⨾ R ⊔ q ⨾ R
    ⊑⟨ ⊔-monotone pL qL ⟩
       R ⊔ R
    ≈⟨ ⊔-idempotent ⟩
       R
    □

record JoinCompDistrR {i j k₁ k₂ : Level} {Obj : Set i}
                      {OSG : OrderedSemigroupoid j k₁ k₂ Obj}
                      (joinOp : JoinOp OSG)
                      : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  open OrderedSemigroupoid OSG
  open JoinOp joinOp
  field
    ⨾-⊔-subdistribR : {A B C : Obj} {R : Mor A B} {S₁ S₂ : Mor B C}
                    → (R ⨾ (S₁ ⊔ S₂)) ⊑ (R ⨾ S₁) ⊔ (R ⨾ S₂)
  ⨾-⊔-distribR : {A B C : Obj} {R : Mor A B} {S₁ S₂ : Mor B C}
               → (R ⨾ (S₁ ⊔ S₂)) ≈ (R ⨾ S₁) ⊔ (R ⨾ S₂)
  ⨾-⊔-distribR = ⊑-antisym ⨾-⊔-subdistribR ⨾-⊔-supdistribR

  ⊔-isRightSubidentity : {A : Obj} {p q : Mor A A}
    → isRightSubidentity p → isRightSubidentity q → isRightSubidentity (p ⊔ q)
  ⊔-isRightSubidentity {A} {p} {q} pR qR {B} {R} = ⊑-begin
       R ⨾ (p ⊔ q)
    ⊑⟨ ⨾-⊔-subdistribR ⟩
       R ⨾ p ⊔ R ⨾ q
    ⊑⟨ ⊔-monotone pR qR ⟩
       R ⊔ R
    ≈⟨ ⊔-idempotent ⟩
       R
    □

retractJoinCompDistrL : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                      → (F : Obj₂ → Obj₁)
                      → {OSG : OrderedSemigroupoid j k₁ k₂ Obj₁}
                      → {joinOp : JoinOp OSG}
                      → JoinCompDistrL joinOp → JoinCompDistrL (retractJoinOp F joinOp)
retractJoinCompDistrL F x = let open JoinCompDistrL x in record {⨾-⊔-subdistribL = ⨾-⊔-subdistribL}

retractJoinCompDistrR : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                      → (F : Obj₂ → Obj₁)
                      → {OSG : OrderedSemigroupoid j k₁ k₂ Obj₁}
                      → {joinOp : JoinOp OSG}
                      → JoinCompDistrR joinOp → JoinCompDistrR (retractJoinOp F joinOp)
retractJoinCompDistrR F x = let open JoinCompDistrR x in record {⨾-⊔-subdistribR = ⨾-⊔-subdistribR}

record RightUSLSemigroupoid {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field orderedSemigroupoid  : OrderedSemigroupoid j k₁ k₂ Obj
  field joinOp               : JoinOp orderedSemigroupoid
  field joinCompDistrR       : JoinCompDistrR joinOp
  open OrderedSemigroupoid orderedSemigroupoid public
```

```
  open LocOrdJoin          Hom             public
  open JoinOp              joinOp          public
  open JoinCompDistrR      joinCompDistrR  public


record LeftUSLSemigroupoid {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field orderedSemigroupoid : OrderedSemigroupoid j k₁ k₂ Obj
  field joinOp              : JoinOp orderedSemigroupoid
  field joinCompDistrL      : JoinCompDistrL joinOp

  open OrderedSemigroupoid orderedSemigroupoid public
  open LocOrdJoin          Hom                 public
  open JoinOp              joinOp              public
  open JoinCompDistrL      joinCompDistrL      public


module JoinCompDistr {i j k₁ k₂ : Level} {Obj : Set i}
  {orderedSemigroupoid : OrderedSemigroupoid j k₁ k₂ Obj}
  {joinOp              : JoinOp orderedSemigroupoid}
  (joinCompDistrL      : JoinCompDistrL joinOp)
  (joinCompDistrR      : JoinCompDistrR joinOp)
  where
  open OrderedSemigroupoid orderedSemigroupoid
  open JoinOp              joinOp
  open JoinCompDistrL      joinCompDistrL
  open JoinCompDistrR      joinCompDistrR


  ⊔-isSubidentity : {A : Obj} {p q : Mor A A} → isSubidentity p → isSubidentity q → isSubidentity (p ⊔ q)
  ⊔-isSubidentity (pL, pR) (qL, qR) = (λ {_} {_} → ⊔-isLeftSubidentity pL qL), (λ {_} {_} → ⊔-isRightSubidentity pR qR)


record USLSemigroupoid {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field orderedSemigroupoid : OrderedSemigroupoid j k₁ k₂ Obj
  field joinOp              : JoinOp orderedSemigroupoid
  field joinCompDistrL      : JoinCompDistrL joinOp
  field joinCompDistrR      : JoinCompDistrR joinOp

  open OrderedSemigroupoid orderedSemigroupoid      public
  open LocOrdJoin          Hom                      public
  open JoinOp              joinOp                   public
  open JoinCompDistrL      joinCompDistrL           public
  open JoinCompDistrR      joinCompDistrR           public
  open JoinCompDistr joinCompDistrL joinCompDistrR public


retractLeftUSLSemigroupoid : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
  → (F : Obj₂ → Obj₁)
  → LeftUSLSemigroupoid j k₁ k₂ Obj₁ → LeftUSLSemigroupoid j k₁ k₂ Obj₂
retractLeftUSLSemigroupoid F base = let open LeftUSLSemigroupoid base in record
  {orderedSemigroupoid = retractOrderedSemigroupoid F orderedSemigroupoid
  ; joinOp = retractJoinOp F joinOp
  ; joinCompDistrL = retractJoinCompDistrL F joinCompDistrL
  }


retractRightUSLSemigroupoid : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
  → (F : Obj₂ → Obj₁)
  → RightUSLSemigroupoid j k₁ k₂ Obj₁ → RightUSLSemigroupoid j k₁ k₂ Obj₂
retractRightUSLSemigroupoid F base = let open RightUSLSemigroupoid base in record
  {orderedSemigroupoid = retractOrderedSemigroupoid F orderedSemigroupoid
  ; joinOp = retractJoinOp F joinOp
```

```
  ; joinCompDistrR = retractJoinCompDistrR F joinCompDistrR
  }

retractUSLSemigroupoid : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                       → (F : Obj₂ → Obj₁)
                       → USLSemigroupoid j k₁ k₂ Obj₁ → USLSemigroupoid j k₁ k₂ Obj₂
retractUSLSemigroupoid F base = let open USLSemigroupoid base in record
  {orderedSemigroupoid = retractOrderedSemigroupoid F orderedSemigroupoid
  ; joinOp = retractJoinOp F joinOp
  ; joinCompDistrL = retractJoinCompDistrL F joinCompDistrL
  ; joinCompDistrR = retractJoinCompDistrR F joinCompDistrR
  }
```

9.6 Categoric.USLCategory

```
record USLCategory {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field orderedCategory : OrderedCategory j k₁ k₂ Obj
  open OrderedCategory orderedCategory
  field joinOp         : JoinOp orderedSemigroupoid
  field joinCompDistrL : JoinCompDistrL joinOp
  field joinCompDistrR : JoinCompDistrR joinOp
  uslSemigroupoid : USLSemigroupoid j k₁ k₂ Obj
  uslSemigroupoid = record
    {orderedSemigroupoid = orderedSemigroupoid
    ; joinOp             = joinOp
    ; joinCompDistrL     = joinCompDistrL
    ; joinCompDistrR     = joinCompDistrR
    }
  open OrderedCategory orderedCategory public
  open JoinOp          joinOp          public
  open JoinCompDistrL  joinCompDistrL  public
  open JoinCompDistrR  joinCompDistrR  public
  open JoinCompDistr joinCompDistrL joinCompDistrR public

retractUSLCategory : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                   → (F : Obj₂ → Obj₁)
                   → USLCategory j k₁ k₂ Obj₁ → USLCategory j k₁ k₂ Obj₂
retractUSLCategory F base = let open USLCategory base in record
  {orderedCategory = retractOrderedCategory F orderedCategory
  ; joinOp = retractJoinOp F joinOp
  ; joinCompDistrL = retractJoinCompDistrL F joinCompDistrL
  ; joinCompDistrR = retractJoinCompDistrR F joinCompDistrR
  }
```

9.7 Categoric.LatticeSemigroupoid

A partial order with binary joins and meets is a lattice, so in addition to the lattice properties involving only join or only meet, we also have the absorbtion laws, collected here into a modules depending on a MeetOp and a JoinOp conforming to a common OrderedSemigroupoid.

```
module HomLattice {i j k₁ k₂ : Level} {Obj : Set i}
                  (OSG : OrderedSemigroupoid j k₁ k₂ Obj)
```

```
                   (meetOp : MeetOp OSG)
                   (joinOp : JoinOp OSG)
  where
  open OrderedSemigroupoid OSG
  open MeetOp meetOp
  open JoinOp  joinOp

  module HomLatticeProps {A B : Obj} where
    open LatticeProps-square (Hom A B) meet join public
  open HomLatticeProps public


record LatticeSemigroupoid {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field orderedSemigroupoid : OrderedSemigroupoid j k₁ k₂ Obj
  field meetOp : MeetOp orderedSemigroupoid
  field joinOp  : JoinOp  orderedSemigroupoid

  rawUSLSemigroupoid : RawUSLSemigroupoid j k₁ k₂ Obj
  rawUSLSemigroupoid = record
    {orderedSemigroupoid = orderedSemigroupoid
    ; joinOp = joinOp
    }

  lslSemigroupoid : LSLSemigroupoid j k₁ k₂ Obj
  lslSemigroupoid = record
    {orderedSemigroupoid = orderedSemigroupoid
    ; meetOp = meetOp
    }

  open OrderedSemigroupoid orderedSemigroupoid        public
  open MeetOp              meetOp                     public
  open JoinOp              joinOp                     public
  open HomLattice orderedSemigroupoid meetOp joinOp public


retractLatticeSemigroupoid : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
  → (F : Obj₂ → Obj₁) → LatticeSemigroupoid j k₁ k₂ Obj₁ → LatticeSemigroupoid j k₁ k₂ Obj₂
retractLatticeSemigroupoid F base = let open LatticeSemigroupoid base in record
  {orderedSemigroupoid = retractOrderedSemigroupoid F orderedSemigroupoid
  ; meetOp = retractMeetOp F meetOp
  ; joinOp = retractJoinOp F joinOp
  }
```

9.8 Categoric.DistrLatSemigroupoid

```
record HomLatticeDistr {i j k₁ k₂ : Level} {Obj : Set i}
  (LSG : LatticeSemigroupoid j k₁ k₂ Obj)
  : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  open LatticeSemigroupoid LSG
  field
    ⊓-⊔-subdistribR : {A B : Obj} {Q R S : Mor A B} → (Q ⊓ (R ⊔ S)) ⊑ (Q ⊓ R) ⊔ (Q ⊓ S)
  module HomDistrLat {A B : Obj} where
    open RDistrLattice (Hom A B) meet join ⊓-⊔-subdistribR public renaming
      (∧-∨-distribR to ⊓-⊔-distribR   -- : {Q R S : Mor A B} → (Q ⊓ (R ⊔ S)) ≈ (Q ⊓ R) ⊔ (Q ⊓ S)
      ; ∧-∨-distribL  to ⊓-⊔-distribL   -- : {Q R S : Mor A B} → ((Q ⊔ R) ⊓ S) ≈ (Q ⊓ S) ⊔ (R ⊓ S)
      ; ∨-∧-distribR to ⊔-⊓-distribR   -- : {Q R S : Mor A B} → (Q ⊔ (R ⊓ S)) ≈ (Q ⊔ R) ⊓ (Q ⊔ S)
      ; ∨-∧-distribL  to ⊔-⊓-distribL   -- : {Q R S : Mor A B} → ((Q ⊓ R) ⊔ S) ≈ (Q ⊔ S) ⊓ (R ⊔ S)
      )
  open HomDistrLat public
```

```
record DistrLatSemigroupoid {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field latticeSemigroupoid : LatticeSemigroupoid j k₁ k₂ Obj
  field homLatDistr         : HomLatticeDistr latticeSemigroupoid

  open LatticeSemigroupoid latticeSemigroupoid
  field joinCompDistrL      : JoinCompDistrL joinOp
  field joinCompDistrR      : JoinCompDistrR joinOp

  uslSemigroupoid : USLSemigroupoid j k₁ k₂ Obj
  uslSemigroupoid = record
    {orderedSemigroupoid = orderedSemigroupoid
    ; joinOp = joinOp
    ; joinCompDistrR = joinCompDistrR
    ; joinCompDistrL = joinCompDistrL
    }

  open LatticeSemigroupoid latticeSemigroupoid public
  open HomLatticeDistr     homLatDistr         public
  open JoinCompDistrL      joinCompDistrL      public
  open JoinCompDistrR      joinCompDistrR      public
```

```
retractHomLatticeDistr : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                       → (F : Obj₂ → Obj₁)
                       → {base : LatticeSemigroupoid j k₁ k₂ Obj₁}
                       → HomLatticeDistr base → HomLatticeDistr (retractLatticeSemigroupoid F base)
retractHomLatticeDistr F homLatticeDistr = let open HomLatticeDistr homLatticeDistr in record
{⊓-⊔-subdistribR = ⊓-⊔-subdistribR}
```

```
retractDistrLatSemigroupoid : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
  → (F : Obj₂ → Obj₁) → DistrLatSemigroupoid j k₁ k₂ Obj₁ → DistrLatSemigroupoid j k₁ k₂ Obj₂
retractDistrLatSemigroupoid F base = let open DistrLatSemigroupoid base in record
  {latticeSemigroupoid = retractLatticeSemigroupoid F latticeSemigroupoid
  ; homLatDistr = retractHomLatticeDistr F homLatDistr
  ; joinCompDistrL = retractJoinCompDistrL F joinCompDistrL
  ; joinCompDistrR = retractJoinCompDistrR F joinCompDistrR
  }
```

9.9 Categoric.ZeroMor

```
module LeastMor {i j k₁ k₂ : Level} {Obj : Set i}
                (OSG : OrderedSemigroupoid j k₁ k₂ Obj) where
  open OrderedSemigroupoid OSG

  isLeastMor : {A B : Obj} → Mor A B → Set (j ⊍ k₂)
  isLeastMor {A} {B} b = (R : Mor A B) → b ⊑ R

  record LeastMor {A B : Obj} : Set (j ⊍ k₂) where
    field
      mor : Mor A B
      proof : isLeastMor mor
  leastMor-≈ : {A B : Obj} {b₁ b₂ : Mor A B} → isLeastMor b₁ → isLeastMor b₂ → b₁ ≈ b₂
  leastMor-≈ {_} {_} {b₁} {b₂} b₁-least b₂-least = ⊑-antisym (b₁-least b₂) (b₂-least b₁)
```

```
record BotMor {i j k₁ k₂ : Level} {Obj : Set i}
              (OSG : OrderedSemigroupoid j k₁ k₂ Obj)
              : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  open OrderedSemigroupoid OSG
  open LeastMor            OSG
```

```
  field
    leastMor : {A B : Obj} → LeastMor {A} {B}
  ⊥ : {A B : Obj} → Mor A B
  ⊥ {A} {B} = LeastMor.mor (leastMor {A} {B})
  is-⊥ : {A B : Obj} → isLeastMor (⊥ {A} {B})
  is-⊥ {A} {B} = LeastMor.proof (leastMor {A} {B})
  ⊥-⊑ : {A B : Obj} {R : Mor A B} → ⊥ ⊑ R
  ⊥-⊑ {A} {B} {R} = is-⊥ {A} {B} R
  leastMor-≈-⊥ : {A B : Obj} {b : Mor A B} → isLeastMor b → b ≈ ⊥
  leastMor-≈-⊥ b-least = leastMor-≈ b-least is-⊥
  ⊑⊥-≈ : {A B : Obj} {R : Mor A B} → R ⊑ ⊥ → R ≈ ⊥
  ⊑⊥-≈ {A} {B} {R} R⊑⊥ = ⊑-antisym R⊑⊥ ⊥-⊑
  ≈⊥-⊑ : {A B : Obj} {b R : Mor A B} → b ≈ ⊥ → b ⊑ R
  ≈⊥-⊑ {A} {B} {R} b≈⊥ = b≈⊥ ⟨≈⊑⟩ ⊥-⊑


record LeftZeroLaw {i j k₁ k₂ : Level} {Obj : Set i}
                   {OSG : OrderedSemigroupoid j k₁ k₂ Obj}
                   (BM : BotMor OSG)
                   : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  open OrderedSemigroupoid OSG
  open LeastMor            OSG
  open BotMor              BM
  field
    leftZero⊑ : {A B C : Obj} {R : Mor B C} → ⊥ {A} {B} ⨾ R ⊑ ⊥
  leftZero : {A B C : Obj} {R : Mor B C} → ⊥ {A} {B} ⨾ R ≈ ⊥
  leftZero = ⊑-antisym leftZero⊑ ⊥-⊑
  ⊥⨾-is-⊥ : {A B C : Obj} {R : Mor B C} → isLeastMor (⊥ {A} {B} ⨾ R)
  ⊥⨾-is-⊥ S = leftZero⊑ ⟨⊑⊑⟩ ⊥-⊑
  is-⊥-⨾ : {A B C : Obj} {R : Mor A B} {S : Mor B C} → isLeastMor R → isLeastMor (R ⨾ S)
  is-⊥-⨾ {R = R} {S} R-least T = ⊑-begin
      R ⨾ S
    ≈⟨ ⨾-cong₁ (leastMor-≈-⊥ R-least) ⟩
      ⊥ ⨾ S
    ⊑⟨ ⊥⨾-is-⊥ T ⟩
      T
    □


record RightZeroLaw {i j k₁ k₂ : Level} {Obj : Set i}
                    {OSG : OrderedSemigroupoid j k₁ k₂ Obj}
                    (BM : BotMor OSG)
                    : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  open OrderedSemigroupoid OSG
  open LeastMor            OSG
  open BotMor              BM
  field
    rightZero⊑ : {A B C : Obj} {R : Mor A B} → R ⨾ ⊥ {B} {C} ⊑ ⊥
  rightZero : {A B C : Obj} {R : Mor A B} → R ⨾ ⊥ {B} {C} ≈ ⊥
  rightZero = ⊑-antisym rightZero⊑ ⊥-⊑
  ⨾⊥-is-⊥ : {A B C : Obj} {R : Mor A B} → isLeastMor (R ⨾ ⊥ {B} {C})
  ⨾⊥-is-⊥ S = rightZero⊑ ⟨⊑⊑⟩ ⊥-⊑
  ⨾-is-⊥ : {A B C : Obj} {R : Mor A B} {S : Mor B C} → isLeastMor S → isLeastMor (R ⨾ S)
  ⨾-is-⊥ {R = R} {S} S-least T = ⊑-begin
      R ⨾ S
    ≈⟨ ⨾-cong₂ (leastMor-≈-⊥ S-least) ⟩
      R ⨾ ⊥
    ⊑⟨ ⨾⊥-is-⊥ T ⟩
```

```
  ⊤
□

record ZeroMor {i j k₁ k₂ : Level} {Obj : Set i}
  (OSG : OrderedSemigroupoid j k₁ k₂ Obj)
  : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  field botMor        : BotMor OSG
        leftZeroLaw   : LeftZeroLaw botMor
        rightZeroLaw  : RightZeroLaw botMor
  open BotMor        botMor        public
  open LeftZeroLaw   leftZeroLaw   public
  open RightZeroLaw  rightZeroLaw  public

retractBotMor : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
              → (F : Obj₂ → Obj₁)
              → {base : OrderedSemigroupoid j k₁ k₂ Obj₁}
              → BotMor base → BotMor (retractOrderedSemigroupoid F base)
retractBotMor F botMor = let open BotMor botMor in
  record {leastMor = λ {A} {B} → record {mor = ⊥; proof = λ R → ⊥-⊑}}

retractLeftZeroLaw : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                   → (F : Obj₂ → Obj₁)
                   → {base : OrderedSemigroupoid j k₁ k₂ Obj₁}
                   → {botMor : BotMor base}
                   → LeftZeroLaw botMor → LeftZeroLaw (retractBotMor F botMor)
retractLeftZeroLaw F z = let open LeftZeroLaw z in record {leftZero⊑ = leftZero⊑}

retractRightZeroLaw : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
  → (F : Obj₂ → Obj₁)
  → {base : OrderedSemigroupoid j k₁ k₂ Obj₁}
  → {botMor : BotMor base}
  → RightZeroLaw botMor → RightZeroLaw (retractBotMor F botMor)
retractRightZeroLaw F z = let open RightZeroLaw z in record {rightZero⊑ = rightZero⊑}

retractZeroMor : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
  → (F : Obj₂ → Obj₁)
  → {base : OrderedSemigroupoid j k₁ k₂ Obj₁}
  → ZeroMor base → ZeroMor (retractOrderedSemigroupoid F base)
retractZeroMor F zeroMor = let open ZeroMor zeroMor in record
  {botMor = retractBotMor F botMor
  ; leftZeroLaw = retractLeftZeroLaw F leftZeroLaw
  ; rightZeroLaw = retractRightZeroLaw F rightZeroLaw
  }
```

Chapter 10

Domain

Domain can be *defined* as a derived operation in allegories (see Sect. 12.3). For weaker theories, Desharnais et al. (2006) axiomatised domain operators in semirings and Kleene algebras, having the domain operator produce idempotent subidentities. These definition have been adapted to the ordered category setting (Kahl, 2004) and later to ordered semigroupoids (Kahl, 2008); we formalise domain in orered semigroupoids in Sect. 10.2, and move this to ordered categories in Sect. 10.3.

A more recent alternative is the purely equational approach of Desharnais et al. (2009), which starts just from semigroups. Sect. 10.1 formalises this latter approach, again concentrating on the aspects that do not require complements.

10.1 Categoric.DomainSemigroupoid

```
record LeftClosOp {i j k : Level} {Obj : Set i}
                  (base : Semigroupoid j k Obj)
                  : Set (i ⊍ j ⊍ k) where
  open Semigroupoid base
  field
    dom : {A B : Obj} → Mor A B → Mor A A
    dom-cong : {A B : Obj} {R S : Mor A B} → R ≈ S → dom R ≈ dom S
    D1 : {A B : Obj} {R : Mor A B} → dom R ⨾ R ≈ R
    L2 : {A B : Obj} {R : Mor A B} → dom (dom R) ≈ dom R
    L3 : {A B C : Obj} {R : Mor A B} {S : Mor B C} → dom R ⨾ dom (R ⨾ S) ≈ dom (R ⨾ S)
    D4 : {A B C : Obj} {R : Mor A B} {S : Mor A C} → dom R ⨾ dom S ≈ dom S ⨾ dom R
  dom-fixpoint : {A B : Obj} → {R : Mor A A} → {S : Mor A B}
               → R ≈ dom S → dom R ≈ R
  dom-fixpoint {A} {B} {R} {S} R≈domS = ≈-begin
      dom R
    ≈⟨ dom-cong R≈domS ⟩
      dom (dom S)
    ≈⟨ L2 ⟩
      dom S
    ≈⟨ ≈-sym R≈domS ⟩
      R
    □
  dom-⨾-idempotent : {A B : Obj} {R : Mor A B} → dom R ⨾ dom R ≈ dom R
  dom-⨾-idempotent {A} {B} {R} = ≈-begin
      dom R ⨾ dom R
    ≈⟨ ≈-sym (⨾-cong L2 (dom-cong D1)) ⟩
      dom (dom R) ⨾ dom (dom R ⨾ R)
    ≈⟨ L3 ⟩
```

```
    dom (dom R ⨾ R)
  ≈⟨ dom-cong D1 ⟩
    dom R
  □

dom-dom⨾dom : {A B C : Obj} {R : Mor A B} {S : Mor A C}
             → dom (dom R ⨾ dom S) ≈ dom R ⨾ dom S
dom-dom⨾dom {A} {B} {C} {R} {S} = ≈-begin
    dom (dom R ⨾ dom S)
  ≈⟨ ≈-sym L3 ⟩
    dom (dom R) ⨾ dom (dom R ⨾ dom S)
  ≈⟨ ⨾-cong L2 (dom-cong D4) ⟩
    dom R ⨾ dom (dom S ⨾ dom R)
  ≈⟨ ⨾-cong₂ (≈-sym L3) ⟩
    dom R ⨾ dom (dom S) ⨾ dom (dom S ⨾ dom R)
  ≈⟨ ⨾-cong₂₁ L2 ⟩
    dom R ⨾ dom S ⨾ dom (dom S ⨾ dom R)
  ≈⟨ ⨾-cong₂ D4 ⟩
    dom R ⨾ dom (dom S ⨾ dom R) ⨾ dom S
  ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ D4 ⟨≈≈⟩ ⨾-assoc ⟩
    dom (dom S ⨾ dom R) ⨾ dom R ⨾ dom S
  ≈⟨ ⨾-cong₁ (dom-cong D4) ⟩
    dom (dom R ⨾ dom S) ⨾ dom R ⨾ dom S
  ≈⟨ D1 ⟩
    dom R ⨾ dom S
  □

L5 : {A B C : Obj} {R : Mor A B} {S : Mor A C} → dom S ⨾ dom (dom R ⨾ S) ≈ dom (dom R ⨾ S)
L5 {A} {B} {C} {R} {S} = ≈-begin
    dom S ⨾ dom (dom R ⨾ S)
  ≈⟨ ⨾-cong₂ (dom-cong (⨾-cong₂ (≈-sym D1))) ⟩
    dom S ⨾ dom (dom R ⨾ dom S ⨾ S)
  ≈⟨ ⨾-cong (≈-sym L2) (dom-cong (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ D4 ⟨≈≈⟩ ⨾-assoc)) ⟩
    dom (dom S) ⨾ dom (dom S ⨾ dom R ⨾ S)
  ≈⟨ L3 ⟩
    dom (dom S ⨾ dom R ⨾ S)
  ≈⟨ dom-cong (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ D4 ⟨≈≈⟩ ⨾-assoc) ⟩
    dom (dom R ⨾ dom S ⨾ S)
  ≈⟨ dom-cong (⨾-cong₂ D1) ⟩
    dom (dom R ⨾ S)
  □

L5' : {A B C : Obj} {R : Mor A B} {S : Mor A C} → dom (dom R ⨾ S) ⨾ dom S ≈ dom (dom R ⨾ S)
L5' = D4 ⟨≈≈⟩ L5

  -- The “fundamental order”:
_≼_ : {A B : Obj} (R S : Mor A B) → Set k
R ≼ S = R ≈ dom R ⨾ S

≼-refl : {A B : Obj} {R : Mor A B} → R ≼ R
≼-refl = ≈-sym D1

≼-reflexive : {A B : Obj} {R S : Mor A B} → R ≈ S → R ≼ S
≼-reflexive R≈S = ≈-sym D1 ⟨≈≈⟩ ⨾-cong₂ R≈S

≼-trans : {A B : Obj} {R S T : Mor A B} → R ≼ S → S ≼ T → R ≼ T
≼-trans {A} {B} {R} {S} {T} R≼S S≼T = ≈-begin
    R
  ≈⟨ R≼S ⟩
    dom R ⨾ S
  ≈⟨ ⨾-cong (dom-cong R≼S) S≼T ⟩
    dom (dom R ⨾ S) ⨾ dom S ⨾ T
  ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ L5' ⟩
    dom (dom R ⨾ S) ⨾ T
```

```
    ≈⟨ ⨾-cong₁ (dom-cong (≈-sym R≼S)) ⟩
      dom R ⨾ T
    □

  ≼-antisym : {A B : Obj} {R S T : Mor A B} → R ≼ S → S ≼ R → R ≈ S
  ≼-antisym {A} {B} {R} {S} {T} R≼S S≼R = ≈-begin
      R
    ≈⟨ R≼S ⟩
      dom R ⨾ S
    ≈⟨ ⨾-cong₂ (≈-sym D1) ⟩
      dom R ⨾ dom S ⨾ S
    ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ D4 ⟨≈≈⟩ ⨾-assoc ⟩
      dom S ⨾ dom R ⨾ S
    ≈⟨ ⨾-cong₂ (≈-sym R≼S) ⟩
      dom S ⨾ R
    ≈⟨ ≈-sym S≼R ⟩
      S
    □

  -- composition is meet:

  ≼-to⨾meet : {A B C : Obj} {R : Mor A B} {S : Mor A C}
            → dom R ≼ dom S → dom R ⨾ dom S ≈ dom R
  ≼-to⨾meet {A} {B} {C} {R} {S} domR≼domS = ≈-sym (≈-begin
      dom R
    ≈⟨ domR≼domS ⟩
      dom (dom R) ⨾ dom S
    ≈⟨ ⨾-cong₁ L2 ⟩
      dom R ⨾ dom S
    □)

  ≼-from⨾meet   : {A B C : Obj} {R : Mor A B} {S : Mor A C}
                → dom R ⨾ dom S ≈ dom R → dom R ≼ dom S
  ≼-from⨾meet {A} {B} {C} {R} {S} domRdomS≈domR = ≈-sym (≈-begin
      dom (dom R) ⨾ dom S
    ≈⟨ ⨾-cong₁ L2 ⟩
      dom R ⨾ dom S
    ≈⟨ domRdomS≈domR ⟩
      dom R
    □)

  -- preserved by multiplication from the right:

  ⨾-≼monotone₂ :  {A B C : Obj} {R S : Mor A B} {T : Mor B C} → R ≼ S → (R ⨾ T) ≼ (S ⨾ T)
  ⨾-≼monotone₂ {A} {B} {C} {R} {S} {T} R≼S = ≈-begin
      R ⨾ T
    ≈⟨ ≈-sym D1 ⟩
      dom (R ⨾ T) ⨾ R ⨾ T
    ≈⟨ ⨾-cong₂ (⨾-cong₁ R≼S ⟨≈≈⟩ ⨾-assoc) ⟩
      dom (R ⨾ T) ⨾ dom R ⨾ S ⨾ T
    ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (D4 ⟨≈≈⟩ L3) ⟩
      dom (R ⨾ T) ⨾ S ⨾ T
    □

  -- Additional properties:

  dom⨾-≼decreasing : {A B C : Obj} {Q : Mor A B} {R : Mor A C} → (dom Q ⨾ R) ≼ R
  dom⨾-≼decreasing {A} {B} {C} {Q} {R} = ≈-begin
      dom Q ⨾ R
    ≈⟨ ≈-sym D1 ⟩
      dom (dom Q ⨾ R) ⨾ dom Q ⨾ R
    ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁₂ (≈-sym L2) ⟩
      (dom (dom Q ⨾ R) ⨾ dom (dom Q)) ⨾ R
    ≈⟨ ⨾-cong₁ (D4 ⟨≈≈⟩ L3) ⟩
      dom (dom Q ⨾ R) ⨾ R
```

```
  □
dom⨾-≼monotone : {A B C : Obj} {Q : Mor A B} {R S : Mor A C}
                → R ≼ S → (dom Q ⨾ R) ≼ (dom Q ⨾ S)
dom⨾-≼monotone {A} {B} {C} {Q} {R} {S} R≼S = ≈-begin
    dom Q ⨾ R
  ≈⟨ ≈-sym D1 ⟩
    dom (dom Q ⨾ R) ⨾ dom Q ⨾ R
  ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ D4 ⟨≈≈⟩ ⨾-assoc ⟩
    dom Q ⨾ dom (dom Q ⨾ R) ⨾ R
  ≈⟨ ⨾-cong₂₂ R≼S ⟩
    dom Q ⨾ dom (dom Q ⨾ R) ⨾ dom R ⨾ S
  ≈⟨ ⨾-cong₂ (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ L5') ⟩
    dom Q ⨾ dom (dom Q ⨾ R) ⨾ S
  ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ D4 ⟨≈≈⟩ ⨾-assoc ⟩
    dom (dom Q ⨾ R) ⨾ dom Q ⨾ S
  □
-- ≼-monotonicity of dom follows from D3:
dom-D3-≼monotone : {A B : Obj} {R S : Mor A B}
                  → dom (dom R ⨾ S) ≈ dom R ⨾ dom S → R ≼ S → dom R ≼ dom S
dom-D3-≼monotone {A} {B} {R} {S} D3 R≼S = ≈-begin
    dom R
  ≈⟨ dom-cong R≼S ⟩
    dom (dom R ⨾ S)
  ≈⟨ D3 ⟩
    dom R ⨾ dom S
  ≈⟨ ⨾-cong₁ (≈-sym L2) ⟩
    dom (dom R) ⨾ dom S
  □
```

D3 also holds for arguments in ≼ for which ≼-monotonicity of dom holds:

```
dom-≼monotone-D3 : {A B : Obj} {R S : Mor A B}
                  → R ≼ S → dom R ≼ dom S → dom (dom R ⨾ S) ≈ dom R ⨾ dom S
dom-≼monotone-D3 {A} {B} {R} {S} R≼S domR≼domS = ≈-begin
    dom (dom R ⨾ S)
  ≈⟨ dom-cong (≈-sym R≼S) ⟩
    dom R
  ≈⟨ domR≼domS ⟩
    dom (dom R) ⨾ dom S
  ≈⟨ ⨾-cong₁ L2 ⟩
    dom R ⨾ dom S
  □
```

However, ≼-monotonicity of dom does not imply D3; mace4 finds a four-element counter-example.

```
record PredomainOp {i j k : Level} {Obj : Set i}
  (base : Semigroupoid j k Obj)
  : Set (i ⊍ j ⊍ k) where
  open Semigroupoid base
  field
    dom : {A B : Obj} → Mor A B → Mor A A
    dom-cong : {A B : Obj} {R S : Mor A B} → R ≈ S → dom R ≈ dom S
    D1 : {A B : Obj} {R : Mor A B} → dom R ⨾ R ≈ R
    D3 : {A B C : Obj} {R : Mor A B} {S : Mor A C} → dom (dom R ⨾ S) ≈ dom R ⨾ dom S
    D4 : {A B C : Obj} {R : Mor A B} {S : Mor A C} → dom R ⨾ dom S ≈ dom S ⨾ dom R
  dom-⨾-idempotent : {A B : Obj} {R : Mor A B} → dom R ⨾ dom R ≈ dom R
  dom-⨾-idempotent {A} {B} {R} = ≈-begin
      dom R ⨾ dom R
```

```
    ≈⟨ ≈-sym D3 ⟩
      dom (dom R ⨾ R)
    ≈⟨ dom-cong D1 ⟩
      dom R
    □

  L2 : {A B : Obj} {R : Mor A B} → dom (dom R) ≈ dom R
  L2 {A} {B} {R} = ≈-begin
      dom (dom R)
    ≈⟨ dom-cong (≈-sym dom-⨾-idempotent) ⟩
      dom (dom R ⨾ dom R)
    ≈⟨ D3 ⟩
      dom R ⨾ dom (dom R)
    ≈⟨ D4 ⟩
      dom (dom R) ⨾ dom R
    ≈⟨ D1 ⟩
      dom R
    □

  L3 : {A B C : Obj} {R : Mor A B} {S : Mor B C} → dom R ⨾ dom (R ⨾ S) ≈ dom (R ⨾ S)
  L3 {A} {B} {C} {R} {S} = ≈-begin
      dom R ⨾ dom (R ⨾ S)
    ≈⟨ ≈-sym D3 ⟩
      dom (dom R ⨾ R ⨾ S)
    ≈⟨ dom-cong (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ D1) ⟩
      dom (R ⨾ S)
    □

  leftClosOp : LeftClosOp base
  leftClosOp = record
    {dom = dom
    ; dom-cong = dom-cong
    ; D1 = D1
    ; L2 = L2
    ; L3 = L3
    ; D4 = D4
    }

  open LeftClosOp leftClosOp public hiding (dom; dom-cong; D1; L2; L3; D4; dom-⨾-idempotent)

  dom-≼monotone : {A B : Obj} {R S : Mor A B} → R ≼ S → dom R ≼ dom S
  dom-≼monotone = dom-D3-≼monotone D3

record DomainOp {i j k : Level} {Obj : Set i}
  (base : Semigroupoid j k Obj)
  : Set (i ⊍ j ⊍ k) where
  open Semigroupoid base
  field
    dom : {A B : Obj} → Mor A B → Mor A A
    dom-cong : {A B : Obj} {R S : Mor A B} → R ≈ S → dom R ≈ dom S
    D1 : {A B : Obj} {R : Mor A B} → dom R ⨾ R ≈ R
    D2 : {A B C : Obj} {R : Mor A B} {S : Mor B C} → dom (R ⨾ dom S) ≈ dom (R ⨾ S)
    D3 : {A B C : Obj} {R : Mor A B} {S : Mor A C} → dom (dom R ⨾ S) ≈ dom R ⨾ dom S
    D4 : {A B C : Obj} {R : Mor A B} {S : Mor A C} → dom R ⨾ dom S ≈ dom S ⨾ dom R

  predomainOp : PredomainOp base
  predomainOp = record
    {dom = dom
    ; dom-cong = dom-cong
    ; D1 = D1
    ; D3 = D3
    ; D4 = D4
    }
```

```
  open PredomainOp predomainOp public hiding (dom; dom-cong; D1; D3; D4)
```

10.2 Categoric.OSGD

```
record OSGDomainOp {i j k₁ k₂ : Level} {Obj : Set i}
                   (base : OrderedSemigroupoid j k₁ k₂ Obj)
                   : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  open OrderedSemigroupoid base
  field
    dom : {A B : Obj} → Mor A B → Mor A A
    domSubIdentity       : {A B : Obj} {R : Mor A B} → isSubidentity (dom R)
    dom-⨾-idempotent     : {A B : Obj} {R : Mor A B} → dom R ⨾ dom R ≈ dom R
    domPreserves⊑       : {A B : Obj} {Q R : Mor A B} → Q ⊑ R → Q ⊑ dom R ⨾ Q
    domLeastPreserver    : {A B : Obj} {R : Mor A B} {d : Mor A A}
                         → isSubidentity d → (d ⨾ d ≈ d) → (R ⊑ d ⨾ R) → dom R ⊑ d
    domLocality   : ∀ {A B C : Obj} {R : Mor A B} {S : Mor B C}
                  → dom (R ⨾ dom S) ⊑ dom (R ⨾ S)
  domPreserves : {A B : Obj} {R : Mor A B} → R ⊑ dom R ⨾ R
  domPreserves = domPreserves⊑ ⊑-refl
  dom-D1 : {A B : Obj} {R : Mor A B} → dom R ⨾ R ≈ R
  dom-D1 = ⊑-antisym (proj₁ domSubIdentity) domPreserves
  dom-monotone   : {A B : Obj} {R S : Mor A B} → R ⊑ S → dom R ⊑ dom S
  dom-monotone leq = domLeastPreserver domSubIdentity dom-⨾-idempotent (domPreserves⊑ leq)
  dom-cong : {A B : Obj} {R S : Mor A B} → R ≈ S → dom R ≈ dom S
  dom-cong R≈S = ⊑-antisym (dom-monotone (⊑-reflexive R≈S))
                           (dom-monotone (⊑-reflexive′ R≈S))
  rightSubidentity-dom : {A : Obj} {v : Mor A A} → isRightSubidentity v → v ⊑ dom v
  rightSubidentity-dom {A} {v} v-isRightSubid = domPreserves ⟨⊑⊑⟩ v-isRightSubid
  idempotSubidentity-dom : {A : Obj} {v : Mor A A}
                         → isSubidentity v
                         → (v ⨾ v ≈ v)
                         → dom v ≈ v
  idempotSubidentity-dom {A} {v} v-isSubid v⨾v≈v = ⊑-antisym
    (domLeastPreserver v-isSubid v⨾v≈v (⊑-reflexive′ v⨾v≈v))
    (rightSubidentity-dom (proj₂ v-isSubid))
  dom-idempotent : {A B : Obj} {R : Mor A B} → dom (dom R) ≈ dom R
  dom-idempotent {A} {B} {R} = idempotSubidentity-dom domSubIdentity dom-⨾-idempotent
  domLocality’  : {A B C : Obj} {R : Mor A B} {S : Mor B C} → dom (R ⨾ S) ⊑ dom R
  domLocality’ {A} {B} {C} {R} {S} = domLeastPreserver domSubIdentity dom-⨾-idempotent
    (⊑-begin
        R ⨾ S
      ⊑⟨ ⨾-monotone₁ domPreserves ⟩
        (dom R ⨾ R) ⨾ S
      ≈⟨ ⨾-assoc ⟩
        dom R ⨾ R ⨾ S
      □)
  domLocality≈ : ∀ {A B C : Obj} {R : Mor A B} {S : Mor B C}
               → dom (R ⨾ dom S) ≈ dom (R ⨾ S)
  domLocality≈ {A} {B} {C} {R} {S} = ⊑-antisym domLocality
    -- (dom-cong (⨾-cong₂ (≈-sym dom-D1) ⟨≈≈⟩ ⨾-assocL) ⟨≈⊑⟩ domLocality’)
    (⊑-begin
        dom (R ⨾ S)
      ≈⟨ dom-cong (⨾-cong₂ (≈-sym dom-D1) ⟨≈≈⟩ ⨾-assocL) ⟩
        dom ((R ⨾ dom S) ⨾ S)
```

```
      ⊑⟨ domLocality' ⟩
        dom (R ⨾ dom S)
      □)
  -- dom-L3 is currently unused.
dom-L3 : {A B C : Obj} {R : Mor A B} {S : Mor B C} → dom R ⨾ dom (R ⨾ S) ≈ dom (R ⨾ S)
dom-L3 {A} {B} {C} {R} {S} = ⊑-antisym
  (proj₁ domSubIdentity)    -- Left!
  (⊑-begin
        dom (R ⨾ S)
      ≈⟨ ≈-sym dom-⨾-idempotent ⟩
        dom (R ⨾ S) ⨾ dom (R ⨾ S)
      ⊑⟨ ⨾-monotone₁ domLocality' ⟩
        dom R ⨾ dom (R ⨾ S)
      □)
dom-stutter  : {A B C : Obj} {R : Mor A B} {S : Mor A C}
             → dom R ⨾ S ⊑ dom R ⨾ dom S ⨾ dom R ⨾ S
dom-stutter  {A} {B} {C} {R} {S} = ⊑-begin
        dom R ⨾ S
      ≈⟨ ⨾-cong₁ (≈-sym dom-⨾-idempotent) ⟩
        (dom R ⨾ dom R) ⨾ S
      ≈⟨ ⨾-assoc ⟩
        dom R ⨾ dom R ⨾ S
      ⊑⟨ ⨾-monotone₂ (domPreserves⊑ (proj₁ domSubIdentity)) ⟩    -- Left!
        dom R ⨾ dom S ⨾ dom R ⨾ S
      □
dom-unstutter   : {A B C : Obj} {R : Mor A B} {S : Mor A C}
                → dom R ⨾ dom S ⨾ dom R ⨾ S ⊑ dom R ⨾ S
dom-unstutter {A} {B} {C} {R} {S} = ⊑-begin
        dom R ⨾ dom S ⨾ dom R ⨾ S
      ≈⟨ ⨾-assocL ⟩
        (dom R ⨾ dom S) ⨾ dom R ⨾ S
      ⊑⟨ ⨾-monotone₁ (proj₂ domSubIdentity) ⟩
        dom R ⨾ dom R ⨾ S
      ≈⟨ ⨾-assocL ⟩
        (dom R ⨾ dom R) ⨾ S
      ≈⟨ ⨾-cong₁ dom-⨾-idempotent ⟩
        dom R ⨾ S
      □
dom-stutter≈  : {A B C : Obj} {R : Mor A B} {S : Mor A C}
              → dom R ⨾ S ≈ dom R ⨾ dom S ⨾ dom R ⨾ S
dom-stutter≈  = ⊑-antisym dom-stutter dom-unstutter
-- stutter uses Left:
dom⨾dom-idempotent : ∀ {A B C : Obj} {R : Mor A B} {S : Mor A C}
                   → dom R ⨾ dom S ⨾ dom R ⨾ dom S ≈ dom R ⨾ dom S
dom⨾dom-idempotent {A} {B} {C} {R} {S} = ≈-sym (≈-begin
        dom R ⨾ dom S
      ≈⟨ dom-stutter≈ ⟩
        dom R ⨾ dom (dom S) ⨾ dom R ⨾ dom S
      ≈⟨ ⨾-cong₂₁ dom-idempotent ⟩
        dom R ⨾ dom S ⨾ dom R ⨾ dom S
      □)
-- dom-⨾-dom-subswap is not used, because we have a direct proof for dom-⨾-dom-swap.
dom-⨾-dom-subswap : {A B C : Obj} {R : Mor A B} {S : Mor A C}
                  → dom (dom R ⨾ S) ⊑ dom R ⨾ dom S
dom-⨾-dom-subswap = domLeastPreserver (⨾-isSubidentity domSubIdentity domSubIdentity)
                                      (⨾-assoc ⟨≈≈⟩ dom⨾dom-idempotent)
                                      (dom-stutter ⟨⊑≈⟩ ⨾-assocL)
```

```
-- D3:
dom-⨾-dom-swap : {A B C : Obj} {R : Mor A B} {S : Mor A C}
                 → dom (dom R ⨾ S) ≈ dom R ⨾ dom S
dom-⨾-dom-swap {A} {B} {C} {R} {S} = ≈-begin
      dom (dom R ⨾ S)
  ≈⟨ ≈-sym domLocality≈ ⟩
      dom (dom R ⨾ dom S)
  ≈⟨ idempotSubidentity-dom
    (⨾-isSubidentity domSubIdentity domSubIdentity)
    (⨾-assoc ⟨≈≈⟩ dom⨾dom-idempotent)
    ⟩
      dom R ⨾ dom S
  □

dom-⨾-subcommute : {A B C : Obj} {R : Mor A B} {S : Mor A C}
                   → dom R ⨾ dom S ⊑ dom S ⨾ dom R
dom-⨾-subcommute {A} {B} {C} {R} {S} = ⊑-begin
      dom R ⨾ dom S
    ≈⟨ ≈-sym dom-⨾-dom-swap ⟩
      dom (dom R ⨾ S)
    ≈⟨ ≈-sym dom-⨾-idempotent ⟩
      dom (dom R ⨾ S) ⨾ dom (dom R ⨾ S)
    ⊑⟨ ⨾-monotone (dom-monotone (proj₁ domSubIdentity)) domLocality' ⟩
      dom S ⨾ dom (dom R)
    ≈⟨ ⨾-cong₂ dom-idempotent ⟩
      dom S ⨾ dom R
    □

-- D4:
dom-⨾-commute : {A B C : Obj} {R : Mor A B} {S : Mor A C}
                → dom R ⨾ dom S ≈ dom S ⨾ dom R
dom-⨾-commute = ⊑-antisym dom-⨾-subcommute dom-⨾-subcommute

dom⨾dom-meet-from⊑₁ : {A B C : Obj} {R : Mor A B} {S : Mor A C}
                      → dom R ⊑ dom S → dom R ⨾ dom S ≈ dom R
dom⨾dom-meet-from⊑₁ {A} {B} {C} {R} {S} domR⊑domS = ⊑-antisym
  (⊑-begin
      dom R ⨾ dom S
    ≈⟨ ≈-sym dom-⨾-dom-swap ⟩
      dom (dom R ⨾ S)
    ⊑⟨ domLocality' ⟩
      dom (dom R)
    ≈⟨ dom-idempotent ⟩
      dom R
    □)
  (≈-sym dom-⨾-idempotent ⟨≈⊑⟩ ⨾-monotone₂ domR⊑domS)

dom⨾dom-meet-from⊑₂ : {A B C : Obj} {R : Mor A B} {S : Mor A C}
                      → dom R ⊑ dom S → dom S ⨾ dom R ≈ dom R
dom⨾dom-meet-from⊑₂ domR⊑domS =
      dom-⨾-commute ⟨≈≈⟩ dom⨾dom-meet-from⊑₁ domR⊑domS

isTotalD : {A B : Obj} → Mor A B → Set (i ⊍ j ⊍ k₂)
isTotalD R = isSuperidentity (dom R)

isTotalD-isId : {A B : Obj} {R : Mor A B} → isTotalD R → isIdentity (dom R)
isTotalD-isId = ⊑⊒-isIdentity domSubIdentity

⨾-isTotalD  : {A B C : Obj} {R : Mor A B} {S : Mor B C}
            → isTotalD R → isTotalD S → isTotalD (R ⨾ S)
⨾-isTotalD  {A} {B} {C} {R} {S} isTotalR isTotalS = ⊑-isSuperidentity
  (⊑-begin
      dom R
    ⊑⟨ dom-monotone (proj₂ isTotalS) ⟩
```

```
      dom (R ⨾ dom S)
    ⊑⟨ domLocality ⟩
      dom (R ⨾ S)
    □) isTotalR
isDomainMinimal : {A B : Obj} → Mor A B → Set (j ⊍ k₂)
isDomainMinimal {A} {B} R = ∀ {Q : Mor A B} → Q ⊑ R → dom Q ⨾ R ⊑ Q
isDomainMinimal≈  : {A B : Obj} {R : Mor A B}
                  → isDomainMinimal R → {Q : Mor A B} → Q ⊑ R → dom Q ⨾ R ≈ Q
isDomainMinimal≈ isDomainMinimalR Q⊑R =
      ⊑-antisym (isDomainMinimalR Q⊑R) (domPreserves ⟨⊑⊑⟩ ⨾-monotone₂ Q⊑R)
```

According to Desharnais and Möller (2001), composition of domain-minimal morphisms is not necessarily domain-minimal again; to overcome this, they provide an atom-based restriction. Kahl (2008) generalises this to join-indecomposable elements.

```
isMappingD : {A B : Obj} → Mor A B → Set (i ⊍ j ⊍ k₂)
isMappingD R = isTotalD R × isDomainMinimal R
```

We now introduce range, the opposite concept to domain, via an explicit definition.

```
record OSGRangeOp {i j k₁ k₂ : Level} {Obj : Set i}
             (base : OrderedSemigroupoid j k₁ k₂ Obj)
             : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  open OrderedSemigroupoid base
  field
    ran : {A B : Obj} → Mor A B → Mor B B
    ranSubIdentity : {A B : Obj} {R : Mor A B} → isSubidentity (ran R)
    ran-⨾-idempotent : {A B : Obj} {R : Mor A B} → ran R ⨾ ran R ≈ ran R
    ranPreserves⊑ : {A B : Obj} {Q R : Mor A B} → Q ⊑ R → Q ⊑ Q ⨾ ran R
    ranLeastPreserver : {A B : Obj} {R : Mor A B} {d : Mor B B}
                      → isSubidentity d → (d ⨾ d ≈ d) → (R ⊑ R ⨾ d) → ran R ⊑ d
    ranLocality  : ∀ {A B C : Obj} {R : Mor A B} {S : Mor B C}
                 → ran (ran R ⨾ S) ⊑ ran (R ⨾ S)
```

Range defines a domain operator in the opposite ordered semigroupoid:

```
oppositeOSGDomainOp : OSGDomainOp (oppositeOrderedSemigroupoid base)
oppositeOSGDomainOp = record
  {dom                = λ {B} {A}          → ran               {A} {B}
  ; domSubIdentity    = λ {B} {A} {R}      → swapFromSubid (ranSubIdentity {A} {B} {R})
  ; dom-⨾-idempotent  = λ {B} {A} {R}      → ran-⨾-idempotent  {A} {B} {R}
  ; domPreserves⊑     = λ {B} {A} {Q} {R}  → ranPreserves⊑     {A} {B} {Q} {R}
  ; domLeastPreserver = λ {B} {A} {R} {d} subid
                      → ranLeastPreserver  {A} {B} {R} {d} (swapToSubid subid)
  ; domLocality       = λ {C} {B} {A} {S} {R} → ranLocality     {A} {B} {C} {R} {S}
  }
```

Instead of exporting the domain properties from oppositeOSGDomainOp and renaming them for range, we re-define these properties, since that has two advantages:

- better documentation of the OSGRangeOp interface
- more flexibility to adapt the interface to make it more natural, especially with respect to argument sequence, and more useful, in particular for cases involving isSubidentity.

```
open OSGDomainOp oppositeOSGDomainOp  -- not public
ranPreserves : {A B : Obj} {R : Mor A B} → R ⊑ R ⨾ ran R
ranPreserves {A} {B} {R} = domPreserves {B} {A} {R}
```

```
ran-D1 : {A B : Obj} {R : Mor A B} → R ⨾ ran R ≈ R
ran-D1 {A} {B} {R} = dom-D1 {B} {A} {R}
ran-monotone  : {A B : Obj} {R S : Mor A B} → R ⊑ S → ran R ⊑ ran S
ran-monotone {A} {B} {R} {S} = dom-monotone {B} {A} {R} {S}
ran-cong : {A B : Obj} {R S : Mor A B} → R ≈ S → ran R ≈ ran S
ran-cong {A} {B} {R} {S} = dom-cong {B} {A} {R} {S}
leftSubidentity-ran  : {A : Obj} {v : Mor A A} → isLeftSubidentity v → v ⊑ ran v
leftSubidentity-ran = rightSubidentity-dom
idempotSubidentity-ran : {A : Obj} {v : Mor A A}
                       → isSubidentity v
                       → (v ⨾ v ≈ v)
                       → ran v ≈ v
idempotSubidentity-ran isSubid = idempotSubidentity-dom (swapFromSubid isSubid)
ran-idempotent : {A B : Obj} {R : Mor A B} → ran (ran R) ≈ ran R
ran-idempotent {A} {B} {R} = dom-idempotent {B} {A} {R}
ranLocality' : {A B C : Obj} {R : Mor A B} {S : Mor B C} → ran (R ⨾ S) ⊑ ran S
ranLocality' {A} {B} {C} {R} {S} = domLocality' {C} {B} {A} {S} {R}
ranLocality≈ : ∀ {A B C : Obj} {R : Mor A B} {S : Mor B C}
             → ran (ran R ⨾ S) ≈ ran (R ⨾ S)
ranLocality≈ {A} {B} {C} {R} {S} = domLocality≈ {C} {B} {A} {S} {R}
ran-L3 : {A B C : Obj} {R : Mor A B} {S : Mor B C} → ran (R ⨾ S) ⨾ ran S ≈ ran (R ⨾ S)
ran-L3 {A} {B} {C} {R} {S} = dom-L3 {C} {B} {A} {S} {R}
ran-stutter : {A B C : Obj} {R : Mor A C} {S : Mor B C}
            → R ⨾ ran S ⊑ R ⨾ ran S ⨾ ran R ⨾ ran S
ran-stutter {A} {B} {C} {R} {S} = dom-stutter {C} {B} {A} {S} {R} ⟨⊑≈⟩ ⨾-assoc₄
ran-unstutter  : {A B C : Obj} {R : Mor A C} {S : Mor B C}
               → R ⨾ ran S ⨾ ran R ⨾ ran S ⊑ R ⨾ ran S
ran-unstutter  {A} {B} {C} {R} {S}
               = ⨾-assocL₄ ⟨≈⊑⟩ dom-unstutter {C} {B} {A} {S} {R}
ran-stutter≈ : {A B C : Obj} {R : Mor A C} {S : Mor B C}
             → R ⨾ ran S ≈ R ⨾ ran S ⨾ ran R ⨾ ran S
ran-stutter≈ {A} {B} {C} {R} {S} = dom-stutter≈ {C} {B} {A} {S} {R} ⟨≈≈⟩ ⨾-assoc₄
ran⨾ran-idempotent : {A B C : Obj} {R : Mor A C} {S : Mor B C}
                   → ran R ⨾ ran S ⨾ ran R ⨾ ran S ≈ ran R ⨾ ran S
ran⨾ran-idempotent {A} {B} {C} {R} {S}
                   = ⨾-assocL₄ ⟨≈≈⟩ dom⨾dom-idempotent {C} {B} {A} {S} {R}
ran-⨾-ran-subswap : {A B C : Obj} {R : Mor A C} {S : Mor B C}
                  → ran (R ⨾ ran S) ⊑ ran R ⨾ ran S
ran-⨾-ran-subswap {A} {B} {C} {R} {S} = dom-⨾-dom-subswap {C} {B} {A} {S} {R}
ran-⨾-ran-swap : {A B C : Obj} {R : Mor A C} {S : Mor B C}
               → ran (R ⨾ ran S) ≈ ran R ⨾ ran S
ran-⨾-ran-swap {A} {B} {C} {R} {S} = dom-⨾-dom-swap {C} {B} {A} {S} {R}
ran-⨾-subcommute : {A B C : Obj} {R : Mor A C} {S : Mor B C} → ran R ⨾ ran S ⊑ ran S ⨾ ran R
ran-⨾-subcommute {A} {B} {C} {R} {S} = dom-⨾-subcommute {C} {B} {A} {S} {R}
ran-⨾-commute : ∀ {A B C : Obj} {R : Mor A C} {S : Mor B C} → ran R ⨾ ran S ≈ ran S ⨾ ran R
ran-⨾-commute {A} {B} {C} {R} {S} = dom-⨾-commute {C} {B} {A} {S} {R}
isSurjectiveR : {A B : Obj} → Mor A B → Set (i ⊍ j ⊍ k₂)
isSurjectiveR R = isSuperidentity (ran R)
isRangeMinimal : {A B : Obj} → Mor A B → Set (j ⊍ k₂)
isRangeMinimal {A} {B} R = ∀ {Q : Mor A B} → Q ⊑ R → R ⨾ ran Q ⊑ Q
isRangeMinimal≈ : {A B : Obj} → {R : Mor A B}
                → isRangeMinimal R → {Q : Mor A B} → Q ⊑ R → R ⨾ ran Q ≈ Q
isRangeMinimal≈ isRangeMinimalR Q⊑R =
  ⊑-antisym (isRangeMinimalR Q⊑R) (ranPreserves ⟨⊑⊑⟩ ⨾-monotone₁ Q⊑R)
```

```
record OSGD {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field
    orderedSemigroupoid : OrderedSemigroupoid j k₁ k₂ Obj
    domainOp            : OSGDomainOp orderedSemigroupoid
  open OrderedSemigroupoid orderedSemigroupoid public
  open OSGDomainOp         domainOp            public
```

To avoid mutual recursion, oppositeOSGRangeOp is not defined inside OSGDomainOp.

```
  oppositeOSGRangeOp : OSGRangeOp (oppositeOrderedSemigroupoid orderedSemigroupoid)
  oppositeOSGRangeOp = record
    {ran              = λ {B} {A}         → dom              {A} {B}
    ; ranSubIdentity  = λ {B} {A} {R}     → swapFromSubid (domSubIdentity {A} {B} {R})
    ; ran-⨾-idempotent = λ {B} {A} {R}     → dom-⨾-idempotent {A} {B} {R}
    ; ranPreserves⊑   = λ {B} {A} {Q} {R} → domPreserves⊑    {A} {B} {Q} {R}
    ; ranLeastPreserver = λ {B} {A} {R} {d} subid
                      → domLeastPreserver {A} {B} {R} {d} (swapToSubid subid)
    ; ranLocality     = λ {C} {B} {A} {S} {R} → domLocality  {A} {B} {C} {R} {S}
    }

record OSGR {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field
    orderedSemigroupoid : OrderedSemigroupoid j k₁ k₂ Obj
    rangeOp             : OSGRangeOp orderedSemigroupoid
  open OrderedSemigroupoid orderedSemigroupoid public
  open OSGRangeOp          rangeOp             public

record OSGDR {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field
    orderedSemigroupoid : OrderedSemigroupoid j k₁ k₂ Obj
    domainOp            : OSGDomainOp orderedSemigroupoid
    rangeOp             : OSGRangeOp orderedSemigroupoid
  open OrderedSemigroupoid orderedSemigroupoid public
  open OSGDomainOp         domainOp            public
  open OSGRangeOp          rangeOp             public
  osgd : OSGD j k₁ k₂ Obj
  osgd = record
    {orderedSemigroupoid = orderedSemigroupoid
    ; domainOp           = domainOp
    }
  osgr : OSGR j k₁ k₂ Obj
  osgr = record
    {orderedSemigroupoid = orderedSemigroupoid
    ; rangeOp = rangeOp
    }
  open OSGD osgd public using (oppositeOSGRangeOp)
  dom-ran : {A B : Obj} {R : Mor A B} → dom (ran R) ≈ ran R
  dom-ran = idempotSubidentity-dom ranSubIdentity ran-⨾-idempotent
  ran-dom : {A B : Obj} {R : Mor A B} → ran (dom R) ≈ dom R
  ran-dom = idempotSubidentity-ran domSubIdentity dom-⨾-idempotent
```

10.3 Categoric.OCD

```
record OCD {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field orderedCategory : OrderedCategory j k₁ k₂ Obj
```

```
  open OrderedCategory orderedCategory
  field
    domainOp : OSGDomainOp orderedSemigroupoid
  osgd : OSGD j k₁ k₂ Obj
  osgd = record
    {orderedSemigroupoid = orderedSemigroupoid
    ; domainOp = domainOp
    }
  open OSGDomainOp  domainOp   public
  open OSGD         osgd       public using (oppositeOSGRangeOp)
  dom-Id : {A : Obj} → dom (Id {A}) ≈ Id
  dom-Id = ⊑-antisym (subidentityIsCoreflexive domSubIdentity) (domPreserves ⟨⊑≈⟩ rightId)
  open OrderedCategory orderedCategory public


record OCR {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field orderedCategory : OrderedCategory j k₁ k₂ Obj
  open OrderedCategory orderedCategory
  field
    rangeOp         : OSGRangeOp orderedSemigroupoid
  osgr : OSGR j k₁ k₂ Obj
  osgr = record
    {orderedSemigroupoid = orderedSemigroupoid
    ; rangeOp = rangeOp
    }
  open OSGRangeOp   rangeOp    public
  ran-Id : {A : Obj} → ran (Id {A}) ≈ Id
  ran-Id = ⊑-antisym (subidentityIsCoreflexive ranSubIdentity) (ranPreserves ⟨⊑≈⟩ leftId)
  open OrderedCategory orderedCategory public
```

The opposite of an OCD is an OCR, and vice versa.

```
oppositeOCR : {i j k₁ k₂ : Level} {Obj : Set i} → OCD j k₁ k₂ Obj → OCR j k₁ k₂ Obj
oppositeOCR ocd = let open OCD ocd in record
  {orderedCategory = oppositeOrderedCategory orderedCategory
  ; rangeOp = oppositeOSGRangeOp
  }


oppositeOCD : {i j k₁ k₂ : Level} {Obj : Set i} → OCR j k₁ k₂ Obj → OCD j k₁ k₂ Obj
oppositeOCD ocr = let open OCR ocr in record
  {orderedCategory = oppositeOrderedCategory orderedCategory
  ; domainOp = oppositeOSGDomainOp
  }


record OCDR {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field orderedCategory : OrderedCategory j k₁ k₂ Obj
  open OrderedCategory orderedCategory
  field
    domainOp : OSGDomainOp orderedSemigroupoid
    rangeOp  : OSGRangeOp orderedSemigroupoid
  ocd : OCD j k₁ k₂ Obj
  ocd = record
    {orderedCategory = orderedCategory
    ; domainOp = domainOp
    }
```

```
  ocr : OCR j k₁ k₂ Obj
  ocr = record
    {orderedCategory = orderedCategory
    ; rangeOp = rangeOp
    }
  osgdr : OSGDR j k₁ k₂ Obj
  osgdr = record
    {orderedSemigroupoid = orderedSemigroupoid
    ; domainOp = domainOp
    ; rangeOp = rangeOp
    }
  open OSGDR osgdr public hiding (domainOp; rangeOp)   -- imports the additional OSGDR material.
    -- Agda-2.3.0 using still takes too much time and memory.
    -- open OCD ocd public using (dom-Id)
    -- open OCR ocr public using (ran-Id)
  dom-Id = λ {A} → OCD.dom-Id ocd {A}
  ran-Id = λ {A} → OCR.ran-Id ocr {A}
```

10.4 Categoric.OSGC.Domain

While oppositeOSGRangeOp takes a domain operator to produce a range operator in the *opposite* ordered semigroupoid, once we also have converse, a domain operator gives rise to a range operator in the same OSGC.

```
module FromDomainOp {i j k₁ k₂ : Level} {Obj : Set i}
  (base : OSGC j k₁ k₂ Obj)
  (domainOp : OSGDomainOp (OSGC.orderedSemigroupoid base)) where
  open OSGC base
  open OSGDomainOp domainOp
  osgd : OSGD j k₁ k₂ Obj
  osgd = record
    {orderedSemigroupoid = orderedSemigroupoid
    ; domainOp           = domainOp
    }
```

Since even idempotent subidentities are not necessarily symmetric, we have to take the converse of the domain of the converse as range, when deriving range from domain.

```
  rangeOp : OSGRangeOp orderedSemigroupoid
  rangeOp = let
    ran : {A B : Obj} → Mor A B → Mor B B
    ran R = (dom (R ˘)) ˘
    in record
    {ran               = ran
    ; ranSubIdentity   = λ {A} {B} {R} → ˘-isSubidentity (domSubIdentity {B} {A} {R ˘})
    ; ran-⨾-idempotent = λ {A} {B} {R} → ≈-begin
          ran R ⨾ ran R
      ≈⟨ ≈-refl ⟩
          (dom (R ˘)) ˘ ⨾ (dom (R ˘)) ˘
      ≈⟨ ≈-sym ˘-involution ⟩
          (dom (R ˘) ⨾ dom (R ˘)) ˘
      ≈⟨ ˘-cong (dom-⨾-idempotent {B} {A} {R ˘}) ⟩
          (dom (R ˘)) ˘
      ≈⟨ ≈-refl ⟩
          ran R
      □
    ; ranPreserves⊑    = λ {A} {B} {Q} {R} Q⊑R → ⊑-begin
```

```
        Q
    ≈⟨ ≈-sym ˘˘ ⟩
        (Q ˘) ˘
    ⊑⟨ ˘-monotone (domPreserves⊑ {B} {A} {Q ˘} {R ˘} (˘-monotone Q⊑R)) ⟩
      ((dom (R ˘)) ⨾ Q ˘) ˘
    ≈⟨ ˘-involutionRightConv ⟩
        Q ⨾ (dom (R ˘)) ˘
    ≈⟨ ≈-refl ⟩
        Q ⨾ ran R
    □
  ; ranLeastPreserver = λ {A} {B} {R} {d} subid d⨾d≈d R⊑R⨾d → ⊑-begin
        ran R
    ≈⟨ ≈-refl ⟩
        (dom (R ˘)) ˘
    ⊑⟨ ˘-monotone (domLeastPreserver {B} {A} {R ˘} {d ˘} (˘-isSubidentity subid)
                    (≈-sym ˘-involution ⟨≈≈⟩ ˘-cong d⨾d≈d)
                    (˘-monotone R⊑R⨾d ⟨⊑≈⟩ ˘-involution))
      ⟩
        (d ˘) ˘
    ≈⟨ ˘˘ ⟩
        d
    □
  ; ranLocality = λ {A} {B} {C} {R} {S} → ⊑-begin
        ran (ran R ⨾ S)
    ≈⟨ ≈-refl ⟩
        (dom (((dom (R ˘)) ˘ ⨾ S) ˘)) ˘
    ≈⟨ ˘-cong (dom-cong ˘-involutionLeftConv) ⟩
        (dom (S ˘ ⨾ dom (R ˘))) ˘
    ⊑⟨ ˘-monotone (domLocality {C} {B} {A} {S ˘} {R ˘}) ⟩
        (dom (S ˘ ⨾ R ˘)) ˘
    ≈⟨ ˘-cong (dom-cong (≈-sym ˘-involution)) ⟩
        (dom ((R ⨾ S) ˘)) ˘
    ≈⟨ ≈-refl ⟩
        ran (R ⨾ S)
    □
  }
```

Since even idempotent subidentities are not necessarily symmetric, we have to take the converse of the domain of the converse as range, when deriving range from domain.

```
osgr : OSGR j k₁ k₂ Obj
osgr = record
  {orderedSemigroupoid = orderedSemigroupoid
  ; rangeOp            = rangeOp
  }
osgdr : OSGDR j k₁ k₂ Obj
osgdr = record
  {orderedSemigroupoid = orderedSemigroupoid
  ; domainOp           = domainOp
  ; rangeOp            = rangeOp
  }
open OSGDR osgdr public using (dom-ran; ran-dom)
```

Chapter 11

Locally Ordered Semigroupoids and Categories with Converse

Many of the "typically relation-algebraic" definitions of concepts like univalence, totality, etc. do not require more than locally ordered semigroupoids with converse (OSGCs), where the only additional axiom is monotony of converse.

The re-exporting module Categoric.OSGC is an interface that allows the user to ignore the internal modularisation; the basic definition of locally ordered semigroupoids with converse with immediate consequences of monotony of converse is in Sect. 11.3. Definitions of properties and of "proof-carrying morphisms" including partial functions and mappings are provided in Sect. 11.4, while sections 11.5–11.9 contain lemmata involving these properties.

A similar development for ordered categories with converse (OCCs) is in 11.10–11.16.

Finally, Sect. 11.17 defines the semigroupoid respectively category of mappings in an OSGC respectively OCC.

11.1 Categoric.OSGC.Monolithic

```
record OSGC′ {i j k₁ k₂ : Level} {Obj : Set i}
                (Hom : Obj → Obj → Poset j k₁ k₂) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field semigroupoid : Semigroupoid′ (λ A B → posetSetoid (Hom A B))
  open Semigroupoid′ semigroupoid hiding (semigroupoid)
  infix 4 _⊑_; infix 10 _˘
  _⊑_ = λ {A} {B} → Poset._≤_ (Hom A B)
  field
    ⨾-monotone : {A B C : Obj} {f f' : Mor A B} {g g' : Mor B C}
               → f ⊑ f' → g ⊑ g' → (f ⨾ g) ⊑ (f' ⨾ g')
    _˘          : {A B : Obj}                              → Mor A B → Mor B A
    ˘˘          : {A B : Obj}    {R : Mor A B}             → (R ˘) ˘ ≈ R
    ˘-involution : {A B C : Obj} {R : Mor A B} {S : Mor B C} → (R ⨾ S) ˘ ≈ S ˘ ⨾ R ˘
    ˘-monotone : {A B : Obj} {R S : Mor A B} → R ⊑ S → (R ˘) ⊑ (S ˘)
  orderedSemigroupoid : OrderedSemigroupoid j k₁ k₂ Obj
  orderedSemigroupoid = record {Hom = Hom; compOp = compOp
    ; locOrd = record {⨾-monotone = ⨾-monotone}}
  convOp : ConvOp (Semigroupoid′.semigroupoid semigroupoid)
  convOp = record {_˘ = _˘
    ; ˘-cong = λ {A} {B} {R} {S} R≈S → Poset.antisym (Hom B A)
        (˘-monotone (Poset.reflexive (Hom A B) R≈S))
        (˘-monotone (Poset.reflexive (Hom A B) (≈-sym R≈S)))
    ; ˘˘ = ˘˘
    ; ˘-involution = ˘-involution
    }
```

```
  open ConvOp convOp using (˘-cong; ˘-coinvolution)
  osgc : OSGC₀.OSGC j k₁ k₂ Obj
  osgc = record {OSGC_Base = record
    {orderedSemigroupoid = orderedSemigroupoid
    ; convOp = convOp
    ; ˘-monotone = ˘-monotone
    }}
  open OSGC₀.OSGC osgc using (Mapping; module Mapping)
```

11.2 Categoric.OSGC

```
record OSGC {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field OSGC_Base : OSGC-Base j k₁ k₂ Obj
  open OSGC-Base                OSGC_Base public
  open OSGC-Props               OSGC_Base public
  open OSGC-Prop-Conversions    OSGC_Base public
  open OSGC-Prop-Lemmas         OSGC_Base public
  open OSGC-CompProps           OSGC_Base public
  open OSGC-MappingProps        OSGC_Base public
  open OSGC-DifunctionalProps   OSGC_Base public

retractOSGC : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
            → (F : Obj₂ → Obj₁) → OSGC j k₁ k₂ Obj₁ → OSGC j k₁ k₂ Obj₂
retractOSGC F base = let open OSGC base in record {OSGC_Base = retractOSGC-Base F OSGC_Base}

attachOSGC : {i j k₁ k₂ : Level} {Obj : Set i}
           → OSGC j k₁ k₂ Obj → OSGC (i ⊍ j) k₁ k₂ Obj
attachOSGC base = let open OSGC base in record {OSGC_Base = attachOSGC-Base OSGC_Base}
```

11.3 Categoric.OSGC.Base

```
record OSGC-Base {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field orderedSemigroupoid : OrderedSemigroupoid j k₁ k₂ Obj
  open OrderedSemigroupoid orderedSemigroupoid
  field convOp : ConvOp semigroupoid
  open ConvOp convOp
  field
    ˘-monotone : {A B : Obj} {R S : Mor A B} → R ⊑ S → (R ˘) ⊑ (S ˘)
  ˘-isotone : {A B : Obj} {R S : Mor A B} → (R ˘) ⊑ (S ˘) → R ⊑ S
  ˘-isotone {A} {B} {R} {S} R⊑S = ⊑-trans₂ (≈-sym ˘˘) (⊑-trans₁ (˘-monotone R⊑S) ˘˘)
  ⊑-˘-swap : {A B : Obj} {R : Mor B A} {S : Mor A B} → (R ˘) ⊑ S → R ⊑ (S ˘)
  ⊑-˘-swap R⊑S˘ = ⊑-trans₂ (≈-sym ˘˘) (˘-monotone R⊑S˘)
  ˘-⊑-swap : {A B : Obj} {R : Mor A B} {S : Mor B A} → R ⊑ (S ˘) → (R ˘) ⊑ S
  ˘-⊑-swap R˘⊑S = ⊑-trans₁ (˘-monotone R˘⊑S) ˘˘
  convSemigroupoid : ConvSemigroupoid j k₁ Obj
  convSemigroupoid = record
    {semigroupoid = semigroupoid
    ; convOp = convOp
    }
  open OrderedSemigroupoid orderedSemigroupoid public
  open ConvOp              convOp              public
```

```
retractOSGC-Base : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                 → (F : Obj₂ → Obj₁) → OSGC-Base j k₁ k₂ Obj₁ → OSGC-Base j k₁ k₂ Obj₂
retractOSGC-Base F base = let open OSGC-Base base in record
  {orderedSemigroupoid = retractOrderedSemigroupoid F orderedSemigroupoid
  ; convOp = retractConvOp F convOp
  ; ˘-monotone = ˘-monotone
  }


attachOSGC-Base : {i j k₁ k₂ : Level} {Obj : Set i}
                → OSGC-Base j k₁ k₂ Obj → OSGC-Base (i ⊍ j) k₁ k₂ Obj
attachOSGC-Base base = let open OSGC-Base base in record
  {orderedSemigroupoid = attachOrderedSemigroupoid orderedSemigroupoid
  ; convOp              = attachConvOp convOp
  ; ˘-monotone          = ˘-monotone
  }
```

11.4 Categoric.OSGC.Props

```
module OSGC-Props {i j k₁ k₂ : Level} {Obj : Set i} (Base : OSGC-Base j k₁ k₂ Obj) where
  open OSGC-Base Base

  isUnivalent : {A B : Obj} → Mor A B → Set (i ⊍ j ⊍ k₂)
  isUnivalent R = isSubidentity (R ˘ ⨾ R)
  isTotal : {A B : Obj} → Mor A B → Set (i ⊍ j ⊍ k₂)
  isTotal R = isSuperidentity (R ⨾ R ˘)
  isMapping : {A B : Obj} → Mor A B → Set (i ⊍ j ⊍ k₂)
  isMapping R = isUnivalent R × isTotal R
  isInjective : {A B : Obj} → Mor A B → Set (i ⊍ j ⊍ k₂)
  isInjective R = isSubidentity (R ⨾ R ˘)
  isSurjective : {A B : Obj} → Mor A B → Set (i ⊍ j ⊍ k₂)
  isSurjective R = isSuperidentity (R ˘ ⨾ R)
  isBijective : {A B : Obj} → Mor A B → Set (i ⊍ j ⊍ k₂)
  isBijective R = isInjective R × isSurjective R
  isPIso : {A B : Obj} → Mor A B → Set (i ⊍ j ⊍ k₂)
  isPIso R = isUnivalent R × isInjective R


  record Univalent (A B : Obj) : Set (i ⊍ j ⊍ k₂) where
    constructor mkUnivalent
    field
      mor : Mor A B
      prf : isUnivalent mor
  record Mapping (A B : Obj) : Set (i ⊍ j ⊍ k₂) where
    constructor mkMapping
    field
      mor : Mor A B
      prf : isMapping mor
    unival : isUnivalent mor
    unival = proj₁ prf
    total : isTotal mor
    total = proj₂ prf
    Unival : Univalent A B
    Unival = record
      {mor = mor
```

```
    ; prf = unival
    }
record PIso (A B : Obj) : Set (i ⊍ j ⊍ k₂) where
  constructor mkPIso
  field
    mor : Mor A B
    prf : isPIso mor
  unival : isUnivalent mor
  unival = proj₁ prf
  inj : isInjective mor
  inj = proj₂ prf
  Unival : Univalent A B
  Unival = record
    {mor = mor
    ; prf = proj₁ prf
    }

isDifunctional : {A B : Obj} → Mor A B → Set k₂
isDifunctional R = R ⨾ R ˘ ⨾ R ⊑ R
record Difun (A B : Obj) : Set (i ⊍ j ⊍ k₂) where
  constructor mkDifun
  field
    mor : Mor A B
    prf : isDifunctional mor

isCodifunctional : {A B : Obj} → Mor A B → Set k₂
isCodifunctional R = R ⊑ R ⨾ R ˘ ⨾ R
```

We define an abstract version of "partial equivalence relations" to be symmetric, transitive, and codifunctional: **[WK:** *Reorganise and refer to Sect. 3.15!* **]**

```
record isPER {A : Obj} (R : Mor A A) : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
    field
      isPER-symmetric      : isSymmetric R
      isPER-transitive     : isTransitive R
      isPER-codifunctional : isCodifunctional R
    isPER-R⨾R˘⨾R⊑R⨾R :  R ⨾ R ˘ ⨾ R ⊑ R ⨾ R
    isPER-R⨾R˘⨾R⊑R⨾R = ⨾-monotone₂ (⊑-trans₂ (⨾-cong₁ isPER-symmetric) isPER-transitive)
    isPER-idempotent : isIdempotent R
    isPER-idempotent = ⊑-antisym isPER-transitive
      (⊑-trans isPER-codifunctional isPER-R⨾R˘⨾R⊑R⨾R)
    isPER-difunctional : isDifunctional R
    isPER-difunctional = ⊑-trans isPER-R⨾R˘⨾R⊑R⨾R isPER-transitive
```

These are exactly the symmetric idempotents:

```
symIdempot-isPER : {A : Obj} {R : Mor A A} → isSymmetric R → isIdempotent R → isPER R
symIdempot-isPER sym idem = let idem′ = ⊑-reflexive′ idem in record
  {isPER-symmetric      = sym
  ; isPER-transitive     = ⊑-reflexive idem
  ; isPER-codifunctional = ⊑-trans idem′
    (⨾-monotone₂ (⊑-trans idem′ (⨾-monotone₁ (⊑-reflexive′ sym))))
  }
record PER (A : Obj) : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  constructor mkPER
  field
    mor : Mor A A
    prf : isPER mor
```

11.5 Categoric.OSGC.Props.Conversions

```
module OSGC-Prop-Conversions {i j k₁ k₂ : Level} {Obj : Set i}
                              (Base : OSGC-Base j k₁ k₂ Obj) where
  open OSGC-Base Base
  open OSGC-Props Base
```

For symmetry, an inclusion is sufficient:

```
  isSymmetric⊑ : {A : Obj} {R : Mor A A} → R ˘ ⊑ R → isSymmetric R
  isSymmetric⊑ R˘⊑R = ⊑-antisym R˘⊑R (⊑-˘-swap R˘⊑R)

  isSymmetric⊑˘ : {A : Obj} {R : Mor A A} → R ⊑ R ˘ → isSymmetric R
  isSymmetric⊑˘ R⊑R˘ = ⊑-antisym (˘-⊑-swap R⊑R˘) R⊑R˘
```

Taking the converse of a one-sided subidentity makes it switch sides:

```
  ˘-isLeftSubidentity    : {A : Obj} {p : Mor A A}
                         → isRightSubidentity p → isLeftSubidentity (p ˘)
  ˘-isLeftSubidentity    {A} {p} right {B} {R} = ⊑-begin
        p ˘ ⨾ R             ≈⟨ ⨾-cong₂ (≈-sym ˘˘) ⟩  p ˘ ⨾ (R ˘) ˘
                            ≈⟨ ≈-sym ˘-involution ⟩  (R ˘ ⨾ p) ˘
                            ⊑⟨ ˘-monotone right ⟩    (R ˘) ˘
                            ≈⟨ ˘˘ ⟩                  R                    □

  ˘-isRightSubidentity  : {A : Obj} {p : Mor A A}
                        → isLeftSubidentity p → isRightSubidentity (p ˘)
  ˘-isRightSubidentity  {A} {p} left {B} {S} = ⊑-begin
        S ⨾ p ˘             ≈⟨ ⨾-cong₁ (≈-sym ˘˘) ⟩  (S ˘) ˘ ⨾ p ˘
                            ≈⟨ ≈-sym ˘-involution ⟩  (p ⨾ S ˘) ˘
                            ⊑⟨ ˘-monotone left ⟩     (S ˘) ˘
                            ≈⟨ ˘˘ ⟩                  S                    □

  ˘-isSubidentity : {A : Obj} → {p : Mor A A} → isSubidentity p → isSubidentity (p ˘)
  ˘-isSubidentity (left, right) = (λ {B} {R} → ˘-isLeftSubidentity right {B} {R})
                                ,  (λ {B} {S} → ˘-isRightSubidentity left {B} {S})

  isSubidentity-˘ : {A : Obj} → {p : Mor A A} → isSubidentity (p ˘) → isSubidentity p
  isSubidentity-˘ {A} {p} prf = ≈-isSubidentity (≈-sym ˘˘) (˘-isSubidentity {A} {p ˘} prf)
```

Symmetric one-sided subidentities are both-sided:

```
  isRightSubidentitySymLeft : {A : Obj} → {p : Mor A A}
                            → p ˘ ≈ p → isLeftSubidentity p → isRightSubidentity p
  isRightSubidentitySymLeft {A} {p} pSym left {B} {S} = ⊑-begin
        S ⨾ p ≈⟨ ≈-sym ˘-coinvolution ⟩                    (p ˘ ⨾ S ˘) ˘
              ⊑⟨ ˘-monotone (≈-isLeftSubidentity pSym left) ⟩ (S ˘) ˘
              ≈⟨ ˘˘ ⟩                                        S          □

  isSubidentitySymLeft : {A : Obj} {p : Mor A A}
                       → p ˘ ≈ p → isLeftSubidentity p → isSubidentity p
  isSubidentitySymLeft {A} {p} pSym left = (λ {B} {R} → left {B} {R})
                                         , (λ {B} {S} → isRightSubidentitySymLeft pSym left {B} {S})
```

Taking the converse of a one-sided superidentity makes it switch sides:

```
  ˘-isLeftSuperidentity  : {A : Obj} {p : Mor A A}
                         → isRightSuperidentity p → isLeftSuperidentity (p ˘)
  ˘-isLeftSuperidentity {A} {p} right {B} {R} = ⊑-begin
        R                   ≈⟨ ≈-sym ˘˘ ⟩            (R ˘) ˘
                            ⊑⟨ ˘-monotone right ⟩    (R ˘ ⨾ p) ˘
                            ≈⟨ ˘-involution ⟩        p ˘ ⨾ (R ˘) ˘
```

```
                         ≈⟨ ⨾-cong₂ ˘˘ ⟩            p ˘ ⨾ R              □
˘-isRightSuperidentity  : {A : Obj} {p : Mor A A}
                         → isLeftSuperidentity p  →  isRightSuperidentity (p ˘)
˘-isRightSuperidentity  {A} {p} left {B} {S}  =  ⊑-begin
   S                     ≈⟨ ≈-sym ˘˘ ⟩              (S ˘) ˘
                         ⊑⟨ ˘-monotone left ⟩       (p ⨾ S ˘) ˘
                         ≈⟨ ˘-involution ⟩          (S ˘) ˘ ⨾ p ˘
                         ≈⟨ ⨾-cong₁ ˘˘ ⟩            S ⨾ p ˘              □
˘-isSuperidentity  : {A : Obj} {p : Mor A A} → isSuperidentity p  →  isSuperidentity (p ˘)
˘-isSuperidentity (left, right)  =  (λ {B} {R} → ˘-isLeftSuperidentity right {B} {R})
                                 ,  (λ {B} {S} → ˘-isRightSuperidentity left {B} {S})
isSuperidentity-˘  : {A : Obj} → {p : Mor A A} → isSuperidentity (p ˘)  →  isSuperidentity p
isSuperidentity-˘ {A} {p} prf  =  ≈-isSuperidentity (≈-sym ˘˘) (˘-isSuperidentity {A} {p ˘} prf)
```

Symmetric one-sided superidentities are both-sided:

```
isRightSuperidentitySymLeft : {A : Obj} {p : Mor A A}
                            → p ˘ ≈ p → isLeftSuperidentity p  →  isRightSuperidentity p
isRightSuperidentitySymLeft {A} {p} pSym left {B} {S}  =  ⊑-begin
   S ≈⟨ ≈-sym ˘˘ ⟩                                          (S ˘) ˘
     ⊑⟨ ˘-monotone (≈-isLeftSuperidentity pSym left) ⟩ (p ˘ ⨾ S ˘) ˘
     ≈⟨ ˘-coinvolution ⟩                                    S ⨾ p        □
isSuperidentitySymLeft : {A : Obj} {p : Mor A A}
                       → p ˘ ≈ p → isLeftSuperidentity p  →  isSuperidentity p
isSuperidentitySymLeft {A} {p} pSym left
    =  (λ {B} {R} → left {B} {R})
   , (λ {B} {S} → isRightSuperidentitySymLeft pSym left {B} {S})
```

The properties of Sect. 11.4 are invariant under morphism equivalence _≈_:

```
≈-isUnivalent     :   {A B : Obj} → {P Q : Mor A B} → Q ≈ P → isUnivalent P → isUnivalent Q
≈-isUnivalent qp  =   ≈-isSubidentity (⨾-cong (˘-cong qp) qp)
≈-isInjective     :   {A B : Obj} → {P Q : Mor A B} → Q ≈ P → isInjective P → isInjective Q
≈-isInjective qp  =   ≈-isSubidentity (⨾-cong qp (˘-cong qp))
≈-isTotal         :   {A B : Obj} → {P Q : Mor A B} → Q ≈ P → isTotal P → isTotal Q
≈-isTotal qp      =   ≈-isSuperidentity (⨾-cong qp (˘-cong qp))
≈-isSurjective    :   {A B : Obj} → {P Q : Mor A B} → Q ≈ P → isSurjective P → isSurjective Q
≈-isSurjective qp =   ≈-isSuperidentity (⨾-cong (˘-cong qp) qp)
≈-isMapping : {A B : Obj} → {P Q : Mor A B} → Q ≈ P → isMapping P → isMapping Q
≈-isMapping qp (u, t)        =  ≈-isUnivalent qp u, ≈-isTotal qp t
≈-isBijective : {A B : Obj} → {P Q : Mor A B} → Q ≈ P → isBijective P → isBijective Q
≈-isBijective qp (i, s)      =  ≈-isInjective qp i, ≈-isSurjective qp s
```

The right combinations of sub- and super-identities produce identities:

```
univalSurj-identity      : {A B : Obj} {P : Mor A B}
                         → isUnivalent P → isSurjective P → isIdentity (P ˘ ⨾ P)
univalSurj-identity      isUnival isSurj  =  ⊑⊒-isIdentity isUnival isSurj
totalInj-identity        : {A B : Obj} {P : Mor A B}
                         → isTotal P → isInjective P → isIdentity (P ⨾ P ˘)
totalInj-identity        isTotal isInj  =  ⊑⊒-isIdentity isInj isTotal
bijMapping-identities : {A B : Obj} {P : Mor A B} → isBijective P → isMapping P
                      → (isIdentity (P ⨾ P ˘) × isIdentity (P ˘ ⨾ P))
bijMapping-identities (isInj, isSurj) (isUnival, isTotal)      =  totalInj-identity isTotal isInj
                                                               , univalSurj-identity isUnival isSurj
```

Identities also are mappings:

```
isIdentity-isUnivalent : {A : Obj} {I : Mor A A} → isIdentity I → isUnivalent I
isIdentity-isUnivalent {A} {I} (leftId, rightId) =
  (λ {B} {R} → ⊑-begin
     (I ˘ ⨾ I) ⨾ R
  ≈⟨ ⨾-cong₁ rightId ⟩
     I ˘ ⨾ R
  ≈⟨ ⨾-cong₁ (isLeftIdentity-isSymmetric leftId) ⟨≈≈⟩ leftId ⟩
     R
  □),
  (λ {B} {S} → ⊑-begin
     S ⨾ (I ˘ ⨾ I)
  ≈⟨ ⨾-cong₂ rightId ⟩
     S ⨾ I ˘
  ≈⟨ ⨾-cong₂ (isRightIdentity-isSymmetric rightId) ⟨≈≈⟩ rightId ⟩
     S
  □)
isIdentity-isTotal : {A : Obj} {I : Mor A A} → isIdentity I → isTotal I
isIdentity-isTotal {A} {I} (leftId, rightId) =
  (λ {B} {R} → ⊑-begin
     R
  ≈˘⟨ ⨾-cong₁ (isLeftIdentity-isSymmetric leftId) ⟨≈≈⟩ leftId ⟩
     I ˘ ⨾ R
  ≈˘⟨ ⨾-cong₁ leftId ⟩
     (I ⨾ I ˘) ⨾ R
  □),
  (λ {B} {S} → ⊑-begin
     S
  ≈˘⟨ ⨾-cong₂ (isRightIdentity-isSymmetric rightId) ⟨≈≈⟩ rightId ⟩
     S ⨾ I ˘
  ≈˘⟨ ⨾-cong₂ leftId ⟩
     S ⨾ (I ⨾ I ˘)
  □)
isIdentity-isMapping : {A : Obj} {I : Mor A A} → isIdentity I → isMapping I
isIdentity-isMapping {A} {I} isId = isIdentity-isUnivalent isId, isIdentity-isTotal isId
isIdentity-Mapping : {A : Obj} {I : Mor A A} → isIdentity I → Mapping A A
isIdentity-Mapping {A} {I} isId = OSGC-Props.mkMapping I (isIdentity-isMapping isId)
```

Conversions between isUnivalent and isInjective:

```
  isInjectiveFromUnivalent : {A C : Obj} → {Q : Mor A C} → isUnivalent (Q ˘) → isInjective Q
  isInjectiveFromUnivalent {A} {C} {Q} (left, right) = isSubidentitySymLeft ˘-involutionRightConv
    (λ {B} {R} → ⊑-begin
      (Q ⨾ Q ˘) ⨾ R       ≈⟨ ⨾-cong₁₁ (≈-sym ˘˘) ⟩ ((Q ˘) ˘ ⨾ Q ˘) ⨾ R
                          ⊑⟨ left ⟩              R                        □)
  isInjectiveToUnivalent : {A C : Obj} → {Q : Mor A C} → isInjective Q → isUnivalent (Q ˘)
  isInjectiveToUnivalent {A} {C} {Q} (left, right) = isSubidentitySymLeft ˘-involutionLeftConv
    (λ {B} {R} → ⊑-begin
      ((Q ˘) ˘ ⨾ Q ˘) ⨾ R ≈⟨ ⨾-cong₁₁ ˘˘ ⟩            (Q ⨾ Q ˘) ⨾ R
                          ⊑⟨ left ⟩              R                        □)
  isUnivalentFromInjective : {C A : Obj} → {Q : Mor C A} → isInjective (Q ˘) → isUnivalent Q
  isUnivalentFromInjective {C} {A} {Q} (left, right) = isSubidentitySymLeft ˘-involutionLeftConv
    (λ {B} {R} → ⊑-begin
      (Q ˘ ⨾ Q) ⨾ R       ≈⟨ ⨾-cong₁₂ (≈-sym ˘˘) ⟩ (Q ˘ ⨾ (Q ˘) ˘) ⨾ R
                          ⊑⟨ left ⟩              R                        □)
  isUnivalentToInjective : {C A : Obj} → {Q : Mor C A} → isUnivalent Q → isInjective (Q ˘)
  isUnivalentToInjective {C} {A} {Q} (left, right) = isSubidentitySymLeft ˘-involutionRightConv
    (λ {B} {R} → ⊑-begin
```

```
          (Q ˘ ⨾ (Q ˘) ˘) ⨾ R ≈⟨ ⨾-cong₁₂ ˘˘ ⟩          (Q ˘ ⨾ Q) ⨾ R
                              ⊑⟨ left ⟩                  R                     □)
```

Conversions between isSurjective and isTotal:

```
isSurjectiveFromTotal : {C A : Obj} → {Q : Mor C A} → isTotal (Q ˘) → isSurjective Q
isSurjectiveFromTotal {C} {A} {Q} (left, right) = isSuperidentitySymLeft ˘-involutionLeftConv
  (λ {B} {R} → ⊑-begin
    R                 ⊑⟨ left ⟩                  (Q ˘ ⨾ (Q ˘) ˘) ⨾ R
                      ≈⟨ ⨾-cong₁₂ ˘˘ ⟩           (Q ˘ ⨾ Q) ⨾ R       □)
isSurjectiveToTotal : {C A : Obj} → {Q : Mor C A} → isSurjective Q → isTotal (Q ˘)
isSurjectiveToTotal {C} {A} {Q} (left, right) = isSuperidentitySymLeft ˘-involutionRightConv
  (λ {B} {R} → ⊑-begin
    R                 ⊑⟨ left ⟩                  (Q ˘ ⨾ Q) ⨾ R
                      ≈⟨ ⨾-cong₁₂ (≈-sym ˘˘) ⟩ (Q ˘ ⨾ (Q ˘) ˘) ⨾ R □)
isTotalFromSurjective : {A C : Obj} → {Q : Mor A C} → isSurjective (Q ˘) → isTotal Q
isTotalFromSurjective {A} {C} {Q} (left, right) = isSuperidentitySymLeft ˘-involutionRightConv
  (λ {B} {R} → ⊑-begin
    R                 ⊑⟨ left ⟩                  ((Q ˘) ˘ ⨾ Q ˘) ⨾ R
                      ≈⟨ ⨾-cong₁₁ ˘˘ ⟩           (Q ⨾ Q ˘) ⨾ R       □)
isTotalToSurjective : {A C : Obj} → {Q : Mor A C} → isTotal Q → isSurjective (Q ˘)
isTotalToSurjective {A} {C} {Q} (left, right) = isSuperidentitySymLeft ˘-involutionLeftConv
  (λ {B} {R} → ⊑-begin
    R                 ⊑⟨ left ⟩                  (Q ⨾ Q ˘) ⨾ R
                      ≈⟨ ⨾-cong₁₁ (≈-sym ˘˘) ⟩ ((Q ˘) ˘ ⨾ Q ˘) ⨾ R □)
```

isBijective and isMapping are opposites:

```
˘-isBijective : {C A : Obj} → {Q : Mor C A} → isMapping Q → isBijective (Q ˘)
˘-isBijective (isUnival, isTotal) = isUnivalentToInjective isUnival, isTotalToSurjective isTotal
˘-isMapping : {A C : Obj} → {Q : Mor A C} → isBijective Q → isMapping (Q ˘)
˘-isMapping (isInj, isSurj) = isInjectiveToUnivalent isInj, isSurjectiveToTotal isSurj
```

11.6 Categoric.OSGC.Props.Lemmas

```
module OSGC-Prop-Lemmas {i j k₁ k₂ : Level} {Obj : Set i}
                        (Base : OSGC-Base j k₁ k₂ Obj) where
  open OSGC-Base Base
  open OSGC-Props Base
```

If a total morphism is included in a univalent morphism, they are equal:

```
total⊑unival-⊒ : {A B : Obj} {R S : Mor A B} → isTotal R → isUnivalent S → R ⊑ S → S ⊑ R
total⊑unival-⊒ {A} {B} {R} {S} isTotal-R isUnivalent-S R⊑S = ⊑-begin
      S
    ⊑⟨ proj₁ isTotal-R ⟨⊑≈⟩ ⨾-assoc ⟩
      R ⨾ R ˘ ⨾ S
    ⊑⟨ ⨾-monotone₂₁ (˘-monotone R⊑S) ⟩
      R ⨾ S ˘ ⨾ S
    ⊑⟨ proj₂ isUnivalent-S ⟩
      R
    □
total⊑unival-≈ : {A B : Obj} {R S : Mor A B} → isTotal R → isUnivalent S → R ⊑ S → R ≈ S
total⊑unival-≈ isTotal-R isUnivalent-S R⊑S = ⊑-antisym R⊑S (total⊑unival-⊒ isTotal-R isUnivalent-S R⊑S)
```

11.7 Categoric.OSGC.Props.Comp

```
module OSGC-CompProps {i j k₁ k₂ : Level} {Obj : Set i} (Base : OSGC-Base j k₁ k₂ Obj) where
  open OSGC-Base Base
  open OSGC-Props Base
  open OSGC-Prop-Conversions Base
  ⨾-isUnivalent  :  {A B C : Obj} → {P : Mor A B} → {Q : Mor B C}
                 →  isUnivalent P → isUnivalent Q → isUnivalent (P ⨾ Q)
  ⨾-isUnivalent {A} {_} {C} {P} {Q} (leftP, rightP) (leftQ, rightQ)
     = isSubidentitySymLeft ˘-involutionLeftConv
    (λ {B} {R} → ⊑-begin
        ((P ⨾ Q) ˘ ⨾ (P ⨾ Q)) ⨾ R
      ≈⟨ ⨾-cong₁₁ ˘-involution ⟩
        ((Q ˘ ⨾ P ˘) ⨾ P ⨾ Q) ⨾ R
      ≈⟨ ⨾-cong₁ ⨾-assoc ⟩
        (Q ˘ ⨾ (P ˘ ⨾ P ⨾ Q)) ⨾ R
      ≈⟨ ⨾-cong₁₂ ⨾-assocL ⟩
        (Q ˘ ⨾ (P ˘ ⨾ P) ⨾ Q) ⨾ R
      ⊑⟨ ⨾-monotone₁₂ leftP ⟩
        (Q ˘ ⨾ Q) ⨾ R
      ⊑⟨ leftQ ⟩
        R
      □
    )
  ⨾-isInjective  :  {A B C : Obj} → {P : Mor A B} → {Q : Mor B C}
                 →  isInjective P → isInjective Q → isInjective (P ⨾ Q)
  ⨾-isInjective {A} {_} {C} {P} {Q} injP injQ = isInjectiveFromUnivalent
    (≈-isUnivalent ˘-involution
      (⨾-isUnivalent (isInjectiveToUnivalent injQ) (isInjectiveToUnivalent injP)))
  ⨾-isTotal  :  {A B C : Obj} → {P : Mor A B} → {Q : Mor B C}
             →  isTotal P → isTotal Q → isTotal (P ⨾ Q)
  ⨾-isTotal {A} {_} {C} {P} {Q} (leftP, _) (leftQ, _) = isSuperidentitySymLeft ˘-involutionRightConv
    (λ {B} {R} → ⊑-begin
        R
      ⊑⟨ leftP ⟩
        (P ⨾ P ˘) ⨾ R
      ⊑⟨ ⨾-monotone₁₂ leftQ ⟩
        (P ⨾ (Q ⨾ Q ˘) ⨾ P ˘) ⨾ R
      ≈⟨ ⨾-cong₁₂ ⨾-assoc ⟩
        (P ⨾ Q ⨾ (Q ˘ ⨾ P ˘)) ⨾ R
      ≈⟨ ⨾-cong₁ ⨾-assocL ⟩
        ((P ⨾ Q) ⨾ (Q ˘ ⨾ P ˘)) ⨾ R
      ≈⟨ ⨾-cong₁₂ (≈-sym ˘-involution) ⟩
        ((P ⨾ Q) ⨾ (P ⨾ Q) ˘) ⨾ R
      □
    )
  ⨾-isSurjective      : {A B C : Obj} → {P : Mor A B} → {Q : Mor B C}
                      → isSurjective P → isSurjective Q → isSurjective (P ⨾ Q)
  ⨾-isSurjective {a} {B} {C} {P} {Q} surjP surjQ = isSurjectiveFromTotal
    (≈-isTotal ˘-involution (⨾-isTotal (isSurjectiveToTotal surjQ) (isSurjectiveToTotal surjP)))
  ⨾-isMapping    : {A B C : Obj} → {P : Mor A B} → {Q : Mor B C}
                 → isMapping P → isMapping Q → isMapping (P ⨾ Q)
  ⨾-isMapping (univalP, totalP) (univalQ, totalQ) =
    (⨾-isUnivalent univalP univalQ, ⨾-isTotal totalP totalQ)
  ⨾-isBijective       : {A B C : Obj} → {P : Mor A B} → {Q : Mor B C}
                      → isBijective P → isBijective Q → isBijective (P ⨾ Q)
  ⨾-isBijective (injP, surjP) (injQ, surjQ) =
```

```
  (⨾-isInjective injP injQ, ⨾-isSurjective surjP surjQ)
swap-⊑-⨾-unival   : {A B C : Obj} → {Q : Mor A C} → {R : Mor A B} → {S : Mor B C}
                  → isUnivalent R → Q ⊑ R ⨾ S → R ˘ ⨾ Q ⊑ S
swap-⊑-⨾-unival {A} {B} {C} {Q} {R} {S} (left, right) q⊑rs = ⊑-begin
      R ˘ ⨾ Q
    ⊑⟨ ⨾-monotone₂ q⊑rs ⟩
      R ˘ ⨾ (R ⨾ S)
    ≈⟨ ≈-sym ⨾-assoc ⟩
      (R ˘ ⨾ R) ⨾ S
    ⊑⟨ left ⟩
      S
    □
swap-⊑-⨾-inj˘ : {A B C : Obj} → {Q : Mor A C} → {R : Mor B A} → {S : Mor B C}
              → isInjective R → Q ⊑ R ˘ ⨾ S → R ⨾ Q ⊑ S
swap-⊑-⨾-inj˘ inj incl = ⊑-trans     (⊑-reflexive (⨾-cong₁ (≈-sym ˘˘)))
                                       (swap-⊑-⨾-unival (isInjectiveToUnivalent inj) incl)
swap-⊑-⨾-inj    : {A B C : Obj} → {Q : Mor A C} → {R : Mor A B} → {S : Mor B C}
                → isInjective S → Q ⊑ R ⨾ S → Q ⨾ S ˘ ⊑ R
swap-⊑-⨾-inj {A} {B} {C} {Q} {R} {S} (left, right) q⊑rs = ⊑-begin
      Q ⨾ S ˘
    ⊑⟨ ⨾-monotone₁ q⊑rs ⟩
      (R ⨾ S) ⨾ S ˘
    ≈⟨ ⨾-assoc ⟩
      R ⨾ (S ⨾ S ˘)
    ⊑⟨ right ⟩
      R
    □
swap-⊑-⨾-unival˘ : {A B C : Obj} → {Q : Mor A C} → {R : Mor A B} → {S : Mor C B}
                 → isUnivalent S → Q ⊑ R ⨾ S ˘ → Q ⨾ S ⊑ R
swap-⊑-⨾-unival˘ unival incl = ⊑-trans   (⊑-reflexive (⨾-cong₂ (≈-sym ˘˘)))
                                          (swap-⊑-⨾-inj (isUnivalentToInjective unival) incl)
swap-⨾-⊑-surj : {A B C : Obj} → {Q : Mor A B} → {R : Mor B C} → {S : Mor A C}
              → isSurjective Q → Q ⨾ R ⊑ S → R ⊑ Q ˘ ⨾ S
swap-⨾-⊑-surj {A} {B} {C} {Q} {R} {S} (left, right) qr⊑s = ⊑-begin
      R
    ⊑⟨ left ⟩
      (Q ˘ ⨾ Q) ⨾ R
    ≈⟨ ⨾-assoc ⟩
      Q ˘ ⨾ (Q ⨾ R)
    ⊑⟨ ⨾-monotone₂ qr⊑s ⟩
      Q ˘ ⨾ S
    □
swap-⨾-⊑-total˘   : {A B C : Obj} → {Q : Mor B A} → {R : Mor B C} → {S : Mor A C}
                  → isTotal Q → Q ˘ ⨾ R ⊑ S → R ⊑ Q ⨾ S
swap-⨾-⊑-total˘ total incl = ⊑-trans (swap-⨾-⊑-surj (isTotalToSurjective total) incl)
                                     (⊑-reflexive (⨾-cong₁ ˘˘))
swap-⨾-⊑-total     : {A B C : Obj} → {Q : Mor A B} → {R : Mor B C} → {S : Mor A C}
                   → isTotal R → Q ⨾ R ⊑ S → Q ⊑ S ⨾ R ˘
swap-⨾-⊑-total {A} {B} {C} {Q} {R} {S} (left, right) qr⊑s = ⊑-begin
      Q
    ⊑⟨ right ⟩
      Q ⨾ (R ⨾ R ˘)
    ≈⟨ ≈-sym ⨾-assoc ⟩
      (Q ⨾ R) ⨾ R ˘
    ⊑⟨ ⨾-monotone₁ qr⊑s ⟩
      S ⨾ R ˘
    □
```

```
swap-⨾-⊑-surj˘     : {A B C : Obj} → {Q : Mor A B} → {R : Mor C B} → {S : Mor A C}
                   → isSurjective R → Q ⨾ R ˘ ⊑ S → Q ⊑ S ⨾ R
swap-⨾-⊑-surj˘ surj incl = ⊑-trans (swap-⨾-⊑-total (isSurjectiveToTotal surj) incl)
                                    (⊑-reflexive (⨾-cong₂ ˘˘))
swap-≈-⨾-univalSurj : {A B C : Obj} → {Q : Mor A C} → {R : Mor A B} → {S : Mor B C}
                    → isUnivalent R → isSurjective R → Q ≈ R ⨾ S → R ˘ ⨾ Q ≈ S
swap-≈-⨾-univalSurj univalR surjR Q≈RS
    = ⊑-antisym (swap-⊑-⨾-unival univalR (⊑-reflexive Q≈RS))
                (swap-⨾-⊑-surj   surjR (⊑-reflexive (≈-sym Q≈RS)))
swap-≈-⨾-totalInj˘ : {A B C : Obj} → {Q : Mor A C} → {R : Mor B A} → {S : Mor B C}
                   → isTotal R → isInjective R → Q ≈ R ˘ ⨾ S → R ⨾ Q ≈ S
swap-≈-⨾-totalInj˘ totalR injR equ =
  ≈-trans (⨾-cong₁ (≈-sym ˘˘))
    (swap-≈-⨾-univalSurj (isInjectiveToUnivalent injR) (isTotalToSurjective totalR) equ)
swap-≈-⨾-totalInj     : {A B C : Obj} → {Q : Mor A C} → {R : Mor A B} → {S : Mor B C}
                      → isTotal S → isInjective S → Q ≈ R ⨾ S → Q ⨾ S ˘ ≈ R
swap-≈-⨾-totalInj totalS injS Q≈RS
    = ⊑-antisym (swap-⊑-⨾-inj injS       (⊑-reflexive Q≈RS))
                (swap-⨾-⊑-total totalS (⊑-reflexive (≈-sym Q≈RS)))
swap-≈-⨾-univalSurj˘ : {A B C : Obj} → {Q : Mor A C} → {R : Mor A B} → {S : Mor C B}
                     → isUnivalent S → isSurjective S → Q ≈ R ⨾ S ˘ → Q ⨾ S ≈ R
swap-≈-⨾-univalSurj˘ univalS surjS equ =
  ≈-trans (⨾-cong₂ (≈-sym ˘˘))
       (swap-≈-⨾-totalInj (isSurjectiveToTotal surjS) (isUnivalentToInjective univalS) equ)
```

11.8 Categoric.OSGC.Props.Mapping

```
module OSGC-MappingProps {i j k₁ k₂ : Level} {Obj : Set i} (Base : OSGC-Base j k₁ k₂ Obj) where
  open OSGC-Base Base
  open OSGC-Props Base
  mappingUnivalent : {A B : Obj} → (R : Mapping A B) → isUnivalent (Mapping.mor R)
  mappingUnivalent {A} {B} R = proj₁ (Mapping.prf R)
  mappingTotal : {A B : Obj} → (R : Mapping A B) → isTotal (Mapping.mor R)
  mappingTotal {A} {B} R = proj₂ (Mapping.prf R)
  mappingDifunctional : {A B : Obj} → (R : Mapping A B) → isDifunctional (Mapping.mor R)
  mappingDifunctional R = proj₂ (mappingUnivalent R)
  mappingCodifunctional : {A B : Obj} → (R : Mapping A B) → isCodifunctional (Mapping.mor R)
  mappingCodifunctional R = proj₁ (mappingTotal R) ⟨⊑≈⟩ ⨾-assoc
  mappingBiDifunctional : {A B : Obj} → (R : Mapping A B) → isBiDifunctional (Mapping.mor R)
  mappingBiDifunctional R = ⊑-antisym (mappingDifunctional R) (mappingCodifunctional R)
```

11.9 Categoric.OSGC.Props.Difunctional

```
module OSGC-DifunctionalProps {i j k₁ k₂ : Level} {Obj : Set i} (Base : OSGC-Base j k₁ k₂ Obj) where
  open OSGC-Base Base
  open OSGC-Props Base
  ˘-isDifunctional : {A B : Obj} → {R : Mor A B}
                   → isDifunctional R → isDifunctional (R ˘)
  ˘-isDifunctional {A} {B} {R} difunR = ⊑-begin
      R ˘ ⨾ (R ˘) ˘ ⨾ R ˘
    ≈⟨ ⨾-cong₂ (≈-sym ˘-involution) ⟩
```

```
      R ˘ ⨾ (R ⨾ R ˘) ˘
    ≈⟨ ≈-sym ˘-involution ⟩
      ((R ⨾ R ˘) ⨾ R) ˘
    ≈⟨ ˘-cong ⨾-assoc ⟩
      (R ⨾ R ˘ ⨾ R) ˘
    ⊑⟨ ˘-monotone difunR ⟩
      R ˘ □
  univalent-isDifunctional : {A B : Obj} {R : Mor A B}
                            → isUnivalent R → isDifunctional R
  univalent-isDifunctional (_, right) = right
  injective-isDifunctional  : {A B : Obj} {R : Mor A B}
                            → isInjective R → isDifunctional R
  injective-isDifunctional {A} {B} {R} (left, _) = ⊑-begin
      R ⨾ R ˘ ⨾ R
    ≈⟨ ⨾-assocL ⟩
      (R ⨾ R ˘) ⨾ R
    ⊑⟨ left ⟩
      R
    □


  infix 4 _⊑⊞_
  record _⊑⊞_ {A B : Obj} (R S : Mor A B) : Set k₂ where
      field
        ⊑⊞⊑ : R ⊑ S
        ⊑⊞-difun : isDifunctional S
      ⊑⊞-stepL : R ⨾ R ˘ ⨾ S ⊑ S
      ⊑⊞-stepL = ⊑-begin
              R ⨾ R ˘ ⨾ S
          ⊑⟨ ⨾-monotone ⊑⊞⊑ (⨾-monotone₁ (˘-monotone ⊑⊞⊑)) ⟩
              S ⨾ S ˘ ⨾ S
          ⊑⟨ ⊑⊞-difun ⟩
              S
          □
      ⊑⊞-stepR : S ⨾ R ˘ ⨾ R ⊑ S
      ⊑⊞-stepR = ⊑-begin
              S ⨾ R ˘ ⨾ R
          ⊑⟨ ⨾-monotone₂ (⨾-monotone (˘-monotone ⊑⊞⊑) ⊑⊞⊑) ⟩
              S ⨾ S ˘ ⨾ S
          ⊑⟨ ⊑⊞-difun ⟩
              S
          □


  record _isDifunClosOf_ {A B : Obj} (S R : Mor A B) : Set (j ⊍ k₂) where
    field
        difunClos-incl    : R ⊑ S
        difunClos-difun   : isDifunctional S
        difunClos-minimal : {T : Mor A B} → R ⊑⊞ T → S ⊑ T
    difunClos-⊑⊞ : R ⊑⊞ S
    difunClos-⊑⊞ = record {⊑⊞⊑ = difunClos-incl; ⊑⊞-difun = difunClos-difun}
    open _⊑⊞_ difunClos-⊑⊞ public using () renaming
        (⊑⊞-stepL to difunClos-stepL
        ; ⊑⊞-stepR to difunClos-stepR
        )


  _leftIndBoxStarOf_ : {A B : Obj} → Mor A B → Mor A B → Set (i ⊍ j ⊍ k₂)
  _leftIndBoxStarOf_ {A} {B} S R = ∀ {C : Obj} {P : Mor C A} {Q : Mor C B}
```

```
                                        → P ⨾ R ⊑ Q → Q ⨾ R ˘ ⨾ R ⊑ Q → P ⨾ S ⊑ Q
  _rightIndBoxStarOf_ :     {A B : Obj} → Mor A B → Mor A B → Set (i ⊍ j ⊍ k₂)
  _rightIndBoxStarOf_ {A} {B} S R = ∀ {C : Obj} {P : Mor B C} {Q : Mor A C}
                                        → R ⨾ P ⊑ Q → R ⨾ R ˘ ⨾ Q ⊑ Q → S ⨾ P ⊑ Q

  record _isBoxStarOf_ {A B : Obj} (S : Mor A B) (R : Mor A B) : Set (i ⊍ j ⊍ k₂) where
    field
        isBoxStar-incl      : R ⊑ S
        isBoxStar-difun     : isDifunctional S
        isBoxStar-leftInd   : S leftIndBoxStarOf R
        isBoxStar-rightInd  : S rightIndBoxStarOf R
    isBoxStar-⊑⊞ : R ⊑⊞ S
    isBoxStar-⊑⊞ = record {⊑⊞⊑ = isBoxStar-incl; ⊑⊞-difun = isBoxStar-difun}
    open _⊑⊞_ isBoxStar-⊑⊞ public using () renaming
        (⊑⊞-stepL to isBoxStar-stepL
        ; ⊑⊞-stepR to isBoxStar-stepR
        )


    isDifunClosOf-from-isBoxStarOf : (v : Mor A A) → isSubidentity v → S ⊑ v ⨾ S
                                    → S isDifunClosOf R
    isDifunClosOf-from-isBoxStarOf v vSubid S⊑vS
      = record
          {difunClos-incl     = isBoxStar-incl
          ; difunClos-difun    = isBoxStar-difun
          ; difunClos-minimal  = λ {T} R⊑⊞T →   -- S ⊑ T
            let open _⊑⊞_ R⊑⊞T
              vS⊑T = isBoxStar-leftInd {A} {P = v} {Q = T}
                      (   -- v ⨾ R ⊑ T
                        ⊑-begin    v ⨾ R ⊑⟨ proj₁ vSubid ⟩      R ⊑⟨ ⊑⊞⊑ ⟩   T □)
                      ⊑⊞-stepR   -- T ⨾ R ˘ ⨾ R ⊑ T
            in S⊑vS ⟨⊑⊑⟩ vS⊑T
          }
```

Auxiliary properties:

```
  isBoxStar-rightSubId : {A B : Obj} → {R S : Mor A B}
    →                    {rr : Mor B B} → R ⨾ rr ⊑ R
    →                    S isBoxStarOf R → S ⨾ rr ⊑ S
  isBoxStar-rightSubId {_} {_} {R} {S} {rr} R⨾rr⊑R Sibs =
    let open _isBoxStarOf_ Sibs
    in isBoxStar-rightInd
      (⊑-begin
              R ⨾ rr
            ⊑⟨ R⨾rr⊑R ⟩
              R
            ⊑⟨ isBoxStar-incl ⟩
              S
        □)
      (⊑-begin
            R ⨾ R ˘ ⨾ S
            ⊑⟨ isBoxStar-stepL ⟩
              S
        □)
```

If the target of R has a right-identity, then R has a unique box-star.

```
  isBoxStar-unique⊑ : {A B : Obj} → {R S T : Mor A B}
    →                 {rr : Mor B B} → isRightIdentity rr
    →                 S isBoxStarOf R → T isBoxStarOf R → S ⊑ T
```

```
  isBoxStar-unique⊑ {_} {_} {R} {S} {T} {rr} rightId Sibs Tibs = ≈-sym rightId ⟨≈⊑⟩
    let open _isBoxStarOf_ in
    isBoxStar-rightInd Sibs
      (⊑-begin
          R ⨾ rr
        ≈⟨ rightId ⟩
          R
        ⊑⟨ isBoxStar-incl Tibs ⟩
          T
        □)
      (⊑-begin
          R ⨾ R ˘ ⨾ T
        ⊑⟨ isBoxStar-stepL Tibs ⟩
          T
        □)
  isBoxStar-unique : {A B : Obj} → {R S T : Mor A B}
    →                   {rr : Mor B B} → isRightIdentity rr
    →                   S isBoxStarOf R → T isBoxStarOf R → S ≈ T
  isBoxStar-unique rightId Sibs Tibs = ⊑-antisym (isBoxStar-unique⊑ rightId Sibs Tibs)
                                                  (isBoxStar-unique⊑ rightId Tibs Sibs)
```

11.10 Categoric.OCC

```
record OCC {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field OCC_Base : OCC-Base j k₁ k₂ Obj
  open OCC-Base OCC_Base                public
  open OCC-Props OCC_Base               public
  open OCC-Prop-Conversions OCC_Base public
  open OCC-CompProps OCC_Base           public
  open OCC-IdProps OCC_Base             public
  open OCC-MappingProps OCC_Base        public
```

```
retractOCC : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
           → (F : Obj₂ → Obj₁) → OCC j k₁ k₂ Obj₁ → OCC j k₁ k₂ Obj₂
retractOCC F base = let open OCC base in record {OCC_Base = retractOCC-Base F OCC_Base}
```

```
attachOCC : {i j k₁ k₂ : Level} {Obj : Set i}
          → OCC j k₁ k₂ Obj → OCC (i ⊍ j) k₁ k₂ Obj
attachOCC base = let open OCC base in record {OCC_Base = attachOCC-Base OCC_Base}
```

11.11 Categoric.OCC.Base

```
record OCC-Base {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field osgc : OSGC j k₁ k₂ Obj
  open OSGC osgc
  field idOp : IdOp Hom≈ _⨾_
  orderedCategory : OrderedCategory j k₁ k₂ Obj
  orderedCategory = record
    {orderedSemigroupoid = orderedSemigroupoid
    ; idOp = idOp
    }
  convCategory : ConvCategory j k₁ Obj
```

```
  convCategory = record
    {convSemigroupoid = convSemigroupoid
    ; idOp = idOp
    }
  open OrderedCategory.OC-Props orderedCategory   public
  open ConvCategory             convCategory      public using (Id˘; category)
  open CategoryProps            semigroupoid idOp public
  open OSGC                     osgc              public
  open IdOp                     idOp              public
```

```
retractOCC-Base : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                → (F : Obj₂ → Obj₁)
                → OCC-Base j k₁ k₂ Obj₁ → OCC-Base j k₁ k₂ Obj₂
retractOCC-Base F base = let open OCC-Base base in record
  {osgc = retractOSGC F osgc
  ; idOp = retractIdOp F idOp
  }
```

```
attachOCC-Base : {i j k₁ k₂ : Level} {Obj : Set i}
               → OCC-Base j k₁ k₂ Obj → OCC-Base (i ⊍ j) k₁ k₂ Obj
attachOCC-Base base = let open OCC-Base base in record
  {osgc = attachOSGC osgc
  ; idOp = attachIdOp idOp
  }
```

11.12 Categoric.OCC.Props

```
module OCC-Props {i j k₁ k₂ : Level} {Obj : Set i} (Base : OCC-Base j k₁ k₂ Obj) where
  open OCC-Base Base
  isUnivalentI : {A B : Obj} → Mor A B → Set k₂
  isUnivalentI R = isCoreflexive (R ˘ ⨾ R)
  isInjectiveI : {A B : Obj} → Mor A B → Set k₂
  isInjectiveI R = isCoreflexive (R ⨾ R ˘)
  isTotalI : {A B : Obj} → Mor A B → Set k₂
  isTotalI R = isReflexive (R ⨾ R ˘)
  isSurjectiveI : {A B : Obj} → Mor A B → Set k₂
  isSurjectiveI R = isReflexive (R ˘ ⨾ R)
  isMappingI : {A B : Obj} → Mor A B → Set k₂
  isMappingI R = isUnivalentI R × isTotalI R
  isBijectiveI : {A B : Obj} → Mor A B → Set k₂
  isBijectiveI R = isInjectiveI R × isSurjectiveI R
```

According to (Freyd and Scedrov, 1990, 2.15), an object U in an allegory is a *partial unit* if Id {U} is a top morphism. The object U is a *unit* if, further, every object is the source of a total morphism targeted at U.

```
  record IsUnit (U : Obj) : Set (i ⊍ j ⊍ k₂) where
    field
      Id-isTop        : isTop (Id {U})
      toUnit          : {A : Obj} → Mor A U
      toUnit-isTotalI : {A : Obj} → isTotalI (toUnit {A})
    toUnit-isTotal : {A : Obj} → isTotal (toUnit {A})
    toUnit-isTotal {A} = reflexiveIsSuperidentity toUnit-isTotalI
```

11.13 Categoric.OCC.Props.Conversions

```
module OCC-Prop-Conversions {i j k₁ k₂ : Level} {Obj : Set i} (Base : OCC-Base j k₁ k₂ Obj) where
  open OCC-Base Base
  open OCC-Props Base
  ≈-isUnivalentI : {A B : Obj} → {P Q : Mor A B} → Q ≈ P → isUnivalentI P → isUnivalentI Q
  ≈-isUnivalentI qp = ≈-isCoreflexive (⨾-cong (˘-cong qp) qp)
  ≈-isInjectiveI :  {A B : Obj} → {P Q : Mor A B} → Q ≈ P → isInjectiveI P → isInjectiveI Q
  ≈-isInjectiveI qp   = ≈-isCoreflexive (⨾-cong qp (˘-cong qp))
  ≈-isTotalI : {A B : Obj} → {P Q : Mor A B} → Q ≈ P → isTotalI P → isTotalI Q
  ≈-isTotalI qp = ≈-isReflexive (⨾-cong qp (˘-cong qp))
  ≈-isSurjectiveI :  {A B : Obj} → {P Q : Mor A B} → Q ≈ P → isSurjectiveI P → isSurjectiveI Q
  ≈-isSurjectiveI qp = ≈-isReflexive (⨾-cong (˘-cong qp) qp)
  ≈-isMappingI : {A B : Obj} → {P Q : Mor A B} → Q ≈ P → isMappingI P → isMappingI Q
  ≈-isMappingI qp (u, t) = ≈-isUnivalentI qp u, ≈-isTotalI qp t
  ≈-isBijectiveI :  {A B : Obj} → {P Q : Mor A B} → Q ≈ P → isBijectiveI P → isBijectiveI Q
  ≈-isBijectiveI qp (i, s) = ≈-isInjectiveI qp i, ≈-isSurjectiveI qp s
```

isUnivalent — isInjective conversions:

```
  isInjectiveFromUnivalentI : {A B : Obj} → {R : Mor A B} → isUnivalentI (R ˘) → isInjectiveI R
  isInjectiveFromUnivalentI {A} {B} {R} univalRC = ⊑-begin
      R ⨾ R ˘
    ≈⟨ ⨾-cong₁ (≈-sym ˘˘) ⟩
      (R ˘) ˘ ⨾ R ˘
    ⊑⟨ univalRC ⟩
      Id
    □
  isInjectiveToUnivalentI : {A B : Obj} → {R : Mor A B} → isInjectiveI R → isUnivalentI (R ˘)
  isInjectiveToUnivalentI {A} {B} {R} injR = ⊑-begin
      (R ˘) ˘ ⨾ R ˘
    ≈⟨ ⨾-cong₁ ˘˘ ⟩
      R ⨾ R ˘
    ⊑⟨ injR ⟩
      Id
    □
  isUnivalentFromInjectiveI : {A B : Obj} → {R : Mor A B} → isInjectiveI (R ˘) → isUnivalentI R
  isUnivalentFromInjectiveI {A} {B} {R} injRC = ⊑-begin
      R ˘ ⨾ R
    ≈⟨ ⨾-cong₂ (≈-sym ˘˘) ⟩
      R ˘ ⨾ (R ˘) ˘
    ⊑⟨ injRC  ⟩
      Id
    □
  isUnivalentToInjectiveI : {A B : Obj} → {R : Mor A B} → isUnivalentI R → isInjectiveI (R ˘)
  isUnivalentToInjectiveI {A} {B} {R} univalR = ⊑-begin
      R ˘ ⨾ (R ˘) ˘
    ≈⟨ ⨾-cong₂ ˘˘ ⟩
      R ˘ ⨾ R
    ⊑⟨ univalR ⟩
      Id
    □
```

isSurjective — isTotal conversions:

```
  isSurjectiveFromTotalI : {A B : Obj} → {R : Mor A B} → isTotalI (R ˘) → isSurjectiveI R
  isSurjectiveFromTotalI {A} {B} {R} totalRC = ⊑-begin
```

```
    Id
  ⊑⟨ totalRC ⟩
    R ˘ ⨾ (R ˘) ˘
  ≈⟨ ⨾-cong₂ ˘˘ ⟩
    R ˘ ⨾ R
  □

isSurjectiveToTotalI : {A B : Obj} → {R : Mor A B} → isSurjectiveI R → isTotalI (R ˘)
isSurjectiveToTotalI {A} {B} {R} surjR = ⊑-begin
    Id
  ⊑⟨ surjR ⟩
    R ˘ ⨾ R
  ≈⟨ ⨾-cong₂ (≈-sym ˘˘) ⟩
    R ˘ ⨾ (R ˘) ˘
  □

isTotalFromSurjectiveI : {A B : Obj} → {R : Mor A B} → isSurjectiveI (R ˘) → isTotalI R
isTotalFromSurjectiveI {A} {B} {R} surjRC = ⊑-begin
    Id
  ⊑⟨ surjRC  ⟩
    (R ˘) ˘ ⨾ R ˘
  ≈⟨ ⨾-cong₁ ˘˘ ⟩
    R ⨾ R ˘
  □

isTotalToSurjectiveI : {A B : Obj} → {R : Mor A B} → isTotalI R → isSurjectiveI (R ˘)
isTotalToSurjectiveI {A} {B} {R} totalR = ⊑-begin
    Id
  ⊑⟨ totalR ⟩
    R ⨾ R ˘
  ≈⟨ ⨾-cong₁ (≈-sym ˘˘) ⟩
    (R ˘) ˘ ⨾ R ˘
  □
```

Conversions to and from the more widely used OSGC properties:

```
isUnivalent-from-I : {A B : Obj} → {R : Mor A B} → isUnivalentI R → isUnivalent R
isUnivalent-from-I univalentR = coreflexiveIsSubidentity univalentR

isInjective-from-I : {A B : Obj} → {R : Mor A B} → isInjectiveI R → isInjective R
isInjective-from-I injectiveR = coreflexiveIsSubidentity injectiveR

isTotal-from-I : {A B : Obj} → {R : Mor A B} → isTotalI R → isTotal R
isTotal-from-I totalR = reflexiveIsSuperidentity totalR

isSurjective-from-I : {A B : Obj} → {R : Mor A B} → isSurjectiveI R → isSurjective R
isSurjective-from-I surjectiveR = reflexiveIsSuperidentity surjectiveR

isMapping-from-I : {A B : Obj} → {R : Mor A B} → isMappingI R → isMapping R
isMapping-from-I (u, t) = isUnivalent-from-I u, isTotal-from-I t

isBijective-from-I : {A B : Obj} → {R : Mor A B} → isBijectiveI R → isBijective R
isBijective-from-I (i, s) = isInjective-from-I i, isSurjective-from-I s


isUnivalent-to-I : {A B : Obj} → {R : Mor A B} → isUnivalent R → isUnivalentI R
isUnivalent-to-I univalentR = subidentityIsCoreflexive univalentR

isInjective-to-I : {A B : Obj} → {R : Mor A B} → isInjective R → isInjectiveI R
isInjective-to-I injectiveR = subidentityIsCoreflexive injectiveR

isTotal-to-I : {A B : Obj} → {R : Mor A B} → isTotal R → isTotalI R
isTotal-to-I totalR = superidentityIsReflexive totalR

isSurjective-to-I : {A B : Obj} → {R : Mor A B} → isSurjective R → isSurjectiveI R
isSurjective-to-I surjectiveR = superidentityIsReflexive surjectiveR

isMapping-to-I : {A B : Obj} → {R : Mor A B} → isMapping R → isMappingI R
isMapping-to-I (u, t) = isUnivalent-to-I u, isTotal-to-I t
```

```
  isBijective-to-I : {A B : Obj} → {R : Mor A B} → isBijective R → isBijectiveI R
  isBijective-to-I (i, s) = isInjective-to-I i, isSurjective-to-I s
```

11.14 Categoric.OCC.Props.Id

```
module OCC-IdProps {i j k₁ k₂ : Level} {Obj : Set i} (Base : OCC-Base j k₁ k₂ Obj) where
  open OCC-Base Base
  open OCC-Props Base
  open OCC-Prop-Conversions Base
  idCoreflexive : {A : Obj} → isCoreflexive (Id {A})
  idCoreflexive {A} = Poset.refl (Hom A A)
  idReflexive : {A : Obj} → isReflexive (Id {A})
  idReflexive {A} = Poset.refl (Hom A A)
  idUnivalentI : {A : Obj} → isUnivalentI (Id {A})
  idUnivalentI {A} = ⊑-begin
      Id ˘ ⨾ Id
    ≈⟨ rightId ⟩
      Id ˘
    ≈⟨ Id˘ ⟩
      Id
    □
  idUnivalent : {A : Obj} → isUnivalent (Id {A})
  idUnivalent = coreflexiveIsSubidentity idUnivalentI
  idInjectiveI : {A : Obj} → isInjectiveI (Id {A})
  idInjectiveI = ≈-isInjectiveI (≈-sym Id˘) (isUnivalentToInjectiveI idUnivalentI)
  idInjective : {A : Obj} → isInjective (Id {A})
  idInjective = coreflexiveIsSubidentity idInjectiveI
  idTotalI : {A : Obj} → isTotalI (Id {A})
  idTotalI {A} = ⊑-begin
      Id
    ≈⟨ ≈-sym Id˘ ⟩
      Id ˘
    ≈⟨ ≈-sym leftId ⟩
      Id ⨾ Id ˘
    □
  idTotal : {A : Obj} → isTotal (Id {A})
  idTotal = reflexiveIsSuperidentity idTotalI
  idSurjectiveI : {A : Obj} → isSurjectiveI (Id {A})
  idSurjectiveI = ≈-isSurjectiveI (≈-sym Id˘) (isTotalToSurjectiveI idTotalI)
  idSurjective : {A : Obj} → isSurjective (Id {A})
  idSurjective = reflexiveIsSuperidentity idSurjectiveI
  idMappingI : {A : Obj} → isMappingI (Id {A})
  idMappingI = idUnivalentI, idTotalI
  idMapping : {A : Obj} → isMapping (Id {A})
  idMapping = idUnivalent, idTotal
  MappingId : {A : Obj} → Mapping A A
  MappingId = record {mor = Id; prf = idMapping}
```

11.15 Categoric.OCC.Props.Comp

```
module OCC-CompProps {i j k₁ k₂ : Level} {Obj : Set i} (Base : OCC-Base j k₁ k₂ Obj) where
```

```
  open OCC-Base Base
  open OCC-Props Base
  open OCC-Prop-Conversions Base
  ⨾-isUnivalentI : {A B C : Obj} → {R : Mor A B} → {S : Mor B C}
                 → isUnivalentI R → isUnivalentI S → isUnivalentI (R ⨾ S)
  ⨾-isUnivalentI {A} {B} {C} {R} {S} univalR univalS = ⊑-begin
        (R ⨾ S) ˘ ⨾ (R ⨾ S)
      ≈⟨ ⨾-cong₁ ˘-involution ⟩
        (S ˘ ⨾ R ˘) ⨾ R ⨾ S
      ≈⟨ ⨾-assoc ⟩
        S ˘ ⨾ (R ˘ ⨾ R ⨾ S)
      ≈⟨ ⨾-cong₂ ⨾-assocL ⟩
        S ˘ ⨾ (R ˘ ⨾ R) ⨾ S
      ⊑⟨ ⨾-monotone₂₁ univalR ⟩
        S ˘ ⨾ Id ⨾ S
      ≈⟨ ⨾-cong₂ leftId ⟩
        S ˘ ⨾ S
      ⊑⟨ univalS ⟩
        Id
      □
  ⨾-isInjectiveI  : {A B C : Obj} → {R : Mor A B} → {S : Mor B C}
                 → isInjectiveI R → isInjectiveI S → isInjectiveI (R ⨾ S)
  ⨾-isInjectiveI injR injS = isInjectiveFromUnivalentI
    (≈-isUnivalentI ˘-involution
      (⨾-isUnivalentI (isInjectiveToUnivalentI injS) (isInjectiveToUnivalentI injR)))
  ⨾-isTotalI : {A B C : Obj} → {R : Mor A B} → {S : Mor B C}
    →         isTotalI R → isTotalI S → isTotalI (R ⨾ S)
  ⨾-isTotalI {A} {B} {C} {R} {S} totalR totalS = ⊑-begin
        Id
      ⊑⟨ totalR ⟩
        R ⨾ R ˘
      ≈⟨ ⨾-cong₂ (≈-sym leftId) ⟩
        R ⨾ Id ⨾ R ˘
      ⊑⟨ ⨾-monotone₂₁ totalS ⟩
        R ⨾ (S ⨾ S ˘) ⨾ R ˘
      ≈⟨ ⨾-cong₂ ⨾-assoc ⟩
        R ⨾ S ⨾ (S ˘ ⨾ R ˘)
      ≈⟨ ⨾-assocL ⟩
        (R ⨾ S) ⨾ (S ˘ ⨾ R ˘)
      ≈⟨ ⨾-cong₂ (≈-sym ˘-involution) ⟩
        (R ⨾ S) ⨾ (R ⨾ S) ˘
      □
  ⨾-isSurjectiveI  : {A B C : Obj} → {R : Mor A B} → {S : Mor B C}
                  → isSurjectiveI R → isSurjectiveI S → isSurjectiveI (R ⨾ S)
  ⨾-isSurjectiveI surjR surjS = isSurjectiveFromTotalI
    (≈-isTotalI ˘-involution (⨾-isTotalI (isSurjectiveToTotalI surjS) (isSurjectiveToTotalI surjR)))
  ⨾isMappingI : {A B C : Obj} → {R : Mor A B} → {S : Mor B C}
              → isMappingI R → isMappingI S → isMappingI (R ⨾ S)
  ⨾isMappingI (univalR, totalR) (univalS, totalS) =
    ⨾-isUnivalentI univalR univalS, ⨾-isTotalI totalR totalS
```

11.16 Categoric.OCC.Props.Mapping

```
module OCC-MappingProps {i j k₁ k₂ : Level} {Obj : Set i} (Base : OCC-Base j k₁ k₂ Obj) where
  open OCC-Base Base
```

```
open OCC-Props Base
open OCC-Prop-Conversions Base using (isUnivalent-from-I; isTotal-from-I)
mappingUnivalentI : {A B : Obj} → (R : Mapping A B) → isUnivalentI (Mapping.mor R)
mappingUnivalentI R = subidentityIsCoreflexive (mappingUnivalent R)
mappingTotalI : {A B : Obj} → (R : Mapping A B) → isTotalI (Mapping.mor R)
mappingTotalI R = superidentityIsReflexive (mappingTotal R)
mkMappingI : {A B : Obj} (R : Mor A B) → isMappingI R → Mapping A B
mkMappingI R (unival, total) = record
  {mor = R
  ; prf = isUnivalent-from-I unival, isTotal-from-I total
  }
```

11.17 Categoric.MapSG

```
MapSG : {i j k₁ k₂ : Level} {Obj : Set i}
      → OSGC j k₁ k₂ Obj → Semigroupoid (i ⊍ j ⊍ k₂) k₁ Obj
MapSG Base = let open OSGC Base in record
  {Hom = retract²LES Hom≈ {Mapping} id Mapping.mor   -- retractHom Hom≈ Mapping Mapping.mor
  ; compOp = record
       {_⨾_ = λ F G → record
         {mor = Mapping.mor F ⨾ Mapping.mor G
         ; prf = ⨾-isMapping (Mapping.prf F) (Mapping.prf G)
         }
       ; ⨾-assoc = ⨾-assoc
       ; ⨾-cong = ⨾-cong
       }
  }
```

11.18 Categoric.MapCat

```
MapCat : {i j k₁ k₂ : Level} {Obj : Set i}
       → OCC j k₁ k₂ Obj → Category (i ⊍ j ⊍ k₂) k₁ Obj
MapCat Base = let open OCC Base in record
  {semigroupoid = MapSG osgc
  ; idOp = record
      {Id = λ {A} → record {mor = Id {A}; prf = idMapping}
      ; leftId = leftId
      ; rightId = rightId
      }
  }
```

Chapter 12

Allegories, Collagories

The essential connection between meet and converse that enables much typically relation-algebraic reasoning is produced byt the *Dedekind rule*

$$(\mathsf{Q} \mathbin{;} \mathsf{R} \sqcap \mathsf{S}) \sqsubseteq (\mathsf{Q} \sqcap \mathsf{S} \mathbin{;} \mathsf{R}\,\breve{}\,) \mathbin{;} (\mathsf{R} \sqcap \mathsf{Q}\,\breve{}\, \mathbin{;} \mathsf{S})$$

or, equivalently, one of the *modal rules*

$$(\mathsf{Q} \mathbin{;} \mathsf{R} \sqcap \mathsf{S}) \sqsubseteq \mathsf{Q} \mathbin{;} (\mathsf{R} \sqcap \mathsf{Q}\,\breve{}\, \mathbin{;} \mathsf{S}) \qquad \text{and} \qquad (\mathsf{Q} \mathbin{;} \mathsf{R} \sqcap \mathsf{S}) \sqsubseteq (\mathsf{Q} \sqcap \mathsf{S} \mathbin{;} \mathsf{R}\,\breve{}\,) \mathbin{;} \mathsf{R}.$$

The minimal context for this connection has long been that of *allegories* as introduced by Freyd and Scedrov (1990), which are categories extended with converse, meet, and a modal rule. In that definition of allegories, the ordering is defined from the meet, and domain is defined as follows:

$$\mathsf{dom}\ \mathsf{R} \ = \ \mathsf{Id} \sqcap \mathsf{R} \mathbin{;} \mathsf{R}\,\breve{}$$

Dougherty and Gutiérrez (2000) presented graphical calculi for allegory reasoning, and proposed to consider domain as a primitive operation, even though it can be derived in allegories, because of the rôle it plays in the graph representation of allegory expressions.

Domain cannot be derived in lower semi-lattice semigroupoids with converse and the Dedekind rules, so in order to define *semi-allegories* as “allegories without identities”, Kahl (2006; 2008) also added a domain operator as primitive.

We formalise semi-allegories in Sect. 12.2, prove the domain properties for the derived definition of domain in allegories in Sect. 12.3, and integrate this into allegories in Sect. 12.4.

Sections 12.5 and 12.6 present upper semi-lattice semigroupoids and categories with converse, where we automatically obtain distributivity of converse over joins.

Distributive allegories (Sect. 12.13) were introduced by Freyd and Scedrov (1990) as allegories with binary joins and least morphism in each homset, with joins distributing over composition from both sides, and zero laws valid for the least morphisms. We introduce theory building blocks providing least morphisms and zero laws in Sect. 9.9.

Collagories were introduced by Kahl (2009; 2011a) essentially as “distributive allegories without zero morphisms”, motivated by the fact that previous formalisations of th relation-algebraic approach to graph transformation in the context of complete distributive allegories never used the zero laws. Here we first introduce semi-collagories in Sect. 12.10, and then extend them to collagories in Sect. 12.11.

12.1 Categoric.OSGC.LeastMor

```
module Categoric.OSGC.LeastMor {i j k₁ k₂ : Level} {Obj : Set i} (base : OSGC j k₁ k₂ Obj)
  where
open OSGC base
open LeastMor orderedSemigroupoid

⊥-˘ : {A B : Obj} {⊥ : Mor A B} (is-⊥ : isLeastMor ⊥) → isLeastMor (⊥ ˘)
⊥-˘ {A} {B} {⊥} is-⊥ R = ⊑-begin
```

```
      ⊥ ˘
   ⊑⟨ ˘-monotone (is-⊥ (R ˘)) ⟩
      (R ˘) ˘
   ≈⟨ ˘˘ ⟩
      R
   □


⊥-⨾-unival : {A B : Obj} {⊥ : Mor A B} (is-⊥ : isLeastMor ⊥) {C : Obj} {F : Mor B C}
           → isUnivalent F → isLeastMor (⊥ ⨾ F)
⊥-⨾-unival {A} {B} {⊥} is-⊥ {C} {F} F-unival R = ⊑-begin
      ⊥ ⨾ F
   ⊑⟨ ⨾-monotone₁ (is-⊥ (R ⨾ F ˘)) ⟨⊑≈⟩ ⨾-assoc ⟩
      R ⨾ F ˘ ⨾ F
   ⊑⟨ proj₂ F-unival ⟩
      R
   □


module OSGC-BotMor (botMor : BotMor orderedSemigroupoid) where
   open BotMor botMor
   ⊥˘ : {A B : Obj} → (⊥ {A} {B}) ˘ ≈ ⊥
   ⊥˘ = leastMor-≈-⊥ (⊥-˘ is-⊥)
```

12.2 Categoric.SemiAllegory

```
module SemiAllegoryProps {i j k₁ k₂ : Level} {Obj : Set i}
   (osgc : OSGC j k₁ k₂ Obj)
   (meetOp : MeetOp (OSGC.orderedSemigroupoid osgc))
   (Dedekind : let open OSGC osgc; open MeetOp meetOp
               in {A B C : Obj} {Q : Mor A B} {R : Mor B C} {S : Mor A C}
               → (Q ⨾ R ⊓ S) ⊑ (Q ⊓ S ⨾ R ˘) ⨾ (R ⊓ Q ˘ ⨾ S))
   where
      open OSGC osgc
      open MeetOp meetOp
      Dedekind' : {A B C : Obj} {Q : Mor A B} {R : Mor B C} {S : Mor A C}
                → (S ⊓ Q ⨾ R) ⊑ (Q ⊓ S ⨾ R ˘) ⨾ (R ⊓ Q ˘ ⨾ S)
      Dedekind' = ⊑-trans₂ ⊓-commutative Dedekind
      Dedekind≈ : {A B C : Obj} {Q : Mor A B} {R : Mor B C} {S : Mor A C}
                → (Q ⨾ R ⊓ S) ≈ ((Q ⊓ S ⨾ R ˘) ⨾ (R ⊓ Q ˘ ⨾ S) ⊓ S)
      Dedekind≈ = ⊑-antisym (⊓-universal Dedekind ⊓-lower₂)
                             (⊓-monotone₁ (⨾-monotone ⊓-lower₁ ⊓-lower₁))
      Dedekind≈' : {A B C : Obj} {Q : Mor A B} {R : Mor B C} {S : Mor A C}
                 → (S ⊓ Q ⨾ R) ≈ ((Q ⊓ S ⨾ R ˘) ⨾ (R ⊓ Q ˘ ⨾ S) ⊓ S)
      Dedekind≈' = ≈-trans ⊓-commutative Dedekind≈
      modal₁ : {A B C : Obj} {Q : Mor A B} {R : Mor B C} {S : Mor A C}
             → (Q ⨾ R ⊓ S) ⊑ Q ⨾ (R ⊓ Q ˘ ⨾ S)
      modal₁ = ⊑-trans Dedekind (⨾-monotone₁ ⊓-lower₁)
      modal₁' : {A B C : Obj} {Q : Mor A B} {R : Mor B C} {S : Mor A C}
             → (S ⊓ Q ⨾ R) ⊑ Q ⨾ (R ⊓ Q ˘ ⨾ S)
      modal₁' = ⊑-trans₂ ⊓-commutative modal₁
      modal₂ : {A B C : Obj} {Q : Mor A B} {R : Mor B C} {S : Mor A C}
             → (Q ⨾ R ⊓ S) ⊑ (Q ⊓ S ⨾ R ˘) ⨾ R
      modal₂ = ⊑-trans Dedekind (⨾-monotone₂ ⊓-lower₁)
      modal₂' : {A B C : Obj} {Q : Mor A B} {R : Mor B C} {S : Mor A C}
```

```
          → (S ⊓ Q ⨾ R) ⊑ (Q ⊓ S ⨾ R ˘) ⨾ R
modal₂' = ⊑-trans₂ ⊓-commutative modal₂
modal≈₁ : {A B C : Obj} {Q : Mor A B} {R : Mor B C} {S : Mor A C}
          → (Q ⨾ R ⊓ S) ≈ (Q ⨾ (R ⊓ Q ˘ ⨾ S) ⊓ S)
modal≈₁ = ⊑-antisym (⊓-universal modal₁ ⊓-lower₂) (⊓-monotone₁ (⨾-monotone₂ ⊓-lower₁))
modal≈₁' : {A B C : Obj} {Q : Mor A B} {R : Mor B C} {S : Mor A C}
           → (S ⊓ Q ⨾ R) ≈ (Q ⨾ (R ⊓ Q ˘ ⨾ S) ⊓ S)
modal≈₁' = ≈-trans ⊓-commutative modal≈₁
modal≈₂ : {A B C : Obj} {Q : Mor A B} {R : Mor B C} {S : Mor A C}
          → (Q ⨾ R ⊓ S) ≈ ((Q ⊓ S ⨾ R ˘) ⨾ R ⊓ S)
modal≈₂ = ⊑-antisym (⊓-universal modal₂ ⊓-lower₂) (⊓-monotone₁ (⨾-monotone₁ ⊓-lower₁))
modal≈₂' : {A B C : Obj} {Q : Mor A B} {R : Mor B C} {S : Mor A C}
           → (S ⊓ Q ⨾ R) ≈ ((Q ⊓ S ⨾ R ˘) ⨾ R ⊓ S)
modal≈₂' = ≈-trans ⊓-commutative modal≈₂
codifunctional-with-leftSupId : {A B : Obj} {R : Mor A B}
      {v : Mor A A} → R ⊑ v ⨾ R → isCodifunctional R
codifunctional-with-leftSupId {A} {B} {R} {v} RvR = ⊑-begin
          R
     ≈⟨ ≈-sym ⊓-idempotent ⟩
          R ⊓ R
     ⊑⟨ ⊓-monotone₁ RvR ⟩
          v ⨾ R ⊓ R
     ⊑⟨ modal₂ ⟩
          (v ⊓ R ⨾ R ˘) ⨾ R
     ⊑⟨ ⨾-monotone₁ ⊓-lower₂ ⟩
          (R ⨾ R ˘) ⨾ R
     ≈⟨ ⨾-assoc ⟩
          R ⨾ R ˘ ⨾ R
     □
codifunctional-with-rightSupId : {A B : Obj} {R : Mor A B}
      {v : Mor B B} → R ⊑ R ⨾ v → isCodifunctional R
codifunctional-with-rightSupId {A} {B} {R} {v} RvR = ⊑-begin
          R
     ≈⟨ ≈-sym ⊓-idempotent ⟩
          R ⊓ R
     ⊑⟨ ⊓-monotone₁ RvR ⟩
          R ⨾ v ⊓ R
     ⊑⟨ modal₁ ⟩
          R ⨾ (v ⊓ R ˘ ⨾ R)
     ⊑⟨ ⨾-monotone₂ ⊓-lower₂ ⟩
          R ⨾ R ˘ ⨾ R
     □
⨾-⊓-distribL : {A B C : Obj} {R₁ R₂ : Mor A B} {S : Mor B C}
  → isInjective S → ((R₁ ⊓ R₂) ⨾ S) ≈ ((R₁ ⨾ S) ⊓ (R₂ ⨾ S))
⨾-⊓-distribL {A} {B} {C} {R₁} {R₂} {S} (_, right) = ⊑-antisym ⨾-⊓-subdistribL
  (⊑-begin
          (R₁ ⨾ S) ⊓ (R₂ ⨾ S)
     ⊑⟨ modal₂ ⟩
          (R₁ ⊓ (R₂ ⨾ S) ⨾ S ˘) ⨾ S
     ≈⟨ ⨾-cong₁ (⊓-cong₂ ⨾-assoc) ⟩
          (R₁ ⊓ R₂ ⨾ (S ⨾ S ˘)) ⨾ S
     ⊑⟨ ⨾-monotone₁ (⊓-monotone₂ right) ⟩
          (R₁ ⊓ R₂) ⨾ S
     □
  )
⨾-⊓-distribR : {A B C : Obj} {R : Mor A B} {S₁ S₂ : Mor B C}
                → isUnivalent R → (R ⨾ (S₁ ⊓ S₂)) ≈ ((R ⨾ S₁) ⊓ (R ⨾ S₂))
```

```
⨾-⊓-distribR {A} {B} {C} {R} {S₁} {S₂} (left, _) = ⊑-antisym ⨾-⊓-subdistribR
  (⊑-begin
        (R ⨾ S₁) ⊓ (R ⨾ S₂)
    ⊑⟨ modal₁ ⟩
        R ⨾ (S₁ ⊓ (R ˘ ⨾ (R ⨾ S₂)))
    ≈⟨ ⨾-cong₂ (⊓-cong₂ (≈-sym ⨾-assoc)) ⟩
        R ⨾ (S₁ ⊓ ((R ˘ ⨾ R) ⨾ S₂))
    ⊑⟨ ⨾-monotone₂ (⊓-monotone₂ left) ⟩
        R ⨾ (S₁ ⊓ S₂)
    □
  )
˘-⊓-subdistrib : {A B : Obj} {R S : Mor A B} → (R ⊓ S) ˘ ⊑ (R ˘ ⊓ S ˘)
˘-⊓-subdistrib {A} {B} {R} {S} = ⊓-universal (˘-monotone ⊓-lower₁) (˘-monotone ⊓-lower₂)
˘-⊓-distrib : {A B : Obj} {R S : Mor A B} → (R ⊓ S) ˘ ≈ (R ˘ ⊓ S ˘)
˘-⊓-distrib {A} {B} {R} {S} = ⊑-antisym ˘-⊓-subdistrib
  (⊑-˘-swap (⊑-trans₁ ˘-⊓-subdistrib (⊓-cong ˘˘ ˘˘)))
⊓-subid-⨾  :    {A B : Obj} {q : Mor A A} {R S : Mor A B}
               → isSubidentity q → (R ⊓ q ⨾ S) ≈ (q ⨾ (R ⊓ S))
⊓-subid-⨾ {A} {B} {q} {R} {S} qSubid = ⊑-antisym
  (⊑-begin
        R ⊓ q ⨾ S
    ⊑⟨ ⊑-trans₁ modal₁' (⨾-cong₂ ⊓-commutative) ⟩
        q ⨾ (q ˘ ⨾ R ⊓ S)
    ⊑⟨ ⨾-monotone₂ (⊓-monotone₁ (˘-isLeftSubidentity (proj₂ qSubid))) ⟩
        q ⨾ (R ⊓ S)
    □
  )
  (⊑-begin
        q ⨾ (R ⊓ S)
    ⊑⟨ ⨾-⊓-subdistribR ⟩
        q ⨾ R ⊓ q ⨾ S
    ⊑⟨ ⊓-monotone₁ (proj₁ qSubid) ⟩
        R ⊓ q ⨾ S
    □
  )
  -- can be expressed in OrderedSemigroupoid,
  -- but apparently needs SemiAllegory for proof.
rightSubidentity-superDownClosed : {A : Obj} {p : Mor A A} → isRightSubidentity p
  → {B : Obj} → {S : Mor A B} → S ⊑ p ⨾ S
  → {R : Mor A B} → R ⊑ S → R ⊑ p ⨾ R
rightSubidentity-superDownClosed {A} {p} pRightSubid {B} {S} SpS {R} rs =
  let R⊓S≈R : R ⊓ S ≈ R
      R⊓S≈R = ⊑-to-⊓₁ rs
  in ⊑-begin
      R
    ≈⟨ ≈-sym R⊓S≈R ⟩
      R ⊓ S
    ⊑⟨ ⊓-monotone₂ SpS ⟩
      R ⊓ p ⨾ S
    ⊑⟨ modal₁' ⟩
      p ⨾ (S ⊓ p ˘ ⨾ R)
    ⊑⟨ ⨾-monotone₂ (⊓-monotone₂ (˘-isLeftSubidentity pRightSubid)) ⟩
      p ⨾ (S ⊓ R)
    ≈⟨ ⨾-cong₂ ⊓-commutative ⟩
      p ⨾ (R ⊓ S)
    ≈⟨ ⨾-cong₂ R⊓S≈R ⟩
      p ⨾ R
  □
```

```
    record _and_tabulate_ {A B C : Obj}
              (P : Mor C A) (Q : Mor C B) (V : Mor A B) : Set (i ⊍ j ⊍ k₁ ⊍ k₂)
      where
        field
          tabProjMapping₁ : isMapping P
          tabProjMapping₂ : isMapping Q
          tabCommutes     : P ˘ ⨾ Q ≈ V
          tabJointInj     : isSubidentity (P ⨾ P ˘ ⊓ Q ⨾ Q ˘)
    lslSemigroupoid : LSLSemigroupoid j k₁ k₂ Obj
    lslSemigroupoid = record
      {orderedSemigroupoid = orderedSemigroupoid
      ;meetOp = meetOp
      }

record SemiAllegory {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field osgc     : OSGC j k₁ k₂ Obj
  open OSGC osgc
  field
    meetOp    : MeetOp orderedSemigroupoid
    domainOp  : OSGDomainOp orderedSemigroupoid
  open MeetOp meetOp
  open OSGDomainOp     domainOp
  field
    Dedekind  : {A B C : Obj} {Q : Mor A B} {R : Mor B C} {S : Mor A C}
              → (Q ⨾ R ⊓ S) ⊑ (Q ⊓ S ⨾ R ˘) ⨾ (R ⊓ Q ˘ ⨾ S)
  open OSGC              osgc                     public
  open MeetOp                 meetOp              public
  open SemiAllegoryProps osgc meetOp Dedekind     public
  open OSGDomainOp            domainOp            public
  open FromDomainOp      osgc domainOp            public
  open OSGRangeOp             rangeOp             public
```

12.3 Categoric.Allegory.Dom

The module AllegoryDom only exports osgDomainOp and domainOp; it has been split out of Categoric.Allegory for performance reasons.

```
module AllegoryDom {i j k₁ k₂ : Level} {Obj : Set i}
    (occ : OCC j k₁ k₂ Obj)
    (meetOp : MeetOp (OCC.orderedSemigroupoid occ))
    (Dedekind : let open OCC     occ        using (Mor; _⨾_; _˘; _⊑_)
                    open MeetOp meetOp      using (_⊓_)
                in {A B C : Obj} {Q : Mor A B} {R : Mor B C} {S : Mor A C}
                → (Q ⨾ R ⊓ S) ⊑ (Q ⊓ S ⨾ R ˘) ⨾ (R ⊓ Q ˘ ⨾ S)
    )
  where
  open OCC occ
  open MeetOp meetOp
  open SemiAllegoryProps osgc meetOp Dedekind
  private
    dom : {A B : Obj} → Mor A B → Mor A A
    dom R = Id ⊓ R ⨾ R ˘
    ˘-dom :  {A B : Obj} → {R : Mor A B} → (dom R) ˘ ≈ dom R
    ˘-dom {A} {B} {R} = ≈-begin
              (dom R) ˘
```

```
        ≈⟨ ˘-cong ≈-refl ⟩
            (Id ⊓ R ⨾ R ˘) ˘
        ≈⟨ ˘-⊓-distrib ⟩
            Id ˘ ⊓ (R ⨾ R ˘) ˘
        ≈⟨ ⊓-cong Id˘ ˘-involutionRightConv ⟩
            Id ⊓ R ⨾ R ˘
        ≈⟨ ≈-refl ⟩
            dom R
      □
  D1 : {A B : Obj} {R : Mor A B} → dom R ⨾ R ≈ R
  D1 {A} {B} {R} = ⊑-antisym
    (⊑-begin
            dom R ⨾ R
        ≈⟨ ⨾-cong₁ ≈-refl ⟩
            (Id ⊓ R ⨾ R ˘) ⨾ R
        ⊑⟨ ⨾-monotone₁ ⊓-lower₁ ⟩
            Id ⨾ R
        ≈⟨ leftId ⟩
            R
    □)
    (⊑-begin
            R
        ≈˘⟨ ⊓-idempotent ⟩
            R ⊓ R
        ≈˘⟨ ⊓-cong₁ leftId ⟩
            Id ⨾ R ⊓ R
        ⊑⟨ modal₂ ⟩
            (Id ⊓ R ⨾ R ˘) ⨾ R
        ≈⟨ ≈-refl ⟩
            dom R ⨾ R
    □)
  domSubIdentity : {A B : Obj} {R : Mor A B} → isSubidentity (dom R)
  domSubIdentity = coreflexiveIsSubidentity ⊓-lower₁
  dom-⨾-idempotent : {A B : Obj} {R : Mor A B} → dom R ⨾ dom R ≈ dom R
  dom-⨾-idempotent {A} {B} {R} = ≈-sym (≈-begin
            dom R
        ≈⟨ ≈-refl ⟩
            Id ⊓ R ⨾ R ˘
        ≈⟨ ⊓-cong₂ (⨾-cong₁ D1 ⟨≈˘≈⟩ ⨾-assoc) ⟩
            Id ⊓ dom R ⨾ R ⨾ R ˘
        ≈⟨ ⊓-subid-⨾ domSubIdentity ⟩
            dom R ⨾ (Id ⊓ R ⨾ R ˘)
        ≈⟨ ≈-refl ⟩
            dom R ⨾ dom R
    □)

osgDomainOp : OSGDomainOp orderedSemigroupoid
osgDomainOp = record
  {dom = dom
  ; domSubIdentity = domSubIdentity
  ; dom-⨾-idempotent = dom-⨾-idempotent
  ; domPreserves⊑   -- {A B : Obj} {Q R : Mor A B} → Q ⊑ R → Q ⊑ dom R ⨾ Q
      = λ Q⊑R → rightSubidentity-superDownClosed (proj₂ domSubIdentity) (⊑-reflexive′ D1) Q⊑R
  ; domLeastPreserver = λ {A} {B} {R} {d} dSubId d⨾d≈d R⊑d⨾R → ⊑-begin
            dom R
        ≈⟨ ≈-refl ⟩
            Id ⊓ R ⨾ R ˘
        ⊑⟨ ⊓-monotone₂ (⨾-monotone₁ R⊑d⨾R ⟨⊑≈⟩ ⨾-assoc) ⟩
```

```
            Id ⊓ d ⨾ R ⨾ R ˘
        ≈⟨ ⊓-subid-⨾ dSubId ⟩
            d ⨾ (Id ⊓ R ⨾ R ˘)
        ⊑⟨ ⨾-monotone₂ ⊓-lower₁ ⟩
            d ⨾ Id
        ≈⟨ rightId ⟩
            d
      □
   ; domLocality   -- ∀ {A B C : Obj} {R : Mor A B} {S : Mor B C} → dom (R ⨾ dom S) ⊑ dom (R ⨾ S)
      = λ {A} {B} {C} {R} {S} → ⊑-begin
            dom (R ⨾ dom S)
        ≈⟨ ⊓-cong₂ (⨾-cong₂ (˘-involution ⟨≈≈⟩ ⨾-cong₁ ˘-dom) ⟨≈≈⟩ ⨾-assoc) ⟩
            Id ⊓ R ⨾ dom S ⨾ dom S ⨾ R ˘
        ≈⟨ ⊓-cong₂ (⨾-cong₂ (⨾-assocL ⟨≈≈⟩ (⨾-cong₁ dom-⨾-idempotent))) ⟩
            Id ⊓ R ⨾ (Id ⊓ S ⨾ S ˘) ⨾ R ˘
        ⊑⟨ ⊓-monotone₂ (⨾-monotone₂ (⨾-monotone₁ ⊓-lower₂ ⟨⊑≈⟩ ⨾-assoc)) ⟩
            Id ⊓ R ⨾ S ⨾ S ˘ ⨾ R ˘
        ≈˘⟨ ⊓-cong₂ (⨾-cong₂₂ ˘-involution) ⟩
            Id ⊓ R ⨾ S ⨾ (R ⨾ S) ˘
        ≈⟨ ⊓-cong₂ ⨾-assocL ⟩
            dom (R ⨾ S)
      □
   }
open OSGDomainOp osgDomainOp hiding (dom)

domainOp : DomainOp semigroupoid
domainOp = record
   {dom = dom
   ; dom-cong = λ {A} {B} {R} {S} R≈S → ⊓-cong₂ (⨾-cong R≈S (˘-cong R≈S))   -- = dom-cong
   ; D1   -- : {A B : Obj} {R : Mor A B} → dom R ⨾ R ≈ R − = dom-D1
      = λ {A} {B} {R} → ⊑-antisym
     (⊑-begin
            dom R ⨾ R
        ≈⟨ ⨾-cong₁ ≈-refl ⟩
            (Id ⊓ R ⨾ R ˘) ⨾ R
        ⊑⟨ ⨾-monotone₁ ⊓-lower₁ ⟩
            Id ⨾ R
        ≈⟨ leftId ⟩
            R
     □)
     (⊑-begin
            R
        ≈˘⟨ ⊓-idempotent ⟩
            R ⊓ R
        ≈˘⟨ ⊓-cong₁ leftId ⟩
            Id ⨾ R ⊓ R
        ⊑⟨ modal₂ ⟩
            (Id ⊓ R ⨾ R ˘) ⨾ R
        ≈⟨ ≈-refl ⟩
            dom R ⨾ R
     □)
   ; D2 = domLocality≈        -- dom (R ⨾ dom S) ≈ dom (R ⨾ S)
   ; D3 = dom-⨾-dom-swap   -- dom (dom R ⨾ S) ≈ dom R ⨾ dom S
   ; D4 = dom-⨾-commute     -- dom R ⨾ dom S ≈ dom S ⨾ dom R
   }
```

12.4 Categoric.Allegory

```
record Allegory {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field  occ : OCC j k₁ k₂ Obj
  open OCC occ

  field meetOp : MeetOp orderedSemigroupoid
  open MeetOp meetOp

  field
    Dedekind : {A B C : Obj} {Q : Mor A B} {R : Mor B C} {S : Mor A C}
              → (Q ⨾ R ⊓ S) ⊑ (Q ⊓ S ⨾ R ˘) ⨾ (R ⊓ Q ˘ ⨾ S)

  open SemiAllegoryProps osgc meetOp Dedekind public
  open AllegoryDom occ meetOp Dedekind public
  open OSGDomainOp osgDomainOp public
  open FromDomainOp osgc osgDomainOp public
  open OSGRangeOp rangeOp public

  semiAllegory : SemiAllegory j k₁ k₂ Obj
  semiAllegory = record
    {osgc = osgc
    ; meetOp = meetOp
    ; domainOp = osgDomainOp
    ; Dedekind = Dedekind
    }


  codifunctional : {A B : Obj} {R : Mor A B} → isCodifunctional R
  codifunctional {A} {B} {R} = ⊑-begin
        R
      ≈⟨ ≈-sym ⊓-idempotent ⟩
        R ⊓ R
      ≈⟨ ⊓-cong₁ (≈-sym leftId) ⟩
        Id ⨾ R ⊓ R
      ⊑⟨ modal₂ ⟩
        (Id ⊓ R ⨾ R ˘) ⨾ R
      ⊑⟨ ⨾-monotone₁ ⊓-lower₂ ⟩
        (R ⨾ R ˘) ⨾ R
      ≈⟨ ⨾-assoc ⟩
        R ⨾ R ˘ ⨾ R
    □
```

Since range is obtained as the dual of dom in FromDomainOp above, its definition does not simplify to its direct definition, proved as a derived equality ran-def here:

```
  ran-def : {A B : Obj} {R : Mor A B} → ran R ≈ Id ⊓ R ˘ ⨾ R
  ran-def {R = R} = ≈-begin
        ran R
      ≈⟨ ≈-refl ⟩
        (dom (R ˘)) ˘
      ≈⟨ ≈-refl ⟩
        (Id ⊓ R ˘ ⨾ (R ˘) ˘) ˘
      ≈⟨ ˘-⊓-distrib ⟩
        Id ˘ ⊓ (R ˘ ⨾ (R ˘) ˘) ˘
      ≈⟨ ⊓-cong Id˘ ˘-coinvolution ⟩
        Id ⊓ R ˘ ⨾ R
      □


  univalent-ran : {A B : Obj} {R : Mor A B} → isUnivalent R → ran R ≈ R ˘ ⨾ R
  univalent-ran {A} {B} {R} unival-R = ≈-begin
```

```
    ran R
  ≈⟨ ran-def ⟩
    Id ⊓ R ˘ ⨾ R
  ≈⟨ ⊑-to-⊓₂ (isUnivalent-to-I unival-R) ⟩
    R ˘ ⨾ R
  □
```

In the original motivation for splitting-from-univalent we have $E_1 \sqsubseteq E$, but this is in fact not neccessary.

```
splitting-from-univalent : {A B Q : Obj} {E : Mor A A} {E₁ : Mor A B} {J : Mor Q B}
                         → isUnivalent E₁ → E₁ ⨾ E₁ ˘ ≈ E
                         → isMapping J → isInjective J → ran J ≈ ran E₁
                         → IsSymSplitting E (E₁ ⨾ J ˘)
splitting-from-univalent {A} {B} {Q} {E} {E₁} {J} unival-E₁ factors-E₁ (unival-J, total-J) inj-J ranJ≈ranE₁
  = record
    {factors = ≈-begin
          (E₁ ⨾ J ˘) ⨾ (E₁ ⨾ J ˘) ˘
      ≈⟨ ⨾-cong₂ ˘-involutionRightConv ⟨≈≈⟩ ⨾-assoc ⟩
          E₁ ⨾ J ˘ ⨾ J ⨾ E₁ ˘
      ≈⟨ ⨾-cong₂ (⨾-assocL ⟨≈≈˘⟩ ⨾-cong₁ (univalent-ran unival-J)) ⟩
          E₁ ⨾ ran J ⨾ E₁ ˘
      ≈⟨ ⨾-cong₂₁ ranJ≈ranE₁ ⟩
          E₁ ⨾ ran E₁ ⨾ E₁ ˘
      ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ ran-D1 ⟩
          E₁ ⨾ E₁ ˘
      ≈⟨ factors-E₁ ⟩
          E
      □
    ; splitId = isIdentity-subst (≈-sym (≈-begin
          (E₁ ⨾ J ˘) ˘ ⨾ (E₁ ⨾ J ˘)
      ≈⟨ ⨾-cong₁ ˘-involutionRightConv ⟨≈≈⟩ ⨾-assoc ⟩
          J ⨾ E₁ ˘ ⨾ E₁ ⨾ J ˘
      ≈⟨ ⨾-cong₂ (⨾-assocL ⟨≈≈˘⟩ ⨾-cong₁ (univalent-ran unival-E₁)) ⟩
          J ⨾ ran E₁ ⨾ J ˘
      ≈˘⟨ ⨾-cong₂₁ ranJ≈ranE₁ ⟩
          J ⨾ ran J ⨾ J ˘
      ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ ran-D1 ⟩
          J ⨾ J ˘
      □)) (⊑⊒-isIdentity inj-J total-J)
    }


splitting-from-univalentI : {A B Q : Obj} {E : Mor A A} {E₁ : Mor A B} {J : Mor Q B}
                          → isUnivalentI E₁ → E₁ ⨾ E₁ ˘ ≈ E
                          → isMappingI J → isInjectiveI J
                          → Id ⊓ J ˘ ⨾ J ≈ Id ⊓ E₁ ˘ ⨾ E₁
                          → IsSymSplitting E (E₁ ⨾ J ˘)
splitting-from-univalentI {A} {B} {Q} {E} {E₁} {J}
    unival-E₁ factors-E₁ mapping-J inj-J rJ≈rE₁
  = splitting-from-univalent {A} {B} {Q} {E} {E₁} {J}
    (isUnivalent-from-I unival-E₁) factors-E₁
    (isMapping-from-I mapping-J) (isInjective-from-I inj-J) ranJ≈ranE₁
  where
    ranJ≈ranE₁ : ran J ≈ ran E₁
    ranJ≈ranE₁ = ≈-begin
        ran J
      ≈⟨ ran-def ⟩
        Id ⊓ J ˘ ⨾ J
      ≈⟨ rJ≈rE₁ ⟩
        Id ⊓ E₁ ˘ ⨾ E₁
```

```
    ≈˘⟨ ran-def ⟩
      ran E₁
    □
```

Public re-**open**ing of modules previously **open**ed only locally for the **field** definitions:

```
  open OCC      occ          public
  open MeetOp meetOp         public
```

```
retractAllegory : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                → (F : Obj₂ → Obj₁) → Allegory j k₁ k₂ Obj₁ → Allegory j k₁ k₂ Obj₂
retractAllegory F base = let open Allegory base in record
  {occ = retractOCC F occ
  ; meetOp = retractMeetOp F meetOp
  ; Dedekind = Dedekind
  }
```

12.5 Categoric.USLSGC

```
module RawUSLSGC-Props
    {i j k₁ k₂ : Level} {Obj : Set i} (osgc : OSGC j k₁ k₂ Obj)
    (joinOp : JoinOp (OSGC.orderedSemigroupoid osgc))
  where
    open OSGC osgc
    open JoinOp joinOp
```

```
˘-⊔-supdistrib : {A B : Obj} {R S : Mor A B} → (R ˘ ⊔ S ˘) ⊑ (R ⊔ S) ˘
˘-⊔-supdistrib {A} {B} {R} {S} = ⊔-universal (˘-monotone ⊔-upper₁) (˘-monotone ⊔-upper₂)
˘-⊔-distrib : {A B : Obj} {R S : Mor A B} → (R ⊔ S) ˘ ≈ (R ˘ ⊔ S ˘)
˘-⊔-distrib {A} {B} {R} {S} = ⊑-antisym
  (˘-⊑-swap (⊔-cong ˘˘ ˘˘ ⟨≈˘⊑⟩ ˘-⊔-supdistrib))
  ˘-⊔-supdistrib
```

```
_isUnivalentUpto_ : {A B : Obj} → Mor A B → Mor B B → Set (i ⊍ j ⊍ k₂)
R isUnivalentUpto W = (R ˘ ⨾ R) isSubidUpto W
_isInjectiveUpto_ : {A B : Obj} → Mor A B → Mor A A → Set (i ⊍ j ⊍ k₂)
R isInjectiveUpto W = (R ⨾ R ˘) isSubidUpto W
```

```
record USLSGC {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field osgc : OSGC j k₁ k₂ Obj
  open OSGC osgc
  field joinOp          : JoinOp orderedSemigroupoid
  field joinCompDistrL  : JoinCompDistrL joinOp
  field joinCompDistrR  : JoinCompDistrR joinOp
  uslSemigroupoid : USLSemigroupoid j k₁ k₂ Obj
  uslSemigroupoid = record
    {orderedSemigroupoid = orderedSemigroupoid
    ; joinOp             = joinOp
    ; joinCompDistrL     = joinCompDistrL
    ; joinCompDistrR     = joinCompDistrR
    }
  open OSGC        osgc        public
  open JoinOp      joinOp      public
```

```
  open JoinCompDistrL      joinCompDistrL public
  open JoinCompDistrR      joinCompDistrR public
  open JoinCompDistr joinCompDistrL joinCompDistrR public
  open RawUSLSGC-Props osgc joinOp        public

retractUSLSGC : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
              → (F : Obj₂ → Obj₁)
              → USLSGC j k₁ k₂ Obj₁ → USLSGC j k₁ k₂ Obj₂
retractUSLSGC F base = let open USLSGC base in record
  {osgc = retractOSGC F osgc
  ; joinOp = retractJoinOp F joinOp
  ; joinCompDistrL = retractJoinCompDistrL F joinCompDistrL
  ; joinCompDistrR = retractJoinCompDistrR F joinCompDistrR
  }
```

12.6 Categoric.USLCC

```
record USLCC {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field occ : OCC j k₁ k₂ Obj
  open OCC occ
  field joinOp          : JoinOp orderedSemigroupoid
  field joinCompDistrL  : JoinCompDistrL joinOp
  field joinCompDistrR  : JoinCompDistrR joinOp
  uslCategory : USLCategory j k₁ k₂ Obj
  uslCategory = record
    {orderedCategory = orderedCategory
    ; joinOp          = joinOp
    ; joinCompDistrL  = joinCompDistrL
    ; joinCompDistrR  = joinCompDistrR
    }
  uslsgc : USLSGC j k₁ k₂ Obj
  uslsgc = record
    {osgc            = osgc
    ; joinOp         = joinOp
    ; joinCompDistrL = joinCompDistrL
    ; joinCompDistrR = joinCompDistrR
    }
  open JoinOp              joinOp         public
  open JoinCompDistrL      joinCompDistrL public
  open JoinCompDistrR      joinCompDistrR public
  open JoinCompDistr joinCompDistrL joinCompDistrR public
  open RawUSLSGC-Props osgc joinOp        public
  open Cotabulation        occ  joinOp joinCompDistrL joinCompDistrR public
  open OCC                 occ            public

retractUSLCC : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
             → (F : Obj₂ → Obj₁)
             → USLCC j k₁ k₂ Obj₁ → USLCC j k₁ k₂ Obj₂
retractUSLCC F base = let open USLCC base in record
  {occ = retractOCC F occ
  ; joinOp = retractJoinOp F joinOp
  ; joinCompDistrL = retractJoinCompDistrL F joinCompDistrL
  ; joinCompDistrR = retractJoinCompDistrR F joinCompDistrR
  }
```

12.7 Categoric.USLCCZ.Monolithic

```
record USLCCZ′ {i j k₁ k₂ : Level} {Obj : Set i}
               (Hom : Obj → Obj → Poset j k₁ k₂) : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  field category : Category′ (λ A B → posetSetoid (Hom A B))
  open Category′ category
  infix 4 _⊑_; infix 10 _˘; infixr 5 _⊔_
  _⊑_ = λ {A} {B} → Poset._≤_ (Hom A B)
  field
    ⨾-monotone   : {A B C : Obj} {f f' : Mor A B} {g g' : Mor B C}
                 → f ⊑ f' → g ⊑ g' → (f ⨾ g) ⊑ (f' ⨾ g')
    _˘           : {A B : Obj}                         → Mor A B → Mor B A
    ˘˘           : {A B : Obj}   {R : Mor A B}          → (R ˘) ˘ ≈ R
    ˘-involution : {A B C : Obj} {R : Mor A B} {S : Mor B C} → (R ⨾ S) ˘ ≈ S ˘ ⨾ R ˘
    ˘-monotone   : {A B : Obj} {R S : Mor A B} → R ⊑ S → (R ˘) ⊑ (S ˘)
    _⊔_          : {A B : Obj} → Mor A B → Mor A B → Mor A B
    ⊔-upper₁     : {A B : Obj} {R S : Mor A B} → R ⊑ (R ⊔ S)
    ⊔-upper₂     : {A B : Obj} {R S : Mor A B} → S ⊑ (R ⊔ S)
    ⊔-universal  : {A B : Obj} {R S X : Mor A B} → R ⊑ X → S ⊑ X → (R ⊔ S) ⊑ X
    ⨾-⊔-subdistribL : {A B C : Obj} {R₁ R₂ : Mor A B} {S : Mor B C}
                   → ((R₁ ⊔ R₂) ⨾ S) ⊑ ((R₁ ⨾ S) ⊔ (R₂ ⨾ S))
    ⊥ : {A B : Obj} → Mor A B
    ⊥-⊑ : {A B : Obj} {R : Mor A B} → ⊥ ⊑ R
    leftZero⊑ : {A B C : Obj} {R : Mor B C} → (⊥ {A} {B} ⨾ R) ⊑ ⊥
  orderedSemigroupoid : OrderedSemigroupoid j k₁ k₂ Obj
  orderedSemigroupoid = record {Hom = Hom; compOp = compOp
    ; locOrd = record {⨾-monotone = ⨾-monotone}}
  convOp : ConvOp semigroupoid
  convOp = record {_˘ = _˘
    ; ˘-cong = λ {A} {B} {R} {S} R≈S → Poset.antisym (Hom B A)
      (˘-monotone (Poset.reflexive (Hom A B) R≈S))
      (˘-monotone (Poset.reflexive (Hom A B) (≈-sym R≈S)))
    ; ˘˘ = ˘˘
    ; ˘-involution = ˘-involution
    }
  open ConvOp convOp using (˘-cong; ˘-coinvolution)
  osgc : OSGC j k₁ k₂ Obj
  osgc = record {OSGC_Base = record
    {orderedSemigroupoid = orderedSemigroupoid
    ; convOp = convOp
    ; ˘-monotone = ˘-monotone
    }}
  occ : OCC j k₁ k₂ Obj
  occ = record {OCC_Base = record {osgc = osgc; idOp = idOp}}
  botMor : BotMor orderedSemigroupoid
  botMor = record {leastMor = λ {A} {B} → record {mor = ⊥; proof = λ _ → ⊥-⊑}}
  open BotMor botMor using ()
  open OSGC-BotMor osgc botMor
  leftZeroLaw : LeftZeroLaw botMor
  leftZeroLaw = record {leftZero⊑ = leftZero⊑}
  open LeftZeroLaw leftZeroLaw using (leftZero)
  zeroMor : ZeroMor orderedSemigroupoid
  zeroMor = record
    {botMor = botMor
    ; leftZeroLaw = record {leftZero⊑ = leftZero⊑}
    ; rightZeroLaw = record {rightZero⊑ =
      λ {A} {B} {C} {R} → Poset.reflexive (Hom A C) (≈-begin
```

```
          R ⨾ ⊥
        ≈˘⟨ ˘-coinvolution ⟩
          (⊥ ˘ ⨾ R ˘) ˘
        ≈⟨ ˘-cong (⨾-cong₁ ⊥˘ ⟨≈≈⟩ leftZero) ⟩
          ⊥ ˘
        ≈⟨ ⊥˘ ⟩
          ⊥
        □)
      }
    }
  joinOp            : JoinOp orderedSemigroupoid
  joinOp            = record {join = λ {A} {B} R S → record
                        {value = R ⊔ S
                        ; proof = record {bound₁ = ⊔-upper₁; bound₂ = ⊔-upper₂; universal = ⊔-universal}
                        }}
  open JoinOp joinOp using (⊔-cong)
  joinCompDistrL : JoinCompDistrL joinOp
  joinCompDistrL = record {⨾-⊔-subdistribL = ⨾-⊔-subdistribL}
  open JoinCompDistrL joinCompDistrL using (⨾-⊔-distribL)
  open RawUSLSGC-Props osgc joinOp using (˘-⊔-distrib)
  joinCompDistrR : JoinCompDistrR joinOp
  joinCompDistrR = record {⨾-⊔-subdistribR =
    λ {A} {B} {C} {R} {S₁} {S₂} → Poset.reflexive (Hom A C) (≈-begin
          (R ⨾ (S₁ ⊔ S₂))
        ≈˘⟨ ˘-coinvolution ⟩
          ((S₁ ⊔ S₂) ˘ ⨾ R ˘) ˘
        ≈⟨ ˘-cong (⨾-cong₁ ˘-⊔-distrib) ⟩
          ((S₁ ˘ ⊔ S₂ ˘) ⨾ R ˘) ˘
        ≈⟨ ˘-cong ⨾-⊔-distribL ⟩
          (S₁ ˘ ⨾ R ˘ ⊔ S₂ ˘ ⨾ R ˘) ˘
        ≈⟨ ˘-⊔-distrib ⟩
          (S₁ ˘ ⨾ R ˘) ˘ ⊔ (S₂ ˘ ⨾ R ˘) ˘
        ≈⟨ ⊔-cong ˘-coinvolution ˘-coinvolution ⟩
          ((R ⨾ S₁) ⊔ (R ⨾ S₂))
        □)
    }
  uslcc : USLCC j k₁ k₂ Obj
  uslcc = record
    {occ              = occ
    ; joinOp          = joinOp
    ; joinCompDistrL  = joinCompDistrL
    ; joinCompDistrR  = joinCompDistrR
    }
  open OSGC osgc using (Mapping; module Mapping)
```

12.8 Categoric.DistrLatSGC

```
record DistrLatSGC {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field osgc : OSGC j k₁ k₂ Obj
  open OSGC osgc
  field meetOp : MeetOp orderedSemigroupoid
  field joinOp : JoinOp orderedSemigroupoid
  latticeSemigroupoid : LatticeSemigroupoid j k₁ k₂ Obj
  latticeSemigroupoid = record
    {orderedSemigroupoid = orderedSemigroupoid
    ; meetOp = meetOp
```

```
    ; joinOp  =  joinOp
    }
  field homLatDistr  :  HomLatticeDistr latticeSemigroupoid
  field joinCompDistrL  :  JoinCompDistrL joinOp
  field joinCompDistrR  :  JoinCompDistrR joinOp
  distrLatSemigroupoid  :  DistrLatSemigroupoid j k₁ k₂ Obj
  distrLatSemigroupoid  =  record
    {latticeSemigroupoid  =  latticeSemigroupoid
    ; homLatDistr          =  homLatDistr
    ; joinCompDistrL       =  joinCompDistrL
    ; joinCompDistrR       =  joinCompDistrR
    }
  uslsgc  :  USLSGC j k₁ k₂ Obj
  uslsgc  =  record
    {osgc                  =  osgc
    ; joinOp               =  joinOp
    ; joinCompDistrL       =  joinCompDistrL
    ; joinCompDistrR       =  joinCompDistrR
    }
  open OSGC                osgc              public
  open MeetOp              meetOp            public
  open JoinOp              joinOp            public
  open RawUSLSGC-Props osgc joinOp           public
  open HomLatticeDistr     homLatDistr       public
    -- In Agda-2.3.0, using takes too much time and memory.
    -- open DistrLatSemigroupoid distrLatSemigroupoid public using (lslSemigroupoid; uslSemigroupoid)
  lslSemigroupoid   =  DistrLatSemigroupoid.lslSemigroupoid  distrLatSemigroupoid
  uslSemigroupoid  =  DistrLatSemigroupoid.uslSemigroupoid distrLatSemigroupoid
  open JoinCompDistrL      joinCompDistrL public
  open JoinCompDistrR      joinCompDistrR public
```

12.9 Categoric.DistrLatCC

```
record DistrLatCC {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field  occ  :  OCC j k₁ k₂ Obj
  open OCC occ
  field meetOp  :  MeetOp orderedSemigroupoid
  field joinOp  :  JoinOp orderedSemigroupoid
  latticeSemigroupoid  :  LatticeSemigroupoid j k₁ k₂ Obj
  latticeSemigroupoid  =  record
    {orderedSemigroupoid  =  orderedSemigroupoid
    ; meetOp  =  meetOp
    ; joinOp  =  joinOp
    }
  field homLatDistr       :  HomLatticeDistr latticeSemigroupoid
  field joinCompDistrL  :  JoinCompDistrL joinOp
  field joinCompDistrR  :  JoinCompDistrR joinOp
  distrLatSGC  :  DistrLatSGC j k₁ k₂ Obj
  distrLatSGC  =  record
    {osgc  =  osgc
    ; meetOp  =  meetOp
    ; joinOp  =  joinOp
    ; homLatDistr       =  homLatDistr
    ; joinCompDistrL  =  joinCompDistrL
    ; joinCompDistrR  =  joinCompDistrR
```

```
  }
uslcc : USLCC j k₁ k₂ Obj
uslcc = record
  {occ = occ
  ; joinOp = joinOp
  ; joinCompDistrL = joinCompDistrL
  ; joinCompDistrR = joinCompDistrR
  }
open MeetOp           meetOp                             public
open JoinOp           joinOp                             public
open HomLattice orderedSemigroupoid meetOp joinOp public
open HomLatticeDistr homLatDistr                     public
open JoinCompDistrL  joinCompDistrL                  public
open JoinCompDistrR joinCompDistrR                   public
  -- open DistrLatSGC distrLatSGC public
    -- using (lslSemigroupoid; uslSemigroupoid; distrLatSemigroupoid; uslsgc)
lslSemigroupoid       = DistrLatSGC.lslSemigroupoid       distrLatSGC
uslSemigroupoid       = DistrLatSGC.uslSemigroupoid       distrLatSGC
distrLatSemigroupoid  = DistrLatSGC.distrLatSemigroupoid distrLatSGC
uslsgc                = DistrLatSGC.uslsgc                distrLatSGC
  -- open USLCC uslcc public
    -- using (uslCategory)
uslCategory = USLCC.uslCategory uslcc
open OCC              occ                                public
```

12.10 Categoric.SemiCollagory

```
record SemiCollagory {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field semiAllegory : SemiAllegory j k₁ k₂ Obj
  open SemiAllegory semiAllegory
  field joinOp : JoinOp orderedSemigroupoid
  latticeSemigroupoid : LatticeSemigroupoid j k₁ k₂ Obj
  latticeSemigroupoid = record
    {orderedSemigroupoid = orderedSemigroupoid
    ; meetOp = meetOp
    ; joinOp = joinOp
    }
  field homLatDistr : HomLatticeDistr latticeSemigroupoid
  field joinCompDistrL  : JoinCompDistrL joinOp
  field joinCompDistrR  : JoinCompDistrR joinOp
  distrLatSGC : DistrLatSGC j k₁ k₂ Obj
  distrLatSGC = record
    {osgc = osgc
    ; meetOp = meetOp
    ; joinOp = joinOp
    ; homLatDistr      = homLatDistr
    ; joinCompDistrL  = joinCompDistrL
    ; joinCompDistrR  = joinCompDistrR
    }
  uslsgc : USLSGC j k₁ k₂ Obj
  uslsgc = record
    {osgc              = osgc
    ; joinOp           = joinOp
    ; joinCompDistrL  = joinCompDistrL
    ; joinCompDistrR  = joinCompDistrR
```

```
    }
  open JoinOp              joinOp          public
  open RawUSLSGC-Props osgc joinOp         public
  open HomLatticeDistr     homLatDistr     public
    -- In Agda-2.3.0, using takes too much time and memory.
    -- open DistrLatSGC distrLatSGC public using (distrLatSemigroupoid; uslSemigroupoid)
  distrLatSemigroupoid = DistrLatSGC.distrLatSemigroupoid distrLatSGC
  uslSemigroupoid      = DistrLatSGC.uslSemigroupoid      distrLatSGC
  open JoinCompDistrL      joinCompDistrL  public
  open JoinCompDistrR      joinCompDistrR  public
  open SemiAllegory        semiAllegory    public
```

12.11 Categoric.Collagory

```
record Collagory {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field allegory : Allegory j k₁ k₂ Obj
  open Allegory allegory
  field joinOp : JoinOp orderedSemigroupoid
  latticeSemigroupoid : LatticeSemigroupoid j k₁ k₂ Obj
  latticeSemigroupoid = record
    {orderedSemigroupoid = orderedSemigroupoid
    ; meetOp = meetOp
    ; joinOp = joinOp
    }
  field homLatDistr     : HomLatticeDistr latticeSemigroupoid
  field joinCompDistrL  : JoinCompDistrL joinOp
  field joinCompDistrR  : JoinCompDistrR joinOp
  distrLatCC : DistrLatCC j k₁ k₂ Obj
  distrLatCC = record
    {occ             = occ
    ; meetOp         = meetOp
    ; joinOp         = joinOp
    ; homLatDistr    = homLatDistr
    ; joinCompDistrL = joinCompDistrL
    ; joinCompDistrR = joinCompDistrR
    }
  semiCollagory : SemiCollagory j k₁ k₂ Obj
  semiCollagory = record
    {semiAllegory    = semiAllegory
    ; joinOp         = joinOp
    ; joinCompDistrL = joinCompDistrL
    ; joinCompDistrR = joinCompDistrR
    ; homLatDistr    = homLatDistr
    }
  open JoinOp          joinOp                                     public
  open HomLattice orderedSemigroupoid meetOp joinOp               public
  open HomLatticeDistr homLatDistr                                public
    -- As of Agda-2.3.0, using is still too expensive.
    -- open DistrLatCC distrLatCC public
      -- using (distrLatSGC; distrLatSemigroupoid; uslSemigroupoid; uslsgc; uslCategory; uslcc)
  distrLatSGC          = DistrLatCC.distrLatSGC          distrLatCC
  distrLatSemigroupoid = DistrLatCC.distrLatSemigroupoid distrLatCC
  uslSemigroupoid      = DistrLatCC.uslSemigroupoid      distrLatCC
  uslsgc               = DistrLatCC.uslsgc               distrLatCC
```

```
  uslCategory            = DistrLatCC.uslCategory          distrLatCC
  uslcc                  = DistrLatCC.uslcc                distrLatCC
  open JoinCompDistrL  joinCompDistrL                  public
  open JoinCompDistrR joinCompDistrR                   public
  open Allegory            allegory                    public


retractCollagory : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
  → (F : Obj₂ → Obj₁) → Collagory j k₁ k₂ Obj₁ → Collagory j k₁ k₂ Obj₂
retractCollagory F base = let open Collagory base in record
  {allegory = retractAllegory F allegory
  ; joinOp = retractJoinOp F joinOp
  ; homLatDistr = retractHomLatticeDistr F homLatDistr
  ; joinCompDistrL = retractJoinCompDistrL F joinCompDistrL
  ; joinCompDistrR = retractJoinCompDistrR F joinCompDistrR
  }
```

12.12 Categoric.DistrSemiAllegory

```
record DistrSemiAllegory {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field semiCollagory : SemiCollagory j k₁ k₂ Obj
  open SemiCollagory  semiCollagory
  field zeroMor        : ZeroMor orderedSemigroupoid
  open SemiCollagory semiCollagory public
  open ZeroMor        zeroMor        public
```

12.13 Categoric.DistrAllegory

```
record DistrAllegory {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field collagory : Collagory j k₁ k₂ Obj
  open Collagory  collagory
  field zeroMor  : ZeroMor orderedSemigroupoid
  distrSemiAllegory : DistrSemiAllegory j k₁ k₂ Obj
  distrSemiAllegory = record
    {semiCollagory = semiCollagory
    ; zeroMor = zeroMor
    }
  open Collagory collagory public
  open ZeroMor  zeroMor  public


retractDistrAllegory : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
  → (F : Obj₂ → Obj₁) → DistrAllegory j k₁ k₂ Obj₁ → DistrAllegory j k₁ k₂ Obj₂
retractDistrAllegory F base = let open DistrAllegory base in record
  {collagory = retractCollagory F collagory
  ; zeroMor = retractZeroMor F zeroMor
  }
```

Chapter 13

Residuals and Division Allegories

Residuals of composition only need the context of locally ordered semigroupoids for their definition and a number of their properties (Sect. 13.1). Some additional properties hold in ordered categories (Sect. 13.2). In the presence of converse, a right-residual operator can be derived from a left-residual operator, and vice versa (Sect. 13.3).

Restricted residuals were first introduced by Kahl (2008) in the context of semigroupoids motivated by applications to finite relations between infinite sets, where the residuals of finite relations are not necessarily finite again. Restricted residuals restrict attention to the "interesting part" of residuals and preserve finiteness in that context; they have since also found applications in the context of relational substitutions (Kahl, 2010). Their definition, in Sect. 13.4, requires the setting of ordered semigroupoids with domain. If, in that setting, residuals are available, they can be used to define restricted residuals (Sect. 13.5).

In allegories with residuals, one can form *symmetric quotients*, which were originally studied by Berghammer et al. (1986, 1989). In the spirit of the axiomatic definitions of the simple residuals, Furusawa and Kahl (1998) gave a general axiomatic definition in distributive allegories without assuming existence of the simple residuals; Kahl (2008) provided the definition in the context of OSGCs that is formalised in Sect. 13.6. In semi-allegories with residuals, we recover the original definition of symmetric quotients (Sect. 13.9).

Division allegories (Sect. 13.9) were introduced by Freyd and Scedrov (1990) as distributive allegories with residuals.

13.1 Categoric.OrderedSemigroupoid.Residuals

```
record LeftResOp {i j k₁ k₂ : Level} {Obj : Set i}
                 (base : OrderedSemigroupoid j k₁ k₂ Obj)
                 : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  open OrderedSemigroupoid base
  infixl 9 _/_
  field
    _/_ : {A B C : Obj} → Mor A C → Mor B C → Mor A B
    /-cancel-outer : {A B C : Obj} {S : Mor A C} {R : Mor B C} → (S / R) ⨾ R ⊑ S
    /-universal  :   {A B C : Obj} {S : Mor A C} {R : Mor B C} {Q : Mor A B}
                 → Q ⨾ R ⊑ S → Q ⊑ S / R
  /-universal′   :   {A B C : Obj} {S : Mor A C} {R : Mor B C} {Q : Mor A B}
                 → Q ⊑ S / R → Q ⨾ R ⊑ S
  /-universal′ Q⊑S/R = ⨾-monotone₁ Q⊑S/R ⟨⊑⊑⟩ /-cancel-outer
  /-cancel-inner : {A B C : Obj} {T : Mor A B} {S : Mor B C} → T ⊑ (T ⨾ S) / S
  /-cancel-inner = /-universal ⊑-refl
  /-monotone : {A B C : Obj} {S₁ S₂ : Mor A C} {R : Mor B C} → S₁ ⊑ S₂ → S₁ / R ⊑ S₂ / R
  /-monotone S₁⊑S₂ = /-universal (/-cancel-outer ⟨⊑⊑⟩ S₁⊑S₂)
  /-cong₁ : {A B C : Obj} {S₁ S₂ : Mor A C} {R : Mor B C} → S₁ ≈ S₂ → S₁ / R ≈ S₂ / R
```

```
  /-cong₁ S₁≈S₂ = ⊑-antisym (/-monotone (⊑-reflexive S₁≈S₂)) (/-monotone (⊑-reflexive′ S₁≈S₂))
  /-antitone : {A B C : Obj} {S : Mor A C} {R₁ R₂ : Mor B C} → R₂ ⊑ R₁ → S / R₁ ⊑ S / R₂
  /-antitone R₂⊑R₁ = /-universal (⨾-monotone₂ R₂⊑R₁ ⟨⊑⊑⟩ /-cancel-outer)
  /-cong₂ : {A B C : Obj} {S : Mor A C} {R₁ R₂ : Mor B C} → R₁ ≈ R₂ → S / R₁ ≈ S / R₂
  /-cong₂ R₁≈R₂ = ⊑-antisym (/-antitone (⊑-reflexive′ R₁≈R₂)) (/-antitone (⊑-reflexive R₁≈R₂))
  /-cong :    {A B C : Obj} {S₁ S₂ : Mor A C} {R₁ R₂ : Mor B C}
          →   S₁ ≈ S₂ → R₁ ≈ R₂ → S₁ / R₁ ≈ S₂ / R₂
  /-cong S₁≈S₂ R₁≈R₂ = /-cong₁ S₁≈S₂ ⟨≈≈⟩ /-cong₂ R₁≈R₂
  /-cancel-outer²    :    {A B C D : Obj} {S : Mor A D} {R : Mor B D} {T : Mor C D}
                       → (S / R) ⨾ (R / T) ⨾ T ⊑ S
  /-cancel-outer²    =    ⨾-monotone₂ /-cancel-outer ⟨⊑⊑⟩ /-cancel-outer
  /-cancel-middle    :    {A B C D : Obj} {S : Mor A D} {R : Mor B D} {T : Mor C D}
                       → (S / R) ⨾ (R / T) ⊑ S / T
  /-cancel-middle    =    /-universal (⨾-assoc ⟨≈⊑⟩ /-cancel-outer²)
  /-cancel-⨾   :    {A B C D : Obj} {S : Mor A C} {R : Mor B C} {T : Mor C D}
                 → S / R ⊑ (S ⨾ T) / (R ⨾ T)
  /-cancel-⨾   =    /-universal (⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ /-cancel-outer)
  /-outer-⨾ :    {A B C D : Obj} {F : Mor A B} {S : Mor B D} {R : Mor C D}
              → F ⨾ (S / R) ⊑ (F ⨾ S) / R
  /-outer-⨾ =    /-universal (⨾-assoc ⟨≈⊑⟩ ⨾-monotone₂ /-cancel-outer)
  // : {A B C D : Obj} {Q : Mor B C} {R : Mor C D} {S : Mor A D}
              → (S / R) / Q ≈ S / (Q ⨾ R)
  // {Q = Q} {R} {S} = ⊑-antisym
    (/-universal ((⊑-begin
          ((S / R) / Q) ⨾ (Q ⨾ R)
       ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ /-cancel-outer ⟩
          (S / R) ⨾ R
       ⊑⟨ /-cancel-outer ⟩
          S
       □
    )))
    (/-universal (/-universal (⊑-begin
        ((S / (Q ⨾ R)) ⨾ Q) ⨾ R
     ⊑⟨ ⨾-assoc ⟨≈⊑⟩ /-cancel-outer ⟩
        S
     □
    )))
  /-cancel-⨾-inner        :  {A B C D : Obj} {Q : Mor B C} {R : Mor C D} {S : Mor A D}
                         → (S / (Q ⨾ R)) ⨾ Q ⊑ S / R
  /-cancel-⨾-inner {Q = Q} {R} {S} = ⊑-begin
       (S / (Q ⨾ R)) ⨾ Q
    ≈˘⟨ ⨾-cong₁ // ⟩
       ((S / R) / Q) ⨾ Q
    ⊑⟨ /-cancel-outer ⟩
       S / R
    □

record RightResOp {i j k₁ k₂ : Level} {Obj : Set i}
                  (base : OrderedSemigroupoid j k₁ k₂ Obj)
                  : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  open OrderedSemigroupoid base
  infixr 9 _\_
  field
    _\_ : {A B C : Obj} → Mor A B → Mor A C → Mor B C
    \-cancel-outer :  {A B C : Obj} {S : Mor A C} {Q : Mor A B} → Q ⨾ (Q \ S) ⊑ S
    \-universal   :    {A B C : Obj} {S : Mor A C} {Q : Mor A B} {R : Mor B C}
```

```
                  → Q ⨾ R ⊑ S → R ⊑ Q \ S
\-universal′    :   {A B C : Obj} {S : Mor A C} {Q : Mor A B} {R : Mor B C}
                  → R ⊑ Q \ S → Q ⨾ R ⊑ S
\-universal′ R⊑Q\S = ⨾-monotone₂ R⊑Q\S ⟨⊑⊑⟩ \-cancel-outer

\-cancel-inner : {A B C : Obj} {T : Mor B C} {S : Mor A B} → T ⊑ S \ (S ⨾ T)
\-cancel-inner = \-universal ⊑-refl

\-monotone : {A B C : Obj} {S₁ S₂ : Mor A C} {Q : Mor A B} → S₁ ⊑ S₂ → Q \ S₁ ⊑ Q \ S₂
\-monotone S₁⊑S₂ = \-universal (\-cancel-outer ⟨⊑⊑⟩ S₁⊑S₂)

\-cong₂ : {A B C : Obj} {S₁ S₂ : Mor A C} → {Q : Mor A B} → S₁ ≈ S₂ → Q \ S₁ ≈ Q \ S₂
\-cong₂ S₁≈S₂ = ⊑-antisym (\-monotone (⊑-reflexive S₁≈S₂)) (\-monotone (⊑-reflexive′ S₁≈S₂))

\-antitone : {A B C : Obj} {S : Mor A C} {Q₁ Q₂ : Mor A B} → Q₂ ⊑ Q₁ → Q₁ \ S ⊑ Q₂ \ S
\-antitone Q₂⊑Q₁ = \-universal (⨾-monotone₁ Q₂⊑Q₁ ⟨⊑⊑⟩ \-cancel-outer)

\-cong₁ : {A B C : Obj} {S : Mor A C} {Q₁ Q₂ : Mor A B} → Q₁ ≈ Q₂ → Q₁ \ S ≈ Q₂ \ S
\-cong₁ Q₁≈Q₂ = ⊑-antisym (\-antitone (⊑-reflexive′ Q₁≈Q₂)) (\-antitone (⊑-reflexive Q₁≈Q₂))

\-cong :   {A B C : Obj} {S₁ S₂ : Mor A C} {Q₁ Q₂ : Mor A B}
         → Q₁ ≈ Q₂ → S₁ ≈ S₂ → Q₁ \ S₁ ≈ Q₂ \ S₂
\-cong Q₁≈Q₂ S₁≈S₂ = \-cong₂ S₁≈S₂ ⟨≈≈⟩ \-cong₁ Q₁≈Q₂

\-cancel-outer²   :   {A B C D : Obj} {S : Mor A D} {Q : Mor A C} {T : Mor A B}
                    → T ⨾ (T \ Q) ⨾ (Q \ S) ⊑ S
\-cancel-outer²   =   ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ \-cancel-outer ⟨⊑⊑⟩ \-cancel-outer

\-cancel-middle   :   {A B C D : Obj} {S : Mor A D} {Q : Mor A C} {T : Mor A B}
                    → (T \ Q) ⨾ (Q \ S) ⊑ T \ S
\-cancel-middle   =   \-universal \-cancel-outer²

\-cancel-⨾   :   {A B C D : Obj} {S : Mor B D} {Q : Mor B C} {T : Mor A B}
               → Q \ S ⊑ (T ⨾ Q) \ (T ⨾ S)
\-cancel-⨾   =   \-universal (⨾-assoc ⟨≈⊑⟩ ⨾-monotone₂ \-cancel-outer)

\-outer-⨾ :   {A B C D : Obj} {F : Mor C D} {S : Mor A C} {Q : Mor A B}
            → (Q \ S) ⨾ F ⊑ Q \ (S ⨾ F)
\-outer-⨾ =   \-universal (⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ \-cancel-outer)

\\ : {A B C D : Obj} {Q : Mor A B} {R : Mor B C} {S : Mor A D}
           → R \ (Q \ S) ≈ (Q ⨾ R) \ S
\\ {Q = Q} {R} {S} = ⊑-antisym
  (\-universal ((⊑-begin
        (Q ⨾ R) ⨾ R \ Q \ S
      ⊑⟨ ⨾-assoc ⟨≈⊑⟩ ⨾-monotone₂ \-cancel-outer ⟩
        Q ⨾ (Q \ S)
      ⊑⟨ \-cancel-outer ⟩
        S
      □
  )))
  (\-universal (\-universal (⊑-begin
      Q ⨾ R ⨾ ((Q ⨾ R) \ S)
    ⊑⟨ ⨾-assocL ⟨≈⊑⟩ \-cancel-outer ⟩
      S
    □
  )))

\-cancel-⨾-inner      :  {A B C D : Obj} {Q : Mor A B} {R : Mor B C} {S : Mor A D}
                       → R ⨾ ((Q ⨾ R) \ S) ⊑ Q \ S
\-cancel-⨾-inner {Q = Q} {R} {S} = ⊑-begin
    R ⨾ ((Q ⨾ R) \ S)
  ≈˘⟨ ⨾-cong₂ \\ ⟩
    R ⨾ (R \ (Q \ S))
  ⊑⟨ \-cancel-outer ⟩
    Q \ S
  □
```

```
module ResidualOps {i j k₁ k₂ : Level} {Obj : Set i}
                   {base : OrderedSemigroupoid j k₁ k₂ Obj}
                   (leftResOp : LeftResOp base)
                   (rightResOp : RightResOp base) where
  open OrderedSemigroupoid base
  open LeftResOp  leftResOp  public
  open RightResOp rightResOp public

  \/-⊑ : {A B C D : Obj} {S : Mor A D} {Q : Mor A B} {R : Mor C D} → Q \ (S / R) ⊑ (Q \ S) / R
  \/-⊑ {S = S} {Q} {R} = /-universal (\-outer-⨾ ⟨⊑⊑⟩ \-monotone /-cancel-outer)

  \/-⊒ : {A B C D : Obj} {S : Mor A D} {Q : Mor A B} {R : Mor C D} → (Q \ S) / R ⊑ Q \ (S / R)
  \/-⊒ {S = S} {Q} {R} = \-universal (/-outer-⨾ ⟨⊑⊑⟩ /-monotone \-cancel-outer)

  \/-≈ : {A B C D : Obj} {S : Mor A D} {Q : Mor A B} {R : Mor C D} → Q \ (S / R) ≈ (Q \ S) / R
  \/-≈ {S = S} {Q} {R} = ⊑-antisym \/-⊑ \/-⊒

  /-twist : {A B C D : Obj} {S : Mor A C} {R : Mor B C} {T : Mor D C} → S / R ⊑ (T / S) \ (T / R)
  /-twist = \-universal /-cancel-middle

  \-twist : {A B C D : Obj} {S : Mor A C} {Q : Mor A B} {T : Mor A D} → Q \ S ⊑ (Q \ T) / (S \ T)
  \-twist = /-universal \-cancel-middle

    -- (Furusawa and Kahl, 1998, Lemma 4.9.ii)
  /-twist-down : {A B C : Obj} {S : Mor A C} {R : Mor B C} → S / R ⊑ (R / S) \ (R / R)
  /-twist-down = \-universal /-cancel-middle

  \-twist-down : {A B C : Obj} {S : Mor A C} {Q : Mor A B} → Q \ S ⊑ (Q \ Q) / (S \ Q)
  \-twist-down = /-universal \-cancel-middle

  /-twist-up : {A B C : Obj} {S : Mor A C} {R : Mor B C} → S / R ⊑ (S / S) \ (S / R)
  /-twist-up = /-twist

  \-twist-up : {A B C : Obj} {S : Mor A C} {Q : Mor A B} → Q \ S ⊑ (Q \ S) / (S \ S)
  \-twist-up = \-twist
```

For /-twist-up in ordered categories, (Furusawa and Kahl, 1998, Lemma 4.9.i) showed ≈, using Id {A} ⊑ S / S, see Sect. 13.2. There is a two-element ordered semigroup that does not satisfy (S / S) ⨾ S ≈ S, and a three-element linearly ordered semigroup that does not satisfy ⊒.

```
  ⊑-S/◦\S : {A B C : Obj} {S : Mor A C} {Q : Mor A B} → Q ⊑ S / (Q \ S)
  ⊑-S/◦\S {A} {B} {C} {S} {Q} = /-universal (⊑-begin
      Q ⨾ (Q \ S)
    ⊑⟨ \-cancel-outer ⟩
      S
    □)

  ⊑-\S◦S/ : {A B C : Obj} {S : Mor A C} {R : Mor B C} → R ⊑ (S / R) \ S
  ⊑-\S◦S/ {A} {B} {C} {S} {R} = \-universal (⊑-begin
      (S / R) ⨾ R
    ⊑⟨ /-cancel-outer ⟩
      S
    □)


  S/◦\S◦S/ : {A B C : Obj} {S : Mor A C} {R : Mor B C} → S / ((S / R) \ S) ≈ S / R
  S/◦\S◦S/ {A} {B} {C} {S} {R} = ⊑-antisym (/-antitone ⊑-\S◦S/) ⊑-S/◦\S

  \S◦S/◦\S : {A B C : Obj} {S : Mor A C} {Q : Mor A B} → (S / (Q \ S)) \ S ≈ Q \ S
  \S◦S/◦\S {A} {B} {C} {S} {Q} = ⊑-antisym (\-antitone ⊑-S/◦\S) ⊑-\S◦S/


  T/◦\S◦S/ : {A₁ A₂ B C : Obj} {T : Mor A₁ C} {S : Mor A₂ C} {R : Mor B C}
    → (S / R) \ S ⊑ (T / R) \ T → T / ((S / R) \ S) ≈ T / R
  T/◦\S◦S/ {A₁} {A₂} {B} {C} {T} {S} {R} p = ⊑-antisym (/-antitone ⊑-\S◦S/)
    (⊑-S/◦\S ⟨⊑⊑⟩ /-antitone p)

  \T◦S/◦\S : {A B C₁ C₂ : Obj} {T : Mor A C₁} {S : Mor A C₂} {Q : Mor A B}
    → S / (Q \ S) ⊑ T / (Q \ T) → (S / (Q \ S)) \ T ≈ Q \ T
```

```
  \T∘S/∘\S {A} {B} {C₁} {C₂} {T} {S} {Q} p = ⊑-antisym (\-antitone ⊑-S/∘\S)
    (⊑-\S∘S/ ⟨⊑⊑⟩ \-antitone p)


retractLeftResOp : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                 → (F : Obj₂ → Obj₁)
                 → {base : OrderedSemigroupoid j k₁ k₂ Obj₁}
                 → LeftResOp base → LeftResOp (retractOrderedSemigroupoid F base)
retractLeftResOp F leftResOp = let open LeftResOp leftResOp in record
  { _/_ = _/_
  ; /-cancel-outer = /-cancel-outer
  ; /-universal = /-universal
  }


retractRightResOp : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                  → (F : Obj₂ → Obj₁)
                  → {base : OrderedSemigroupoid j k₁ k₂ Obj₁}
                  → RightResOp base → RightResOp (retractOrderedSemigroupoid F base)
retractRightResOp F rightResOp = let open RightResOp rightResOp in record
  { _\_ = _\_
  ; \-cancel-outer = \-cancel-outer
  ; \-universal = \-universal
  }
```

13.2 Categoric.OrderedCategory.Residuals

```
module OrdCat-LeftRes-Props {i j k₁ k₂ : Level} {Obj : Set i}
  (base : OrderedCategory j k₁ k₂ Obj)
  (leftResOp : LeftResOp (OrderedCategory.orderedSemigroupoid base))
    where
  open OrderedCategory base
  open LeftResOp leftResOp

  /-isReflexive : {A B : Obj} {R : Mor A B} → Id ⊑ R / R
  /-isReflexive = /-universal (⊑-reflexive leftId)

  /-isSuperidentity : {A B : Obj} {R : Mor A B} → isSuperidentity (R / R)
  /-isSuperidentity = reflexiveIsSuperidentity /-isReflexive

  /-Id : {A B : Obj} {R : Mor A B} → R / Id ≈ R
  /-Id {_} {_} {R} = ⊑-antisym
    (⊑-begin
        R / Id
      ≈⟨ ≈-sym rightId ⟩
        (R / Id) ⨾ Id
      ⊑⟨ /-cancel-outer ⟩
        R
      □)
    (/-universal (⊑-reflexive rightId))


module OrdCat-RightRes-Props {i j k₁ k₂ : Level} {Obj : Set i}
  (base : OrderedCategory j k₁ k₂ Obj)
  (rightResOp : RightResOp (OrderedCategory.orderedSemigroupoid base))
    where
  open OrderedCategory base
  open RightResOp rightResOp

  \-isReflexive : {A B : Obj} {R : Mor A B} → Id ⊑ R \ R
```

```
  \-isReflexive = \-universal (⊑-reflexive rightId)
  \-isSuperidentity : {A B : Obj} {R : Mor A B} → isSuperidentity (R \ R)
  \-isSuperidentity = reflexiveIsSuperidentity \-isReflexive
  Id-\ : {A B : Obj} {R : Mor A B} → Id \ R ≈ R
  Id-\ {_} {_} {R} = ⊑-antisym
    (⊑-begin
        Id \ R
      ≈⟨ ≈-sym leftId ⟩
        Id ⨾ (Id \ R)
      ⊑⟨ \-cancel-outer ⟩
        R
      □)
    (\-universal (⊑-reflexive leftId))


module OrdCat-Residual-Props {i j k₁ k₂ : Level} {Obj : Set i}
  (base : OrderedCategory j k₁ k₂ Obj)
  (leftResOp : LeftResOp (OrderedCategory.orderedSemigroupoid base))
  (rightResOp : RightResOp (OrderedCategory.orderedSemigroupoid base))
    where
  open OrderedCategory base
  open ResidualOps leftResOp rightResOp
  open OrdCat-LeftRes-Props   base leftResOp   public
  open OrdCat-RightRes-Props base rightResOp public
    -- (Furusawa and Kahl, 1998, Lemma 4.9.i)
  /-twist-up-≈ : {A B C : Obj} {S : Mor A C} {R : Mor B C} → S / R ≈ (S / S) \ (S / R)
  /-twist-up-≈ {S = S} {R} = ⊑-antisym /-twist-up
    (⊑-begin
        (S / S) \ (S / R)
      ⊑⟨ \-antitone /-isReflexive ⟩
        Id \ (S / R)
      ≈⟨ Id-\ ⟩
        S / R
      □)
```

13.3 Categoric.OSGC.Residuals

```
module OSGC-Residuals {i j k₁ k₂ : Level} {Obj : Set i}
  (base : OSGC j k₁ k₂ Obj)
  (leftResOp    : LeftResOp (OSGC.orderedSemigroupoid base))
  (rightResOp   : RightResOp (OSGC.orderedSemigroupoid base))
  where
  open OSGC base
  open LeftResOp leftResOp
  open RightResOp rightResOp


  \-inner-⨾-⊑ : {A B C D : Obj} {S : Mor A D} {Q : Mor A B} {F : Mor C B}
               → Q ⨾ F ˘ ⨾ F ⊑ Q → F ⨾ (Q \ S) ⊑ (Q ⨾ F ˘) \ S
  \-inner-⨾-⊑ {S = S} {Q} {F} Q⨾F˘⨾F⊑Q = \-universal (⊑-begin
        (Q ⨾ F ˘) ⨾ F ⨾ (Q \ S)
      ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ (⨾-assoc ⟨≈⊑⟩ Q⨾F˘⨾F⊑Q) ⟩
        Q ⨾ (Q \ S)
      ⊑⟨ \-cancel-outer ⟩
        S
      □)
```

```
/-inner-⨾-⊑ : {A B C D : Obj} {S : Mor A D} {R : Mor C D} {F : Mor B C}
            → F ˘ ⨾ F ⨾ R ⊑ R → (S / R) ⨾ F ˘ ⊑ S / (F ⨾ R)
/-inner-⨾-⊑ {S = S} {R} {F} F˘⨾F⨾R⊑R = /-universal (⊑-begin
      ((S / R) ⨾ F ˘) ⨾ (F ⨾ R)
    ⊑⟨ ⨾-assoc ⟨≈⊑⟩ ⨾-monotone₂ F˘⨾F⨾R⊑R ⟩
      (S / R) ⨾ R
    ⊑⟨ /-cancel-outer ⟩
      S
    □)

\-inner-⨾-unival  : {A B C D : Obj} {S : Mor A D} {Q : Mor A B} {F : Mor C B}
                  → isUnivalent F → F ⨾ (Q \ S) ⊑ (Q ⨾ F ˘) \ S
\-inner-⨾-unival F-unival = \-inner-⨾-⊑ (proj₂ F-unival)
/-inner-⨾-unival  : {A B C D : Obj} {S : Mor A D} {R : Mor C D} {F : Mor B C}
                  → isUnivalent F → (S / R) ⨾ F ˘ ⊑ S / (F ⨾ R)
/-inner-⨾-unival F-unival = /-inner-⨾-⊑ (⨾-assocL ⟨≈⊑⟩ proj₁ F-unival)

\-inner-⨾-total : {A B C D : Obj} {S : Mor A D} {Q : Mor A B} {F : Mor C B}
                → isTotal F → (Q ⨾ F ˘) \ S ⊑ F ⨾ (Q \ S)
\-inner-⨾-total {S = S} {Q} {F} F-total = ⊑-begin
      (Q ⨾ F ˘) \ S
    ⊑⟨ proj₁ F-total ⟨⊑≈⟩ ⨾-assoc ⟩
      F ⨾ F ˘ ⨾ ((Q ⨾ F ˘) \ S)
    ⊑⟨ ⨾-monotone₂₁ \-cancel-inner ⟩
      F ⨾ (Q \ (Q ⨾ F ˘)) ⨾ ((Q ⨾ F ˘) \ S)
    ⊑⟨ ⨾-monotone₂ \-cancel-middle ⟩
      F ⨾ (Q \ S)
    □
/-inner-⨾-total : {A B C D : Obj} {S : Mor A D} {R : Mor C D} {F : Mor B C}
                → isTotal F → S / (F ⨾ R) ⊑ (S / R) ⨾ F ˘
/-inner-⨾-total {S = S} {R} {F} F-total = ⊑-begin
      S / (F ⨾ R)
    ⊑⟨ proj₂ F-total ⟩
      (S / (F ⨾ R)) ⨾ F ⨾ F ˘
    ⊑⟨ ⨾-monotone₂₁ /-cancel-inner ⟩
      (S / (F ⨾ R)) ⨾ ((F ⨾ R) / R) ⨾ F ˘
    ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ /-cancel-middle ⟩
      (S / R) ⨾ F ˘
    □

\-inner-⨾   : {A B C D : Obj} {S : Mor A D} {Q : Mor A B} {F : Mor C B}
            → isMapping F → F ⨾ (Q \ S) ≈ (Q ⨾ F ˘) \ S
\-inner-⨾ {S = S} {Q} {F} (F-unival, F-total) = ⊑-antisym
  (\-inner-⨾-unival F-unival) (\-inner-⨾-total F-total)
/-inner-⨾   : {A B C D : Obj} {S : Mor A D} {R : Mor C D} {F : Mor B C}
            → isMapping F → (S / R) ⨾ F ˘ ≈ S / (F ⨾ R)
/-inner-⨾ {S = S} {R} {F} (F-unival, F-total) = ⊑-antisym
  (/-inner-⨾-unival F-unival) (/-inner-⨾-total F-total)

/-outer-⨾-⊒ : {A B C D : Obj} {F : Mor A B} {S : Mor B D} {R : Mor C D}
            → isMapping F → (F ⨾ S) / R ⊑ F ⨾ (S / R)
/-outer-⨾-⊒ {F = F} {S} {R} (F-unival, F-total) = ⊑-begin
    (F ⨾ S) / R
  ⊑⟨ proj₁ F-total ⟨⊑≈⟩ ⨾-assoc ⟩
    F ⨾ F ˘ ⨾ ((F ⨾ S) / R)
  ⊑⟨ ⨾-monotone₂ /-outer-⨾ ⟩
```

```
      F ⨾ ((F ˘ ⨾ F ⨾ S) / R)
    ⊑⟨ ⨾-monotone₂ (/-monotone (⨾-assocL ⟨≈⊑⟩ proj₁ F-unival)) ⟩
      F ⨾ (S / R)
    □

  /-outer-⨾-≈ : {A B C D : Obj} {F : Mor A B} {S : Mor B D} {R : Mor C D}
              → isMapping F → F ⨾ (S / R) ≈ (F ⨾ S) / R
  /-outer-⨾-≈ F-mapping = ⊑-antisym /-outer-⨾ (/-outer-⨾-⊒ F-mapping)

  /-flip-⊑ :   {A B C D : Obj} {S : Mor A D} {R : Mor B C} {F : Mor C D}
              → isTotal F → S / (R ⨾ F) ⊑ (S ⨾ F ˘) / R
  /-flip-⊑ {S = S} {R} {F} F-total = /-universal (⊑-begin
        (S / (R ⨾ F)) ⨾ R
      ⊑⟨ /-cancel-⨾-inner ⟩
        S / F
      ⊑⟨ proj₂ F-total ⟩
        (S / F) ⨾ F ⨾ F ˘
      ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ /-cancel-outer ⟩
        S ⨾ F ˘
      □)
  /-flip-⊒ :   {A B C D : Obj} {S : Mor A D} {R : Mor B C} {F : Mor C D}
              → isUnivalent F → (S ⨾ F ˘) / R ⊑ S / (R ⨾ F)
  /-flip-⊒ {S = S} {R} {F} F-unival = /-universal (⊑-begin
        ((S ⨾ F ˘) / R) ⨾ (R ⨾ F)
      ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ /-cancel-outer ⟩
        (S ⨾ F ˘) ⨾ F
      ⊑⟨ ⨾-assoc ⟨≈⊑⟩ proj₂ F-unival ⟩
        S
      □)
  /-flip      : {A B C D : Obj} {S : Mor A D} {R : Mor B C} {F : Mor C D}
              → isMapping F → S / (R ⨾ F) ≈ (S ⨾ F ˘) / R
  /-flip (F-unival, F-total) = ⊑-antisym (/-flip-⊑ F-total) (/-flip-⊒ F-unival)

  \-flip-⊑ :   {A B C D : Obj} {S : Mor A D} {F : Mor A B} {Q : Mor B C}
              → isSurjective F → (F ⨾ Q) \ S ⊑ Q \ (F ˘ ⨾ S)
  \-flip-⊑ {S = S} {F} {Q} F-surj = \-universal (⊑-begin
        Q ⨾ ((F ⨾ Q) \ S)
      ⊑⟨ \-cancel-⨾-inner ⟩
        F \ S
      ⊑⟨ proj₁ F-surj ⟨⊑≈⟩ ⨾-assoc ⟩
        F ˘ ⨾ F ⨾ (F \ S)
      ⊑⟨ ⨾-monotone₂ \-cancel-outer ⟩
        F ˘ ⨾ S
      □)
  \-flip-⊒ :   {A B C D : Obj} {S : Mor A D} {F : Mor A B} {Q : Mor B C}
              → isInjective F → Q \ (F ˘ ⨾ S) ⊑ (F ⨾ Q) \ S
  \-flip-⊒ {S = S} {F} {Q} F-inj = \-universal (⊑-begin
        (F ⨾ Q) ⨾ (Q \ (F ˘ ⨾ S))
      ⊑⟨ ⨾-assoc ⟨≈⊑⟩ ⨾-monotone₂ \-cancel-outer ⟩
        F ⨾ F ˘ ⨾ S
      ⊑⟨ ⨾-assocL ⟨≈⊑⟩ proj₁ F-inj ⟩
        S
      □)
  \-flip      : {A B C D : Obj} {S : Mor A D} {F : Mor A B} {Q : Mor B C}
              → isBijective F → (F ⨾ Q) \ S ≈ Q \ (F ˘ ⨾ S)
  \-flip (F-inj, F-surj) = ⊑-antisym (\-flip-⊑ F-surj) (\-flip-⊒ F-inj)

  /-˘ : {A B C : Obj} {S : Mor A C} {R : Mor B C} → (S / R) ˘ ≈ R ˘ \ S ˘
  /-˘ {A} {B} {C} {S} {R} = ⊑-antisym
```

```
    (\-universal (⊑-begin
          R ˘ ⨾ (S / R) ˘
       ≈˘⟨ ˘-involution ⟩
          ((S / R) ⨾ R) ˘
       ⊑⟨ ˘-monotone /-cancel-outer ⟩
          S ˘
       □))
    (⊑-˘-swap (/-universal (⊑-begin
          (R ˘ \ S ˘) ˘ ⨾ R
       ≈˘⟨ ˘-involutionLeftConv ⟩
          (R ˘ ⨾ (R ˘ \ S ˘)) ˘
       ⊑⟨ ˘-monotone \-cancel-outer ⟨⊑≈⟩ ˘˘ ⟩
          S
       □)))
/˘-˘ : {A B C : Obj} {S : Mor A C} {R : Mor C B} → (S / R ˘) ˘ ≈ R \ S ˘
/˘-˘ {A} {B} {C} {S} {R} = ≈-begin
          (S / R ˘) ˘
       ≈⟨ /-˘ ⟩
          R ˘ ˘ \ S ˘
       ≈⟨ \-cong₁ ˘˘ ⟩
          R \ S ˘
       □
˘/-˘ : {A B C : Obj} {S : Mor C A} {R : Mor B C} → (S ˘ / R) ˘ ≈ R ˘ \ S
˘/-˘ {A} {B} {C} {S} {R} = ≈-begin
          (S ˘ / R) ˘
       ≈⟨ /-˘ ⟩
          R ˘ \ S ˘ ˘
       ≈⟨ \-cong₂ ˘˘ ⟩
          R ˘ \ S
       □
˘/˘-˘ : {A B C : Obj} {S : Mor C A} {R : Mor C B} → (S ˘ / R ˘) ˘ ≈ R \ S
˘/˘-˘ {A} {B} {C} {S} {R} = ≈-begin
          (S ˘ / R ˘) ˘
       ≈⟨ /˘-˘ ⟩
          R \ S ˘ ˘
       ≈⟨ \-cong₂ ˘˘ ⟩
          R \ S
       □
\-˘ : {A B C : Obj} {Q : Mor A B} {S : Mor A C} → (Q \ S) ˘ ≈ S ˘ / Q ˘
\-˘ = ≈-sym (˘-≈-swap ˘/˘-˘)
˘\-˘ : {A B C : Obj} {Q : Mor B A} {S : Mor A C} → (Q ˘ \ S) ˘ ≈ S ˘ / Q
˘\-˘ = ≈-sym (˘-≈-swap ˘/-˘)
\˘-˘ : {A B C : Obj} {Q : Mor A B} {S : Mor C A} → (Q \ S ˘) ˘ ≈ S / Q ˘
\˘-˘ = ≈-sym (˘-≈-swap /˘-˘)
˘\˘-˘ : {A B C : Obj} {Q : Mor B A} {S : Mor C A} → (Q ˘ \ S ˘) ˘ ≈ S / Q
˘\˘-˘ = ≈-sym (˘-≈-swap /-˘)

\-cancel-inner-⊑ : {A B C : Obj} {T : Mor B C} {S : Mor A B}
                 → isLeftIdentity (S ˘ ⨾ S) → S \ (S ⨾ T) ⊑ T
\-cancel-inner-⊑ {T = T} {S} S-leftId = ⊑-begin
          S \ (S ⨾ T)
       ≈⟨ S-leftId ⟨≈˘≈⟩ ⨾-assoc ⟩
          S ˘ ⨾ S ⨾ (S \ (S ⨾ T))
       ⊑⟨ ⨾-monotone₂ \-cancel-outer ⟩
          S ˘ ⨾ S ⨾ T
       ≈⟨ ⨾-assocL ⟨≈≈⟩ S-leftId ⟩
          T
       □
```

```
  \-cancel-inner-≈ : {A B C : Obj} {T : Mor B C} {S : Mor A B}
                  → isLeftIdentity (S ˘ ⨾ S) → S \ (S ⨾ T) ≈ T
  \-cancel-inner-≈ S-leftId = ⊑-antisym (\-cancel-inner-⊑ S-leftId) \-cancel-inner
  \-cancel-outer-⊒ : {A B C : Obj} {S : Mor A C} {Q : Mor A B}
                  → isLeftIdentity (Q ⨾ Q ˘) → S ⊑ Q ⨾ (Q \ S)
  \-cancel-outer-⊒ {S = S} {Q} Q-leftId = ⊑-begin
        S
      ≈⟨ Q-leftId ⟨≈˘≈⟩ ⨾-assoc ⟩
        Q ⨾ Q ˘ ⨾ S
      ⊑⟨ ⨾-monotone₂ (\-universal (⊑-begin
          Q ⨾ Q ˘ ⨾ S
        ≈⟨ ⨾-assocL ⟨≈≈⟩ Q-leftId ⟩
          S
        □)) ⟩
        Q ⨾ (Q \ S)
      □
  \-cancel-outer-≈ : {A B C : Obj} {S : Mor A C} {Q : Mor A B}
                  → isLeftIdentity (Q ⨾ Q ˘) → Q ⨾ (Q \ S) ≈ S
  \-cancel-outer-≈ Q-leftId = ⊑-antisym \-cancel-outer (\-cancel-outer-⊒ Q-leftId)

  /-cancel-inner-⊑ : {A B C : Obj} {S : Mor A B} {T : Mor B C}
                  → isRightIdentity (T ⨾ T ˘) → (S ⨾ T) / T ⊑ S
  /-cancel-inner-⊑ {S = S} {T} T-rightId = ⊑-begin
        (S ⨾ T) / T
      ≈˘⟨ T-rightId ⟩
        ((S ⨾ T) / T) ⨾ T ⨾ T ˘
      ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ /-cancel-outer ⟩
        (S ⨾ T) ⨾ T ˘
      ≈⟨ ⨾-assoc ⟨≈≈⟩ T-rightId ⟩
        S
      □
  /-cancel-inner-≈ : {A B C : Obj} {T : Mor B C} {S : Mor A B}
                  → isRightIdentity (T ⨾ T ˘) → (S ⨾ T) / T ≈ S
  /-cancel-inner-≈ T-rightId = ⊑-antisym (/-cancel-inner-⊑ T-rightId) /-cancel-inner
  /-cancel-outer-⊒ : {A B C : Obj} {S : Mor A C} {R : Mor B C}
                  → isRightIdentity (R ˘ ⨾ R) → S ⊑ (S / R) ⨾ R
  /-cancel-outer-⊒ {S = S} {R} R-rightId = ⊑-begin
        S
      ≈⟨ R-rightId ⟨≈˘≈⟩ ⨾-assocL ⟩
        (S ⨾ R ˘) ⨾ R
      ⊑⟨ ⨾-monotone₁ (/-universal (⊑-begin
          (S ⨾ R ˘) ⨾ R
        ≈⟨ ⨾-assoc ⟨≈≈⟩ R-rightId ⟩
          S
        □)) ⟩
        (S / R) ⨾ R
      □
  /-cancel-outer-≈ : {A B C : Obj} {S : Mor A C} {R : Mor B C}
                  → isRightIdentity (R ˘ ⨾ R) → (S / R) ⨾ R ≈ S
  /-cancel-outer-≈ R-rightId = ⊑-antisym /-cancel-outer (/-cancel-outer-⊒ R-rightId)

module RightResOp-from-LeftResOp {i j k₁ k₂ : Level} {Obj : Set i}
  (base : OSGC j k₁ k₂ Obj)
  (leftResOp : LeftResOp (OSGC.orderedSemigroupoid base)) where
  open OSGC base
  open LeftResOp leftResOp
  rightResOp : RightResOp orderedSemigroupoid
```

```
rightResOp = record
  {_\_ = λ {A} {B} {C} Q S → (S ˘ / Q ˘) ˘
  ; \-cancel-outer = λ {_} {_} {_} {S} {Q} → ⊑-begin
      Q ⨾ (S ˘ / Q ˘) ˘
    ≈⟨ ≈-sym ˘-involutionRightConv ⟩
      ((S ˘ / Q ˘) ⨾ Q ˘) ˘
    ⊑⟨ ˘-monotone /-cancel-outer ⟩
      (S ˘) ˘
    ≈⟨ ˘˘ ⟩
      S
    □
  ; \-universal = λ {_} {_} {_} {S} {Q} {R} Q⨾R⊑S → ⊑-˘-swap (/-universal (⊑-begin
      R ˘ ⨾ Q ˘
    ≈⟨ ≈-sym ˘-involution ⟩
      (Q ⨾ R) ˘
    ⊑⟨ ˘-monotone Q⨾R⊑S ⟩
      S ˘
    □))
  }
```

13.4 Categoric.OSGD.RestrictedResiduals

Overview of derived laws for restricted left-residuals:

```
∤-universal′      : Q ⊑ S ∤ R → Q ⨾ R ⊑ S
∤-restr′          : Q ⊑ S ∤ R → ran Q ⊑ dom R
∤-cancel-inner    : ran T ⊑ dom S → T ⊑ (T ⨾ S) ∤ S
∤-monotone        : S₁ ⊑ S₂ → S₁ ∤ R ⊑ S₂ ∤ R
∤-cong₁           : S₁ ≈ S₂ → S₁ ∤ R ≈ S₂ ∤ R
∤-antitone        : R₂ ⊑ R₁ → dom R₁ ⊑ dom R₂ → S ∤ R₁ ⊑ S ∤ R₂
∤-cong₂           : R₁ ≈ R₂ → S ∤ R₁ ≈ S ∤ R₂
∤-cong            : S₁ ≈ S₂ → R₁ ≈ R₂ → S₁ ∤ R₁ ≈ S₂ ∤ R₂
∤-cancel-outer²   : (S ∤ R) ⨾ (R ∤ T) ⨾ T ⊑ S
∤-cancel-middle   : (S ∤ R) ⨾ (R ∤ T) ⊑ S ∤ T
∤-cancel-⨾        : ran (S ∤ R) ⊑ dom (R ⨾ T) → S ∤ R ⊑ (S ⨾ T) ∤ (R ⨾ T)
∤-outer-⨾         : F ⨾ (S ∤ R) ⊑ (F ⨾ S) ∤ R
dom-∤             : dom (S ∤ R) ⊑ dom S
domS⊑S∤S          : dom S ⊑ S ∤ S
domS∤S≈domS       : dom (S ∤ S) ≈ dom S
ranS∤S≈domS       : ran (S ∤ S) ≈ dom S
S∤S-⨾-S           : (S ∤ S) ⨾ S ≈ S
S∤S-isTransitive  : isTransitive (S ∤ S)
```

(The property ∤-cancel-middle has first been shown by Han (2008).)

```
record LeftRestrResOp {i j k₁ k₂ : Level} {Obj : Set i} (base : OSGDR j k₁ k₂ Obj) : Set (i ⊍ j ⊍ k₁ ⊍ k₂)
where
  open OSGDR base
  infixl 9 _∤_
  field
    _∤_ : {A B C : Obj} → Mor A C → Mor B C → Mor A B
    ∤-cancel-outer  :   {A B C : Obj} {S : Mor A C} {R : Mor B C}
                    →   (S ∤ R) ⨾ R ⊑ S
    ∤-restr         :   {A B C : Obj} {S : Mor A C} {R : Mor B C}
                    →   ran (S ∤ R) ⊑ dom R
    ∤-universal     :   {A B C : Obj} {S : Mor A C} {R : Mor B C} {Q : Mor A B}
                    →   Q ⨾ R ⊑ S → ran Q ⊑ dom R → Q ⊑ S ∤ R
```

```
∤-universal′          :   {A B C : Obj} {S : Mor A C} {R : Mor B C} {Q : Mor A B}
                          →   Q ⊑ S ∤ R → Q ⨾ R    ⊑ S
∤-universal′ Q⊑S∤R  =  ⨾-monotone₁ Q⊑S∤R ⟨⊑⊑⟩ ∤-cancel-outer
∤-restr’              :   {A B C : Obj} {S : Mor A C} {R : Mor B C} {Q : Mor A B}
                          →   Q ⊑ S ∤ R → ran Q ⊑ dom R
∤-restr’ Q⊑S∤R  =  ran-monotone Q⊑S∤R ⟨⊑⊑⟩ ∤-restr
∤-cancel-inner        :   {A B C : Obj} {T : Mor A B} {S : Mor B C}
                          →   ran T ⊑ dom S → T ⊑ (T ⨾ S) ∤ S
∤-cancel-inner ranT⊑domS  =  ∤-universal ⊑-refl ranT⊑domS
∤-monotone            :   {A B C : Obj} {S₁ S₂ : Mor A C} {R : Mor B C}
                          →   S₁ ⊑ S₂ → S₁ ∤ R ⊑ S₂ ∤ R
∤-monotone S₁⊑S₂  =   ∤-universal (∤-cancel-outer ⟨⊑⊑⟩ S₁⊑S₂) ∤-restr
∤-cong₁               :   {A B C : Obj} {S₁ S₂ : Mor A C} {R : Mor B C}
                          →   S₁ ≈ S₂ → S₁ ∤ R ≈ S₂ ∤ R
∤-cong₁ S₁≈S₂        =   ⊑-antisym (∤-monotone (⊑-reflexive S₁≈S₂))
                                      (∤-monotone (⊑-reflexive′ S₁≈S₂))
∤-antitone            :   {A B C : Obj} {S : Mor A C} {R₁ R₂ : Mor B C}
                          →   R₂ ⊑ R₁ → dom R₁ ⊑ dom R₂ → S ∤ R₁ ⊑ S ∤ R₂
∤-antitone R₂⊑R₁ domR₁⊑domR₂  =  ∤-universal (⨾-monotone₂ R₂⊑R₁ ⟨⊑⊑⟩ ∤-cancel-outer)
                                                   (∤-restr ⟨⊑⊑⟩ domR₁⊑domR₂)
∤-cong₂               :   {A B C : Obj} {S : Mor A C} {R₁ R₂ : Mor B C}
                          →   R₁ ≈ R₂ → S ∤ R₁ ≈ S ∤ R₂
∤-cong₂ R₁≈R₂        =   ⊑-antisym (∤-antitone (⊑-reflexive′ R₁≈R₂) (⊑-reflexive (dom-cong R₁≈R₂)))
                                      (∤-antitone (⊑-reflexive R₁≈R₂) (⊑-reflexive′ (dom-cong R₁≈R₂)))
∤-cong                :   {A B C : Obj} {S₁ S₂ : Mor A C} {R₁ R₂ : Mor B C}
                          →   S₁ ≈ S₂ → R₁ ≈ R₂ → S₁ ∤ R₁ ≈ S₂ ∤ R₂
∤-cong S₁≈S₂ R₁≈R₂  =  ∤-cong₁ S₁≈S₂ ⟨≈≈⟩ ∤-cong₂ R₁≈R₂
∤-cancel-outer²       :   {A B C D : Obj} {S : Mor A D} {R : Mor B D} {T : Mor C D}
                          →   (S ∤ R) ⨾ (R ∤ T) ⨾ T ⊑ S
∤-cancel-outer²       =   ⨾-monotone₂ ∤-cancel-outer ⟨⊑⊑⟩ ∤-cancel-outer
∤-cancel-middle       :   {A B C D : Obj} {S : Mor A D} {R : Mor B D} {T : Mor C D}
                          → (S ∤ R) ⨾ (R ∤ T) ⊑ S ∤ T
∤-cancel-middle       =   ∤-universal (⨾-assoc ⟨≈⊑⟩ ∤-cancel-outer²)
                                          (ranLocality’ ⟨⊑⊑⟩ ∤-restr)
∤-cancel-⨾            :   {A B C D : Obj} {S : Mor A C} {R : Mor B C} {T : Mor C D}
                          →   ran (S ∤ R) ⊑ dom (R ⨾ T) → S ∤ R ⊑ (S ⨾ T) ∤ (R ⨾ T)
∤-cancel-⨾ ran⊑dom  =   ∤-universal (⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ ∤-cancel-outer) ran⊑dom
∤-outer-⨾             :   {A B C D : Obj} {F : Mor A B} {S : Mor B D} {R : Mor C D}
                          →   F ⨾ (S ∤ R) ⊑ (F ⨾ S) ∤ R
∤-outer-⨾             =   ∤-universal (⨾-assoc ⟨≈⊑⟩ ⨾-monotone₂ ∤-cancel-outer)
                                          (ranLocality’ ⟨⊑⊑⟩ ∤-restr)
dom-∤                 :   {A B C : Obj} {S : Mor A C} {R : Mor B C}
                          →   dom (S ∤ R) ⊑ dom S
dom-∤ {A} {B} {C} {S} {R}  =  ⊑-begin
      dom (S ∤ R)
   ⊑⟨ dom-monotone ranPreserves ⟩
      dom ((S ∤ R) ⨾ ran (S ∤ R))
   ⊑⟨ dom-monotone (⨾-monotone₂ ∤-restr) ⟩
      dom ((S ∤ R) ⨾ dom R)
   ⊑⟨ domLocality ⟩
      dom ((S ∤ R) ⨾ R)
   ⊑⟨ dom-monotone ∤-cancel-outer ⟩
      dom S
   □
domS⊑S∤S             :   {A B : Obj} {S : Mor A B}
```

```
                     →    dom S ⊑ S ∤ S
  domS⊑S∤S          =    ∤-universal (proj₁ domSubIdentity) (⊑-reflexive ran-dom)
  domS∤S≈domS       :    {A B : Obj} {S : Mor A B}
                     →    dom (S ∤ S) ≈ dom S
  domS∤S≈domS       =    ⊑-antisym dom-∤
                          (dom-idempotent ⟨≈˘⊑⟩ dom-monotone domS⊑S∤S)
  ranS∤S≈domS       :    {A B : Obj} {S : Mor A B}
                     → ran (S ∤ S) ≈ dom S
  ranS∤S≈domS       =    ⊑-antisym ∤-restr (ran-dom ⟨≈˘⊑⟩ ran-monotone domS⊑S∤S)
  S∤S-⨾-S           :    {A B : Obj} {S : Mor A B}
                     →    (S ∤ S) ⨾ S ≈ S
  S∤S-⨾-S           =    ⊑-antisym ∤-cancel-outer (domPreserves ⟨⊑⊑⟩ ⨾-monotone₁ domS⊑S∤S)
  S∤S-isTransitive  :    {A B : Obj} {S : Mor A B}
                     →    isTransitive (S ∤ S)
  S∤S-isTransitive  =    ∤-cancel-middle


record RightRestrResOp {i j k₁ k₂ : Level} {Obj : Set i} (base : OSGDR j k₁ k₂ Obj) : Set (i ⊍ j ⊍ k₁ ⊍ k₂)
where
  open OSGDR base
  infixr 9 _⍀_
  field
    _⍀_ : {A B C : Obj} → Mor A B → Mor A C → Mor B C
    ⍀-cancel-outer    : {A B C : Obj} {S : Mor A C} {Q : Mor A B}
                      → Q ⨾ (Q ⍀ S)  ⊑ S
    ⍀-restr           : {A B C : Obj} {S : Mor A C} {Q : Mor A B}
                      → dom (Q ⍀ S) ⊑ ran Q
    ⍀-universal       : {A B C : Obj} {S : Mor A C} {Q : Mor A B} {R : Mor B C}
                      → Q ⨾ R ⊑ S → dom R ⊑ ran Q → R ⊑ Q ⍀ S
  ⍀-universal′        :  {A B C : Obj} {S : Mor A C} {Q : Mor A B} {R : Mor B C}
                      → R ⊑ Q ⍀ S → Q ⨾ R ⊑ S
  ⍀-universal′ R⊑Q⍀S  =  ⨾-monotone₂ R⊑Q⍀S ⟨⊑⊑⟩ ⍀-cancel-outer
  ⍀-restr’            :  {A B C : Obj} {S : Mor A C} {Q : Mor A B} {R : Mor B C}
                      → R ⊑ Q ⍀ S → dom R ⊑ ran Q
  ⍀-restr’ R⊑Q⍀S  =     dom-monotone R⊑Q⍀S ⟨⊑⊑⟩ ⍀-restr
  ⍀-cancel-inner      :  {A B C : Obj} {T : Mor B C} {S : Mor A B}
                      → dom T ⊑ ran S → T ⊑ S ⍀ (S ⨾ T)
  ⍀-cancel-inner domT⊑ranS  =  ⍀-universal ⊑-refl domT⊑ranS
  ⍀-monotone          :  {A B C : Obj} {S₁ S₂ : Mor A C} {Q : Mor A B}
                      → S₁ ⊑ S₂ → Q ⍀ S₁ ⊑ Q ⍀ S₂
  ⍀-monotone S₁⊑S₂  =  ⍀-universal (⍀-cancel-outer ⟨⊑⊑⟩ S₁⊑S₂) ⍀-restr
  ⍀-cong₂             :  {A B C : Obj} {S₁ S₂ : Mor A C} {Q : Mor A B}
                      → S₁ ≈ S₂ → Q ⍀ S₁ ≈ Q ⍀ S₂
  ⍀-cong₂ S₁≈S₂       = ⊑-antisym (⍀-monotone (⊑-reflexive S₁≈S₂))
                                  (⍀-monotone (⊑-reflexive′ S₁≈S₂))
  ⍀-antitone          :  {A B C : Obj} {S : Mor A C} {Q₁ Q₂ : Mor A B}
                      → Q₂ ⊑ Q₁ → ran Q₁ ⊑ ran Q₂ → Q₁ ⍀ S ⊑ Q₂ ⍀ S
  ⍀-antitone Q₂⊑Q₁ ranQ₁⊑ranQ₂  =  ⍀-universal (⨾-monotone₁ Q₂⊑Q₁ ⟨⊑⊑⟩ ⍀-cancel-outer)
                                              (⍀-restr ⟨⊑⊑⟩ ranQ₁⊑ranQ₂)
  ⍀-cong₁             :  {A B C : Obj} {S : Mor A C} {Q₁ Q₂ : Mor A B}
                      → Q₁ ≈ Q₂ → Q₁ ⍀ S ≈ Q₂ ⍀ S
  ⍀-cong₁ Q₁≈Q₂  =  ⊑-antisym (⍀-antitone (⊑-reflexive′ Q₁≈Q₂) (⊑-reflexive (ran-cong Q₁≈Q₂)))
                              (⍀-antitone (⊑-reflexive Q₁≈Q₂) (⊑-reflexive′ (ran-cong Q₁≈Q₂)))
  ⍀-cong              :  {A B C : Obj} {S₁ S₂ : Mor A C} {Q₁ Q₂ : Mor A B}
                      → Q₁ ≈ Q₂ → S₁ ≈ S₂ → Q₁ ⍀ S₁ ≈ Q₂ ⍀ S₂
  ⍀-cong Q₁≈Q₂ S₁≈S₂  =  ⍀-cong₂ S₁≈S₂ ⟨≈≈⟩ ⍀-cong₁ Q₁≈Q₂
```

```
  ⍀-cancel-outer²     :  {A B C D : Obj} {S : Mor A D} {Q : Mor A C} {T : Mor A B}
                         → T ⨾ (T ⍀ Q) ⨾ (Q ⍀ S) ⊑ S
  ⍀-cancel-outer²     =  ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ ⍀-cancel-outer ⟨⊑⊑⟩ ⍀-cancel-outer
  ⍀-cancel-middle     : {A B C D : Obj} {S : Mor A D} {Q : Mor A C} {T : Mor A B}
                         → (T ⍀ Q) ⨾ (Q ⍀ S) ⊑ T ⍀ S
  ⍀-cancel-middle     =  ⍀-universal ⍀-cancel-outer² (domLocality' ⟨⊑⊑⟩ ⍀-restr)
  ⍀-cancel-⨾          :  {A B C D : Obj} {S : Mor B D} {Q : Mor B C} {T : Mor A B}
                         → dom (Q ⍀ S) ⊑ ran (T ⨾ Q) → Q ⍀ S ⊑ (T ⨾ Q) ⍀ (T ⨾ S)
  ⍀-cancel-⨾ dom⊑ran  =  ⍀-universal (⨾-assoc ⟨≈⊑⟩ ⨾-monotone₂ ⍀-cancel-outer) dom⊑ran
  ⍀-outer-⨾           :  {A B C D : Obj} {F : Mor C D} {S : Mor A C} {Q : Mor A B}
                         → (Q ⍀ S) ⨾ F ⊑ Q ⍀ (S ⨾ F)
  ⍀-outer-⨾           =  ⍀-universal     (⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ ⍀-cancel-outer)
                                            (domLocality' ⟨⊑⊑⟩ ⍀-restr)
  ran-⍀               :  {A B C : Obj} {S : Mor A C} {Q : Mor A B}
                         → ran (Q ⍀ S) ⊑ ran S
  ran-⍀ {A} {B} {C} {S} {Q}  =  ⊑-begin
        ran (Q ⍀ S)
     ⊑⟨ ran-monotone domPreserves ⟩
        ran (dom (Q ⍀ S) ⨾ (Q ⍀ S))
     ⊑⟨ ran-monotone (⨾-monotone₁ ⍀-restr) ⟩
        ran (ran Q ⨾ (Q ⍀ S))
     ⊑⟨ ranLocality ⟩
        ran (Q ⨾ (Q ⍀ S))
     ⊑⟨ ran-monotone ⍀-cancel-outer ⟩
        ran S
     □
  ranS⊑S⍀S            :  {A B : Obj} {S : Mor A B} → ran S ⊑ S ⍀ S
  ranS⊑S⍀S            =  ⍀-universal (proj₂ ranSubIdentity) (⊑-reflexive dom-ran)
  ranS⍀S≈ranS         :  {A B : Obj} {S : Mor A B} → ran (S ⍀ S) ≈ ran S
  ranS⍀S≈ranS         =  ⊑-antisym ran-⍀ (ran-idempotent ⟨≈˘⊑⟩ ran-monotone ranS⊑S⍀S)
  domS⍀S≈ranS         :  {A B : Obj} {S : Mor A B} → dom (S ⍀ S) ≈ ran S
  domS⍀S≈ranS         =  ⊑-antisym ⍀-restr (dom-ran ⟨≈˘⊑⟩ dom-monotone ranS⊑S⍀S)
  S-⨾-S⍀S             :  {A B : Obj} {S : Mor A B} → S ⨾ (S ⍀ S) ≈ S
  S-⨾-S⍀S             =  ⊑-antisym ⍀-cancel-outer (ranPreserves ⟨⊑⊑⟩ ⨾-monotone₂ ranS⊑S⍀S)
  S⍀S-isTransitive    :  {A B : Obj} {S : Mor A B}
                         → isTransitive (S ⍀ S)
  S⍀S-isTransitive    =  ⍀-cancel-middle
```

```
module RestrResidualOps {i j k₁ k₂ : Level} {Obj : Set i} (base : OSGDR j k₁ k₂ Obj)
  (leftRestrResOp : LeftRestrResOp     base)
  (rightRestrResOp : RightRestrResOp base) where
  open OSGDR base
  open LeftRestrResOp   leftRestrResOp   public
  open RightRestrResOp rightRestrResOp public
  ⌿-twist :     {A B C D : Obj} {S : Mor A C} {R : Mor B C} {T : Mor D C}
             → dom (S ⌿ R) ⊑ ran (T ⌿ S) → S ⌿ R ⊑ (T ⌿ S) ⍀ (T ⌿ R)
  ⌿-twist =     ⍀-universal ⌿-cancel-middle
  ⌿-twist-down :     {A B C : Obj} {S : Mor A C} {R : Mor B C}
                   → dom (S ⌿ R) ⊑ ran (R ⌿ S) → S ⌿ R ⊑ (R ⌿ S) ⍀ (R ⌿ R)
  ⌿-twist-down =     ⍀-universal ⌿-cancel-middle
  ⌿-twist-up : {A B C : Obj} {S : Mor A C} {R : Mor B C} → S ⌿ R ⊑ (S ⌿ S) ⍀ (S ⌿ R)
  ⌿-twist-up = ⌿-twist (dom-⌿ ⟨⊑≈˘⟩ ranS⌿S≈domS)
```

13.5 Categoric.OSGD.Residuals

```
module OSGDR-LeftRes-Props {i j k₁ k₂ : Level} {Obj : Set i}
                           (base : OSGDR j k₁ k₂ Obj)
                           (leftResOp : LeftResOp (OSGDR.orderedSemigroupoid base))
where
  open OSGDR base
  open LeftResOp leftResOp

  leftRestrResOp : LeftRestrResOp base
  leftRestrResOp = record
    {_⧸•_ = λ {A} {B} {C} S R → (S / R) ⨾ dom R
    ; ⧸•-cancel-outer = λ {A} {B} {C} {S} {R} → ⊑-begin
        ((S / R) ⨾ dom R) ⨾ R
      ≈⟨ ⨾-assoc ⟩
        (S / R) ⨾ dom R ⨾ R
      ⊑⟨ ⨾-monotone₂ (proj₁ domSubIdentity) ⟩
        (S / R) ⨾ R
      ⊑⟨ /-cancel-outer ⟩
        S
      □
    ; ⧸•-restr = λ {A} {B} {C} {S} {R} → ⊑-begin
        ran ((S / R) ⨾ dom R)
      ⊑⟨ ranLocality' ⟩
        ran (dom R)
      ≈⟨ ran-dom ⟩
        dom R
      □
    ; ⧸•-universal = λ {A} {B} {C} {S} {R} {Q} Q⨾R⊑S ranQ⊑domR → ⊑-begin
        Q
      ⊑⟨ ranPreserves ⟩
        Q ⨾ ran Q
      ⊑⟨ ⨾-monotone (/-universal Q⨾R⊑S) ranQ⊑domR ⟩
        (S / R) ⨾ dom R
      □
    }

module OSGDR-RightRes-Props {i j k₁ k₂ : Level} {Obj : Set i}
  (base : OSGDR j k₁ k₂ Obj)
  (rightResOp : RightResOp (OSGDR.orderedSemigroupoid base))
where
  open OSGDR base
  open RightResOp rightResOp

  rightRestrResOp : RightRestrResOp base
  rightRestrResOp = record
    {_⧹•_ = λ {A} {B} {C} Q S → ran Q ⨾ (Q \ S)
    ; ⧹•-cancel-outer = λ {A} {B} {C} {S} {Q} → ⊑-begin
        Q ⨾ ran Q ⨾ (Q \ S)
      ≈⟨ ⨾-assocL ⟩
        (Q ⨾ ran Q) ⨾ (Q \ S)
      ⊑⟨ ⨾-monotone₁ (proj₂ ranSubIdentity) ⟩
        Q ⨾ (Q \ S)
      ⊑⟨ \-cancel-outer ⟩
        S
      □
    ; ⧹•-restr = λ {A} {B} {C} {S} {Q} → ⊑-begin
        dom (ran Q ⨾ (Q \ S))
      ⊑⟨ domLocality' ⟩
        dom (ran Q)
```

```
      ≈⟨ dom-ran ⟩
        ran Q
      □
    ; ⍀-universal  =  λ {A} {B} {C} {S} {Q} {R} Q⨾R⊑S domR⊑ranQ → ⊑-begin
        R
      ⊑⟨ domPreserves ⟩
        dom R ⨾ R
      ⊑⟨ ⨾-monotone domR⊑ranQ (\-universal Q⨾R⊑S) ⟩
        ran Q ⨾ (Q \ S)
      □
    }
```

13.6 Categoric.OSGC.SyQ

For background on **symmetric quotients**, see the papers by Berghammer et al. (1986, 1989), Zierer (1991), Schmidt and Ströhlein (1993, Sect. 4.4) and Furusawa and Kahl (1998).

```
record SyqOp {i j k₁ k₂ : Level} {Obj : Set i}
  (base : OSGC j k₁ k₂ Obj)
  : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  open OSGC base
  infixr 9 _╳_
  field
    _╳_ : {A B C : Obj} → Mor A B → Mor A C → Mor B C
    ╳-cong          :   {A B C : Obj} {Q₁ Q₂ : Mor A B} {S₁ S₂ : Mor A C}
                      → Q₁ ≈ Q₂ → S₁ ≈ S₂ → Q₁ ╳ S₁ ≈ Q₂ ╳ S₂
    ╳-cancel-left   :   {A B C : Obj} {Q : Mor A B} {S : Mor A C} → Q ⨾ (Q ╳ S)  ⊑ S
    ╳-cancel-right  :   {A B C : Obj} {Q : Mor A B} {S : Mor A C} → (Q ╳ S) ⨾ S ˘ ⊑ Q ˘
    ╳-universal     :   {A B C : Obj} {Q : Mor A B} {S : Mor A C} {R : Mor B C}
                      → Q ⨾ R ⊑ S → R ⨾ S ˘ ⊑ Q ˘ → R ⊑ Q ╳ S
  ╳-cong₁ : {A B C : Obj} {Q₁ Q₂ : Mor A B} {S : Mor A C} → Q₁ ≈ Q₂ → Q₁ ╳ S ≈ Q₂ ╳ S
  ╳-cong₁ Q₁≈Q₂  =  ╳-cong Q₁≈Q₂ ≈-refl
  ╳-cong₂ : {A B C : Obj} {Q : Mor A B} {S₁ S₂ : Mor A C} → S₁ ≈ S₂ → Q ╳ S₁ ≈ Q ╳ S₂
  ╳-cong₂  =  ╳-cong ≈-refl
  ╳-universal-right       :   {A B C : Obj} {Q : Mor A B} {S : Mor A C}
                            → {R : Mor B C} → R ⊑ Q ╳ S → Q ⨾ R ⊑ S
  ╳-universal-right R⊑Q╳S  =  ⨾-monotone₂ R⊑Q╳S ⟨⊑⊑⟩ ╳-cancel-left
  ╳-universal-left        :   {A B C : Obj} {Q : Mor A B} {S : Mor A C}
                            → {R : Mor B C} → R ⊑ Q ╳ S → R ⨾ S ˘ ⊑ Q ˘
  ╳-universal-left R⊑Q╳S  =  ⨾-monotone₁ R⊑Q╳S ⟨⊑⊑⟩ ╳-cancel-right
  ╳-universal-left˘       :   {A B C : Obj} {Q : Mor A B} {S : Mor A C} {R : Mor B C}
                            → R ⊑ Q ╳ S → S ⨾ R ˘ ⊑ Q
  ╳-universal-left˘ R⊑Q╳S  =  ≈-sym ˘-involutionRightConv ⟨≈⊑⟩ ˘-⊑-swap (╳-universal-left R⊑Q╳S)
  ╳-˘-⊑ : {A B C : Obj} {Q : Mor A B} {S : Mor A C} → (Q ╳ S) ˘ ⊑ S ╳ Q
  ╳-˘-⊑ {A} {B} {C} {Q} {S}  =  ╳-universal
    (⊑-begin
        S ⨾ (Q ╳ S) ˘
      ≈⟨ ≈-sym ˘-involutionRightConv ⟩
        ((Q ╳ S) ⨾ S ˘) ˘
      ⊑⟨ ˘-monotone ╳-cancel-right ⟩
        Q ˘ ˘
      ≈⟨ ˘˘ ⟩
        Q
    □)
    (⊑-begin
```

```
        (Q ╳ S) ˘ ⨾ Q ˘
      ≈⟨ ≈-sym ˘-involution ⟩
        (Q ⨾ (Q ╳ S)) ˘
      ⊑⟨ ˘-monotone ╳-cancel-left ⟩
        S ˘
    □)
  ╳-˘ : {A B C : Obj} {Q : Mor A B} {S : Mor A C} → (Q ╳ S) ˘ ≈ S ╳ Q
  ╳-˘ = ⊑-antisym ╳-˘-⊑ (≈-sym ˘˘ ⟨≈⊑⟩ ˘-monotone ╳-˘-⊑)
  ╳-cancel-innner :    {A B C Z : Obj} {Q : Mor A B} {S : Mor A C} {P : Mor Z A}
                     → Q ╳ S ⊑ (P ⨾ Q) ╳ (P ⨾ S)
  ╳-cancel-innner {_} {_} {_} {_} {Q} {S} {P} = ╳-universal
    (⨾-assoc ⟨≈⊑⟩ ⨾-monotone₂ ╳-cancel-left)
    (⊑-begin
        (Q ╳ S) ⨾ (P ⨾ S) ˘
      ≈⟨ ⨾-cong₂ ˘-involution ⟩
        (Q ╳ S) ⨾ S ˘ ⨾ P ˘
      ≈⟨ ⨾-assocL ⟩
        ((Q ╳ S) ⨾ S ˘) ⨾ P ˘
      ⊑⟨ ⨾-monotone₁ ╳-cancel-right ⟩
        Q ˘ ⨾ P ˘
      ≈⟨ ≈-sym ˘-involution ⟩
        (P ⨾ Q) ˘
    □)
  ╳-cancel-middle :    {A B C D : Obj} {Q : Mor A B} {S : Mor A C} {T : Mor A D}
                     → (Q ╳ S) ⨾ (S ╳ T) ⊑ Q ╳ T
  ╳-cancel-middle {_} {_} {_} {_} {Q} {S} {T} = ╳-universal
    (⊑-begin
        Q ⨾ (Q ╳ S) ⨾ (S ╳ T)
      ≈⟨ ⨾-assocL ⟩
        (Q ⨾ (Q ╳ S)) ⨾ (S ╳ T)
      ⊑⟨ ⨾-monotone₁ ╳-cancel-left ⟩
        S ⨾ (S ╳ T)
      ⊑⟨ ╳-cancel-left ⟩
        T
    □)
    (⊑-begin
        ((Q ╳ S) ⨾ (S ╳ T)) ⨾ T ˘
      ≈⟨ ⨾-assoc ⟩
        (Q ╳ S) ⨾ (S ╳ T) ⨾ T ˘
      ⊑⟨ ⨾-monotone₂ ╳-cancel-right ⟩
        (Q ╳ S) ⨾ S ˘
      ⊑⟨ ╳-cancel-right ⟩
        Q ˘
    □)
  ╳-isDifunctional : {A B C : Obj} {Q : Mor A B} {S : Mor A C} → isDifunctional (Q ╳ S)
  ╳-isDifunctional {A} {B} {C} {Q} {S} = ⊑-begin
        (Q ╳ S) ⨾ (Q ╳ S) ˘ ⨾ (Q ╳ S)
      ≈⟨ ⨾-cong₂₁ ╳-˘ ⟩
        (Q ╳ S) ⨾ (S ╳ Q) ⨾ (Q ╳ S)
      ⊑⟨ ⨾-monotone₂ ╳-cancel-middle ⟩
        (Q ╳ S) ⨾ (S ╳ S)
      ⊑⟨ ╳-cancel-middle ⟩
        Q ╳ S
    □
  ╳-surjective-cancel-left    :    {A B C : Obj} {Q : Mor A B} {S : Mor A C}
                                 → isSurjective (Q ╳ S) → Q ⨾ (Q ╳ S) ≈ S
  ╳-surjective-cancel-left {_} {_} {_} {Q} {S} isSurj = ⊑-antisym ╳-cancel-left
    (⊑-begin
```

```
          S
        ⊑⟨ proj₂ isSurj ⟩
          S ⨾ (Q ≬ S) ˘ ⨾ (Q ≬ S)
        ≈⟨ ⨾-cong₂₁ ≬-˘ ⟩
          S ⨾ (S ≬ Q) ⨾ (Q ≬ S)
        ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ ≬-cancel-left ⟩
          Q ⨾ (Q ≬ S)
      □)
  ≬-total-cancel-right      :    {A B C : Obj} {Q : Mor A B} {S : Mor A C}
                              → isTotal (Q ≬ S) → (Q ≬ S) ⨾ S ˘ ≈ Q ˘
  ≬-total-cancel-right {_} {_} {_} {Q} {S} isTot = ⊑-antisym ≬-cancel-right
     (⊑-begin
          Q ˘
        ⊑⟨ proj₁ isTot ⟨⊑≈⟩ ⨾-assoc ⟩
          (Q ≬ S) ⨾ (Q ≬ S) ˘ ⨾ Q ˘
        ≈⟨ ⨾-cong₂₁ ≬-˘ ⟩
          (Q ≬ S) ⨾ (S ≬ Q) ⨾ Q ˘
        ⊑⟨ ⨾-monotone₂ ≬-cancel-right ⟩
          (Q ≬ S) ⨾ S ˘
      □)
  ≬-total-cancel-middle      :    {A B C D : Obj} {Q : Mor A B} {S : Mor A C} {T : Mor A D}
                              → isTotal (Q ≬ S) → (Q ≬ S) ⨾ (S ≬ T) ≈ Q ≬ T
  ≬-total-cancel-middle {_} {_} {_} {_} {Q} {S} {T} isTot = ⊑-antisym ≬-cancel-middle
     (⊑-begin
          Q ≬ T
        ⊑⟨ proj₁ isTot ⟨⊑≈⟩ ⨾-assoc ⟩
          (Q ≬ S) ⨾ (Q ≬ S) ˘ ⨾ (Q ≬ T)
        ≈⟨ ⨾-cong₂₁ ≬-˘ ⟩
          (Q ≬ S) ⨾ (S ≬ Q) ⨾ (Q ≬ T)
        ⊑⟨ ⨾-monotone₂ ≬-cancel-middle ⟩
          (Q ≬ S) ⨾ (S ≬ T)
      □)
  ≬-surjective-cancel-middle :    {A B C D : Obj} {Q : Mor A B} {S : Mor A C} {T : Mor A D}
                              → isSurjective (S ≬ T) → (Q ≬ S) ⨾ (S ≬ T) ≈ Q ≬ T
  ≬-surjective-cancel-middle {_} {_} {_} {_} {Q} {S} {T} isSurj = ⊑-antisym ≬-cancel-middle
     (⊑-begin
          Q ≬ T
        ⊑⟨ proj₂ isSurj ⟩
          (Q ≬ T) ⨾ (S ≬ T) ˘ ⨾ (S ≬ T)
        ≈⟨ ⨾-cong₂₁ ≬-˘ ⟩
          (Q ≬ T) ⨾ (T ≬ S) ⨾ (S ≬ T)
        ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ ≬-cancel-middle ⟩
          (Q ≬ S) ⨾ (S ≬ T)
      □)


≬-iso-shift-left : {A B C D : Obj} {Q : Mor A C} {S : Mor B D} {T : Mor A B}
                  → isBijective T → isMapping T → Q ≬ (T ⨾ S) ≈ (T ˘ ⨾ Q) ≬ S
≬-iso-shift-left {A} {B} {C} {D} {Q} {S} {T} isBij isMap = let
      idPair : isIdentity (T ⨾ T ˘) × isIdentity (T ˘ ⨾ T)   -- pattern binding impossible?
      idPair = bijMapping-identities isBij isMap
      isIdA : isIdentity (T ⨾ T ˘)
      isIdA = proj₁ idPair
      isIdB : isIdentity (T ˘ ⨾ T)
      isIdB = proj₂ idPair
   in ⊑-antisym (≬-cancel-innner ⟨⊑≈⟩ ≬-cong₂ (⨾-assocL ⟨≈≈⟩ proj₁ isIdB))
                (≬-cancel-innner ⟨⊑≈⟩ ≬-cong₁ (⨾-assocL ⟨≈≈⟩ proj₁ isIdA))
≬-iso-shift-right   :    {A B C D : Obj} {Q : Mor A C} {S : Mor B D} {T : Mor B A}
                      → isBijective T → isMapping T → (T ⨾ Q) ≬ S ≈ Q ≬ (T ˘ ⨾ S)
```

```
╳-iso-shift-right isBij isMap = ╳-cong₁ (⨾-cong₁ (≈-sym ˘˘))
  ⟨≈≈⟩ ≈-sym (╳-iso-shift-left (˘-isBijective isMap) (˘-isMapping isBij))
╳-unival-in-left :    {A B C D : Obj} {Q : Mor C B} {S : Mor C D} {F : Mor A B}
                      → isUnivalent F → (F ⨾ (Q ╳ S)) ⊑ ((Q ⨾ F ˘) ╳ S)
╳-unival-in-left {_} {_} {_} {_} {Q} {S} {F} isUnival = ╳-universal
  (⊑-begin
      (Q ⨾ F ˘) ⨾ F ⨾ (Q ╳ S)
    ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⨾-assocL ⟩
      Q ⨾ (F ˘ ⨾ F) ⨾ (Q ╳ S)
    ⊑⟨ ⨾-monotone₂ (proj₁ isUnival) ⟩
      Q ⨾ (Q ╳ S)
    ⊑⟨ ╳-cancel-left ⟩
      S
  □)
  (⊑-begin
      (F ⨾ (Q ╳ S)) ⨾ S ˘
    ⊑⟨ ⨾-assoc ⟨≈⊑⟩ ⨾-monotone₂ ╳-cancel-right ⟩
      F ⨾ Q ˘
    ≈⟨ ≈-sym ˘-involutionRightConv ⟩
      (Q ⨾ F ˘) ˘
  □)
╳-inj-in-right   :    {A B C D : Obj} {Q : Mor A B} {S : Mor A C} {F : Mor C D}
                      → isInjective F → ((Q ╳ S) ⨾ F) ⊑ (Q ╳ (S ⨾ F))
╳-inj-in-right {_} {_} {_} {_} {Q} {S} {F} isInj = ╳-universal
  (⊑-begin
      Q ⨾ (Q ╳ S) ⨾ F
    ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ ╳-cancel-left ⟩
      S ⨾ F
  □)
  (⊑-begin
      ((Q ╳ S) ⨾ F) ⨾ (S ⨾ F) ˘
    ≈⟨ ⨾-cong₂ ˘-involution ⟨≈≈⟩ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⨾-assocL ⟩
      (Q ╳ S) ⨾ (F ⨾ F ˘) ⨾ S ˘
    ⊑⟨ ⨾-monotone₂ (proj₁ isInj) ⟩
      (Q ╳ S) ⨾ S ˘
    ⊑⟨ ╳-cancel-right ⟩
      Q ˘
  □)
noy-⊒-subidentity : {A B : Obj} {Q : Mor A B} {p : Mor B B} → isSubidentity p → p ⊑ (Q ╳ Q)
noy-⊒-subidentity {A} {B} {Q} {p} (left, right) = ╳-universal right left
noy-isSubidentity : {A B : Obj} {Q : Mor A B} → isUnivalent Q → isSurjective Q → isSubidentity (Q ╳ Q)
noy-isSubidentity {A} {B} {Q} isUnival isSurj = ⊑-isSubidentity
  (⊑-begin
      Q ╳ Q
    ⊑⟨ proj₁ isSurj ⟨⊑≈⟩ ⨾-assoc ⟩
      Q ˘ ⨾ Q ⨾ (Q ╳ Q)
    ⊑⟨ ⨾-monotone₂ ╳-cancel-left ⟩
      Q ˘ ⨾ Q
  □)
  isUnival
symTrans╳ : {A : Obj} {Q : Mor A A} → isSymmetric Q → isTransitive Q → Q ⊑ Q ˘ ╳ Q
symTrans╳ {A} {Q} isSym isTrans = ╳-universal
  (⊑-begin
      Q ˘ ⨾ Q
    ≈⟨ ⨾-cong₁ isSym ⟩
      Q ⨾ Q
    ⊑⟨ isTrans ⟩
      Q
```

```
  □)
  (⊑-begin
      Q ⨾ Q ˘
    ≈⟨ ⨾-cong₂ isSym ⟩
      Q ⨾ Q
    ⊑⟨ isTrans ⟩
      Q
    ≈⟨ ≈-sym ˘˘ ⟩
      Q ˘ ˘
  □)
╳symTrans :   {A : Obj} {Q : Mor A A}
              → isCodifunctional Q → Q ⊑ Q ˘ ╳ Q → isSymmetric Q × isTransitive Q
╳symTrans {_} {Q} isCodifun Q⊑Q˘╳Q = let
    left : Q ˘ ⨾ Q ⊑ Q
    left = ╳-universal-right Q⊑Q˘╳Q
    left˘ : Q ˘ ⨾ Q ⊑ Q ˘
    left˘ = ≈-sym ˘-involutionLeftConv ⟨≈⊑⟩ ˘-monotone left
    right : Q ⨾ Q ˘ ⊑ Q
    right = ╳-universal-left Q⊑Q˘╳Q ⟨⊑≈⟩ ˘˘
    right˘ : Q ⨾ Q ˘ ⊑ Q ˘
    right˘ = ≈-sym ˘-involutionRightConv ⟨≈⊑⟩ ˘-monotone right
    sym˘ : Q ⊑ Q ˘
    sym˘ = ⊑-begin
             Q
      ⊑⟨ isCodifun ⟩
             Q ⨾ Q ˘ ⨾ Q
      ⊑⟨ ⨾-monotone₂ left˘ ⟩
             Q ⨾ Q ˘
      ⊑⟨ right˘ ⟩
             Q ˘
      □
    sym : Q ˘ ⊑ Q
    sym = ˘-⊑-swap sym˘
    trans = ⊑-begin
             Q ⨾ Q
      ⊑⟨ ⨾-monotone₂ sym˘ ⟩
             Q ⨾ Q ˘
      ⊑⟨ right ⟩
             Q
      □
  in isSymmetric⊑ sym, trans
inj-╳-inj : {A B C : Obj} {Q : Mor A B} {S : Mor A C} → isInjective Q → isInjective S → Q ˘ ⨾ S ⊑ Q ╳ S
inj-╳-inj {A} {B} {C} {Q} {S} isInjQ isInjS = ╳-universal
  (⊑-begin
      Q ⨾ Q ˘ ⨾ S
    ⊑⟨ ⨾-assocL  ⟨≈⊑⟩ proj₁ isInjQ ⟩
      S
  □)
  (⊑-begin
      (Q ˘ ⨾ S) ⨾ S ˘
    ⊑⟨ ⨾-assoc ⟨≈⊑⟩ proj₂ isInjS ⟩
      Q ˘
  □)


retractSyqOp : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
             → (F : Obj₂ → Obj₁)
             → {base : OSGC j k₁ k₂ Obj₁}
             → SyqOp base → SyqOp (retractOSGC F base)
```

```
retractSyqOp F syqOp = let open SyqOp syqOp in record
  {_╳_ = _╳_
  ; ╳-cong = ╳-cong
  ; ╳-cancel-left = ╳-cancel-left
  ; ╳-cancel-right = ╳-cancel-right
  ; ╳-universal = ╳-universal
  }
```

13.7 Categoric.SemiAllegory.Residuals

In a semi-allegory, the original definition of symmetric quotients satisfies the axiomatisation we gave in Sect. 13.6:

```
module SyqOp-from-ResOps {i j k₁ k₂ : Level} {Obj : Set i} (base : SemiAllegory j k₁ k₂ Obj)
    (leftResOp  : LeftResOp  (SemiAllegory.orderedSemigroupoid base))
    (rightResOp : RightResOp (SemiAllegory.orderedSemigroupoid base)) where
  open SemiAllegory base
  open LeftResOp  leftResOp
  open RightResOp rightResOp

  syqOp : SyqOp osgc
  syqOp = record
    {_╳_ = λ Q S → Q \ S ⊓ Q ˘ / S ˘
    ; ╳-cong = λ Q₁≈Q₂ S₁≈S₂ → ⊓-cong (\-cong Q₁≈Q₂ S₁≈S₂)
      (/-cong (˘-cong Q₁≈Q₂) (˘-cong S₁≈S₂))
    ; ╳-cancel-left = λ {_} {_} {_} {Q} {S} → ⊑-begin
          Q ⨾ (Q \ S ⊓ Q ˘ / S ˘)
        ⊑⟨ ⨾-monotone₂ ⊓-lower₁ ⟩
          Q ⨾ (Q \ S)
        ⊑⟨ \-cancel-outer ⟩
          S
      □
    ; ╳-cancel-right = λ {_} {_} {_} {Q} {S} → ⊑-begin
          (Q \ S ⊓ Q ˘ / S ˘) ⨾ S ˘
        ⊑⟨ ⨾-monotone₁ ⊓-lower₂ ⟩
          (Q ˘ / S ˘) ⨾ S ˘
        ⊑⟨ /-cancel-outer ⟩
          Q ˘
      □
    ; ╳-universal = λ {_} {_} {_} {Q} {S} {R} Q⨾R⊑S R⨾S˘⊑Q˘ → ⊑-begin
          R
        ≈⟨ ≈-sym ⊓-idempotent ⟩
          R ⊓ R
        ⊑⟨ ⊓-monotone (\-universal Q⨾R⊑S) (/-universal R⨾S˘⊑Q˘) ⟩
          (Q \ S ⊓ Q ˘ / S ˘)
      □
    }
```

In addition, this is the only possible symmetric quotient operation:

```
  module _ (syqOp₀ : SyqOp osgc) where
    open SyqOp syqOp₀
    syqOp-unique : {A B C : Obj} {Q : Mor A B} {S : Mor A C}
                 → Q ╳ S ≈ Q \ S ⊓ Q ˘ / S ˘
    syqOp-unique {A} {B} {C} {Q} {S} = ⊑-antisym
      (⊓-universal (\-universal ╳-cancel-left) (/-universal ╳-cancel-right))
      (╳-universal
        (⨾-monotone₂ ⊓-lower₁ ⟨⊑⊑⟩ \-cancel-outer)
```

```
    (⨾-monotone₁ ⊓-lower₂ ⟨⊑⊑⟩ /-cancel-outer)
  )
```

13.8 Categoric.DivSemiAllegory

For division semi-allegories, even though right residuals, restricted residuals, and symmetric quotients all can be derived from left residuals, we still assume them all as primitive here, since this produces more readable goals, and also makes connecting to optimised implementations easier.

[WK: *The restricted residual operators remain derived for the time being, because they cannot in general be retracted.* **]**

```
record DivSemiAllegory {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field distrSemiAllegory : DistrSemiAllegory j k₁ k₂ Obj
  field leftResOp         : LeftResOp (DistrSemiAllegory.orderedSemigroupoid distrSemiAllegory)
  field rightResOp        : RightResOp (DistrSemiAllegory.orderedSemigroupoid distrSemiAllegory)
  field syqOp             : SyqOp (DistrSemiAllegory.osgc distrSemiAllegory)
  open DistrSemiAllegory          distrSemiAllegory                  public
  open ResidualOps                leftResOp rightResOp               public
  open OSGC-Residuals        osgc leftResOp rightResOp               public
  open SyqOp                      syqOp                              public
  open OSGDR-LeftRes-Props   osgdr leftResOp                         public
  open OSGDR-RightRes-Props  osgdr rightResOp                        public
  open RestrResidualOps      osgdr leftRestrResOp rightRestrResOp    public
```

13.9 Categoric.DivAllegory

As in division semi-allegories (Categoric.DivSemiAllegory (Sect. 13.8)), we assume all different kinds of residuals in separate **field**s:

[WK: *The restricted residual operators remain derived for the time being, because they cannot in general be retracted.* **]**

```
record DivAllegory {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field distrAllegory : DistrAllegory j k₁ k₂ Obj
  field leftResOp     : LeftResOp (DistrAllegory.orderedSemigroupoid distrAllegory)
  field rightResOp    : RightResOp (DistrAllegory.orderedSemigroupoid distrAllegory)
  field syqOp         : SyqOp (DistrAllegory.osgc distrAllegory)
  open DistrAllegory distrAllegory public
  divSemiAllegory : DivSemiAllegory j k₁ k₂ Obj
  divSemiAllegory = record
    {distrSemiAllegory = distrSemiAllegory
    ;leftResOp = leftResOp
    ;rightResOp = rightResOp
    ;syqOp = syqOp
    }
  open ResidualOps                                 leftResOp rightResOp  public
  open OSGC-Residuals          osgc                leftResOp rightResOp  public
  open SyqOp                                       syqOp                 public
  open OrdCat-Residual-Props   orderedCategory     leftResOp rightResOp  public
  open OSGDR-LeftRes-Props     osgdr               leftResOp             public
  open OSGDR-RightRes-Props    osgdr               rightResOp            public
  open RestrResidualOps        osgdr leftRestrResOp rightRestrResOp      public
```

```
retractDivAllegory : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
  →                   (F : Obj₂ → Obj₁) → DivAllegory j k₁ k₂ Obj₁ → DivAllegory j k₁ k₂ Obj₂
retractDivAllegory F base = let open DivAllegory base in record
  {distrAllegory = retractDistrAllegory F distrAllegory
  ; leftResOp = retractLeftResOp F leftResOp
  ; rightResOp = retractRightResOp F rightResOp
  ; syqOp = retractSyqOp F syqOp
  }
```

Chapter 14

Iteration

We define *Kleene categories* in Sect. 14.1 as a typed version of Kleene algebras, following essentially the axiomatization of Kozen (1994a). We also define a semigroupoid version (Sect. 14.1), where it is natural to consider transitive closure $\_^+$ instead of reflexive-transitive closure, or Kleene star, $\_^*$. In sections 14.2 and 14.4, we add converse to the respective iteration operators.

We then proceed to define *action lattice categories* as a typed version of the action lattices of Kozen (1994b). Interestingly, Kozen does not even mention possible distributivity (nor lack thereof) of the lattice component of action lattices. Kozen proposed action lattices as an extension of the action algebras introducted by Pratt (1991), which are essentially Kleene algebras with residuals of compositions, where the Kleene algebra induction axioms have been replaced with equational axioms relating residuals and iteration:

$$(R \mathbin{/} R)^* = R \mathbin{/} R \qquad \text{and} \qquad (R \setminus R)^* = R \setminus R \quad .$$

For the time being, we do not include separate theories of action algebra semigroupoids and categories.

Instead, we first introduce *action lattice semigroupoids* for exploration (Sect. 14.6), then proceed to *action lattice cetegories* (Sect. 14.7), and finally jump immediately to *distributive action allegories* (Sect. 14.8), which are action lattice categories that are also division allegories.

14.1 Categoric.KleeneSemigroupoid

Kleene semigroupoids, understood essentially as "Kleene categories without identities", are essentially a heterogeneous version of the "1-free Kleene algebras" of Kozen (1998), except that we also omit the zero morphisms.

```
record TransClosOp {i j k₁ k₂ : Level} {Obj : Set i}
                   (base : USLSemigroupoid j k₁ k₂ Obj)
                   : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  open USLSemigroupoid base
  field
    _⁺ : {A : Obj} → Mor A A → Mor A A   -- This is \^+
    ⁺-recDef₁ : {A : Obj} {R : Mor A A} → R ⁺ ≈ R ⊔ R ⨾ R ⁺
    ⁺-recDef₂ : {A : Obj} {R : Mor A A} → R ⁺ ≈ R ⊔ R ⁺ ⨾ R
    ⁺-leftInd : {A : Obj} {R : Mor A A} {B : Obj} {S : Mor A B} → R ⨾ S ⊑ S → R ⁺ ⨾ S ⊑ S
    ⁺-rightInd : {A : Obj} {R : Mor A A} {B : Obj} {Q : Mor B A} → Q ⨾ R ⊑ Q → Q ⨾ R ⁺ ⊑ Q
```

We chose to include ⁺-leftInd and ⁺-rightInd in the axioms here, which are Kozen's axioms (11) and (12). We now show that these imply his alternative axioms (9) (⁺-leftInd′) and (10) (⁺-rightInd′); the opposite implication is shown below in mkTransClosOp′.

```
⁺-leftInd′ : {A : Obj} {R : Mor A A} {B : Obj} {P S : Mor A B} → P ⊔ R ⨾ S ⊑ S → P ⊔ R ⁺ ⨾ P ⊑ S
⁺-leftInd′ {_} {R} {_} {P} {S} P⊔R⨾S⊑S = let
      P⊑S : P ⊑ S
```

```
      P⊑S  =  ⊑-trans ⊔-upper₁ P⊔R⨾S⊑S
      R⨾S⊑S  :  R ⨾ S ⊑ S
      R⨾S⊑S  =  ⊑-trans ⊔-upper₂ P⊔R⨾S⊑S
    in ⊔-universal P⊑S (⊑-begin
          R ⁺ ⨾ P
        ⊑⟨ ⨾-monotone₂ P⊑S ⟩
          R ⁺ ⨾ S
        ⊑⟨ ⁺-leftInd R⨾S⊑S ⟩
          S
        □)
⁺-rightInd′ : {A : Obj} {R : Mor A A} {B : Obj} {P Q : Mor B A} → P ⊔ Q ⨾ R ⊑ Q → P ⊔ P ⨾ R ⁺ ⊑ Q
⁺-rightInd′ {_} {R} {_} {P} {Q} P⊔Q⨾R⊑Q  =  let
      P⊑Q  :  P ⊑ Q
      P⊑Q  =  ⊑-trans ⊔-upper₁ P⊔Q⨾R⊑Q
      Q⨾R⊑Q  :  Q ⨾ R ⊑ Q
      Q⨾R⊑Q  =  ⊑-trans ⊔-upper₂ P⊔Q⨾R⊑Q
    in ⊔-universal P⊑Q (⊑-begin
          P ⨾ R ⁺
        ⊑⟨ ⨾-monotone₁ P⊑Q ⟩
          Q ⨾ R ⁺
        ⊑⟨ ⁺-rightInd Q⨾R⊑Q ⟩
          Q
        □)

⁺-increases : {A : Obj} {R : Mor A A} → R ⊑ R ⁺
⁺-increases {A} {R}  =  ⊑-begin
        R
    ⊑⟨  ⊔-upper₁ ⟩
        R ⊔ R ⨾ R ⁺
    ≈⟨ ≈-sym ⁺-recDef₁ ⟩
        R ⁺
    □
⁺-isTransitive :    {A : Obj} {R : Mor A A} → R ⁺ ⨾ R ⁺ ⊑ R ⁺
⁺-isTransitive {A} {R}  =  ⁺-leftInd (⊑-trans₁ ⊔-upper₂ (≈-sym ⁺-recDef₁))

⁺-monotone : {A : Obj} {R S : Mor A A} → R ⊑ S → R ⁺ ⊑ S ⁺
⁺-monotone {A} {R} {S} R⊑S  =  ⊑-begin
        R ⁺
    ≈⟨ ⁺-recDef₂ ⟩
        R ⊔ R ⁺ ⨾ R
    ⊑⟨ ⁺-leftInd′ R⊔R⨾S⁺⊑S⁺ ⟩
        S ⁺
    □
  where
    R⊔R⨾S⁺⊑S⁺  :  R ⊔ R ⨾ S ⁺ ⊑ S ⁺
    R⊔R⨾S⁺⊑S⁺  =  ⊑-begin
          R ⊔ R ⨾ S ⁺
      ⊑⟨ ⊔-monotone R⊑S (⨾-monotone₁ R⊑S) ⟩
          S ⊔ S ⨾ S ⁺
      ≈⟨ ≈-sym ⁺-recDef₁ ⟩
          S ⁺
      □
⁺-cong : {A : Obj} {R S : Mor A A} → R ≈ S → R ⁺ ≈ S ⁺
⁺-cong R≈S  =  ⊑-antisym (⁺-monotone (⊑-reflexive R≈S)) (⁺-monotone (⊑-reflexive′ R≈S))

R⨾R⁺⊑R⁺  :  {A : Obj} {R : Mor A A} → R ⨾ R ⁺ ⊑ R ⁺
R⨾R⁺⊑R⁺  =  ⊑-trans₁ ⊔-upper₂ (≈-sym ⁺-recDef₁)
```

```
R⁺⨾R⊑R⁺ : {A : Obj} {R : Mor A A} → R ⁺ ⨾ R ⊑ R ⁺
R⁺⨾R⊑R⁺ = ⊑-trans₁ ⊔-upper₂ (≈-sym ⁺-recDef₂)

R⁺⨾R≈R⨾R⁺ : {A : Obj} {R : Mor A A} → R ⁺ ⨾ R ≈ R ⨾ R ⁺
R⁺⨾R≈R⨾R⁺ {_} {R} = ≈-begin
      R ⁺ ⨾ R
    ≈⟨ ⨾-cong₁ ⁺-recDef₁ ⟩
      (R ⊔ R ⨾ R ⁺) ⨾ R
    ≈⟨ ⨾-⊔-distribL ⟨≈≈⟩ ⊔-cong₂ ⨾-assoc ⟩
      R ⨾ R ⊔ R ⨾ R ⁺ ⨾ R
    ≈⟨ ≈-sym ⨾-⊔-distribR ⟩
      R ⨾ (R ⊔ R ⁺ ⨾ R)
    ≈⟨ ⨾-cong₂ (≈-sym ⁺-recDef₂) ⟩
      R ⨾ R ⁺
    □

⁺⨾⁺₁ :  {A : Obj} {R : Mor A A} → R ⁺ ⨾ R ⁺ ≈ R ⨾ R ⁺
⁺⨾⁺₁ {A} {R} =
  ⊑-antisym (⊑-begin
      R ⁺ ⨾ R ⁺
   ≈⟨ ⨾-cong₁ ⁺-recDef₁ ⟩
      (R ⊔ R ⨾ R ⁺) ⨾ R ⁺
   ≈⟨ ⨾-⊔-distribL ⟩
      R ⨾ R ⁺ ⊔ (R ⨾ R ⁺) ⨾ R ⁺
   ⊑⟨ ⊔-monotone₂ (⨾-assoc ⟨≈⊑⟩ ⨾-monotone₂ ⁺-isTransitive) ⟩
      R ⨾ R ⁺ ⊔ R ⨾ R ⁺
   ≈⟨ ⊔-idempotent ⟩
      R ⨾ R ⁺
   □)
   (⨾-monotone₁ ⁺-increases)

⁺⨾⁺₂ :  {A : Obj} {R : Mor A A} → R ⁺ ⨾ R ⁺ ≈ R ⁺ ⨾ R
⁺⨾⁺₂ {A} {R} =
  ⊑-antisym (⊑-begin
      R ⁺ ⨾ R ⁺
   ≈⟨ ⨾-cong₂ ⁺-recDef₂ ⟩
      R ⁺ ⨾ (R ⊔ R ⁺ ⨾ R)
   ≈⟨ ⨾-⊔-distribR ⟩
      R ⁺ ⨾ R ⊔ R ⁺ ⨾ R ⁺ ⨾ R
   ⊑⟨ ⊔-monotone₂ (⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ ⁺-isTransitive) ⟩
      R ⁺ ⨾ R ⊔ R ⁺ ⨾ R
   ≈⟨ ⊔-idempotent ⟩
      R ⁺ ⨾ R
   □)
   (⨾-monotone₂ ⁺-increases)

⁺-recDef : {A : Obj} {R : Mor A A} → R ⁺ ≈ R ⊔ R ⁺ ⨾ R ⁺
⁺-recDef {A} {R} = ≈-begin
      R ⁺
    ≈⟨ ⁺-recDef₁ ⟩
      R ⊔ R ⨾ R ⁺
    ≈⟨ ⊔-cong₂ (≈-sym ⁺⨾⁺₁) ⟩
      R ⊔ R ⁺ ⨾ R ⁺ □

⁺-isTC :  {A : Obj} → {R S : Mor A A} → R ⊑ S → S ⨾ S ⊑ S → R ⁺ ⊑ S
⁺-isTC {A} {R} {S} R⊑S S⨾S⊑S = ⊑-begin
```

```
        R ⁺
      ≈⟨ ⁺-recDef₁ ⟩
        R ⊔ R ⨾ R ⁺
      ⊑⟨ ⊔-monotone R⊑S (⨾-monotone₁ R⊑S) ⟩
        S ⊔ S ⨾ R ⁺
      ⊑⟨ ⊔-monotone₂ (⁺-rightInd (⨾-monotone₂ R⊑S ⟨⊑⊑⟩ S⨾S⊑S)) ⟩
        S ⊔ S
      ≈⟨ ⊔-idempotent ⟩
        S
      □


⁺⁺ :  {A : Obj} {R : Mor A A} → (R ⁺) ⁺ ≈ R ⁺
⁺⁺ {A} {R} = ⊑-antisym (⁺-isTC ⊑-refl ⁺-isTransitive) ⁺-increases


⁺-⨾-roll⊑ : {A B : Obj} {R : Mor A B} {S : Mor B A} → (R ⨾ S) ⁺ ⨾ R ⊑ R ⨾ (S ⨾ R) ⁺
⁺-⨾-roll⊑ {_} {_} {R} {S} = ⊑-begin
      (R ⨾ S) ⁺ ⨾ R
    ≈⟨ ⨾-cong₁ ⁺-recDef₂ ⟩
      ((R ⨾ S) ⊔ (R ⨾ S) ⁺ ⨾ (R ⨾ S)) ⨾ R
    ⊑⟨ ⨾-⊔-subdistribL ⟩
      (R ⨾ S) ⨾ R ⊔ ((R ⨾ S) ⁺ ⨾ (R ⨾ S)) ⨾ R
    ≈⟨ ⊔-cong ⨾-assoc (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⨾-assoc) ⟩
      R ⨾ S ⨾ R ⊔ (R ⨾ S) ⁺ ⨾ R ⨾ S ⨾ R
    ⊑⟨ ⊔-universal (⨾-monotone₂ ⁺-increases) (⊑-begin
            (R ⨾ S) ⁺ ⨾ R ⨾ S ⨾ R
        ⊑⟨ ⨾-monotone₂₂ ⁺-increases ⟩
            (R ⨾ S) ⁺ ⨾ R ⨾ (S ⨾ R) ⁺
        ⊑⟨ ⁺-leftInd (⊑-begin
              (R ⨾ S) ⨾ R ⨾ (S ⨾ R) ⁺
          ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⨾-assocL ⟩
              R ⨾ (S ⨾ R) ⨾ (S ⨾ R) ⁺
          ⊑⟨ ⨾-monotone₂ R⨾R⁺⊑R⁺ ⟩
              R ⨾ (S ⨾ R) ⁺
          □) ⟩
            R ⨾ (S ⨾ R) ⁺
        □) ⟩
      R ⨾ (S ⨾ R) ⁺
    □


⨾-⁺-roll⊑ : {A B : Obj} {R : Mor A B} {S : Mor B A} → R ⨾ (S ⨾ R) ⁺ ⊑ (R ⨾ S) ⁺ ⨾ R
⨾-⁺-roll⊑ {_} {_} {R} {S} = ⊑-begin
      R ⨾ (S ⨾ R) ⁺
    ≈⟨ ⨾-cong₂ ⁺-recDef₁ ⟩
      R ⨾ ((S ⨾ R) ⊔ (S ⨾ R) ⨾ (S ⨾ R) ⁺)
    ⊑⟨ ⨾-⊔-subdistribR ⟩
      R ⨾ (S ⨾ R) ⊔ R ⨾ (S ⨾ R) ⨾ (S ⨾ R) ⁺
    ≈⟨ ⊔-cong ⨾-assocL (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ ⨾-assocL) ⟩
      (R ⨾ S) ⨾ R ⊔ ((R ⨾ S) ⨾ R) ⨾ (S ⨾ R) ⁺
    ⊑⟨ ⊔-universal (⨾-monotone₁ ⁺-increases) (⊑-begin
            ((R ⨾ S) ⨾ R) ⨾ (S ⨾ R) ⁺
        ⊑⟨ ⨾-monotone₁₁ ⁺-increases ⟩
            ((R ⨾ S) ⁺ ⨾ R) ⨾ (S ⨾ R) ⁺
        ⊑⟨ ⁺-rightInd (⊑-begin
              ((R ⨾ S) ⁺ ⨾ R) ⨾ (S ⨾ R)
          ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⨾-assocL ⟨≈≈⟩ ⨾-assocL ⟩
              ((R ⨾ S) ⁺ ⨾ (R ⨾ S)) ⨾ R
          ⊑⟨ ⨾-monotone₁ R⁺⨾R⊑R⁺ ⟩
```

```
                 (R ⨾ S) ⁺ ⨾ R
               □) ⟩
                 (R ⨾ S) ⁺ ⨾ R
             □) ⟩
           (R ⨾ S) ⁺ ⨾ R
         □

⁺-⨾-roll : {A B : Obj} {R : Mor A B} {S : Mor B A} → (R ⨾ S) ⁺ ⨾ R ≈ R ⨾ (S ⨾ R) ⁺
⁺-⨾-roll = ⊑-antisym ⁺-⨾-roll⊑ ⨾-⁺-roll⊑
```

To show that Kozen's (9) and (10) imply his (11) and (12), we define:

```
mkTransClosOp′ : {i j k₁ k₂ : Level} {Obj : Set i}
                 (base : USLSemigroupoid j k₁ k₂ Obj)
               → let open USLSemigroupoid base in
                 (_⁺ : {A : Obj} → Mor A A → Mor A A)
                 (⁺-recDef₁ : {A : Obj} {R : Mor A A} → R ⁺ ≈ R ⊔ R ⨾ R ⁺)
                 (⁺-recDef₂ : {A : Obj} {R : Mor A A} → R ⁺ ≈ R ⊔ R ⁺ ⨾ R)
                 (⁺-leftInd′  : {A : Obj} {R : Mor A A} {B : Obj} {P S : Mor A B}
                              → P ⊔ R ⨾ S ⊑ S → P ⊔ R ⁺ ⨾ P ⊑ S)
                 (⁺-rightInd′ : {A : Obj} {R : Mor A A} {B : Obj} {P Q : Mor B A}
                              → P ⊔ Q ⨾ R ⊑ Q → P ⊔ P ⨾ R ⁺ ⊑ Q)
               → TransClosOp base
mkTransClosOp′ {Obj = Obj} base _⁺ ⁺-recDef₁ ⁺-recDef₂ ⁺-leftInd′ ⁺-rightInd′ = let
  open USLSemigroupoid base
  in record
  {_⁺ = _⁺
  ;⁺-recDef₁ = ⁺-recDef₁
  ;⁺-recDef₂ = ⁺-recDef₂
  ;⁺-leftInd = λ {A} {R} {B} {S} R⨾S⊑S → let
     S⊔R⨾S⊑S : S ⊔ R ⨾ S ⊑ S
     S⊔R⨾S⊑S = ⊔-universal ⊑-refl R⨾S⊑S
     in ⊔-upper₂   ⟨⊑⊑⟩ ⁺-leftInd′ S⊔R⨾S⊑S
  ;⁺-rightInd = λ {_} {R} {_} {Q} Q⨾R⊑Q → let
     Q⊔Q⨾R⊑Q : Q ⊔ Q ⨾ R ⊑ Q
     Q⊔Q⨾R⊑Q = ⊔-universal ⊑-refl Q⨾R⊑Q
     in ⊔-upper₂   ⟨⊑⊑⟩ ⁺-rightInd′ Q⊔Q⨾R⊑Q
  }


record KleeneSemigroupoid {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field uslSemigroupoid : USLSemigroupoid j k₁ k₂ Obj
  open USLSemigroupoid uslSemigroupoid
  field transClosOp : TransClosOp uslSemigroupoid
  open USLSemigroupoid uslSemigroupoid public
  open TransClosOp      transClosOp      public


retractTransClosOp : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                   → (F : Obj₂ → Obj₁)
                   → {base : USLSemigroupoid j k₁ k₂ Obj₁}
                   → TransClosOp base → TransClosOp (retractUSLSemigroupoid F base)
retractTransClosOp F transClosOp = let open TransClosOp transClosOp in record
  {_⁺ = _⁺
  ;⁺-recDef₁    = ⁺-recDef₁
  ;⁺-recDef₂    = ⁺-recDef₂
  ;⁺-leftInd    = ⁺-leftInd
  ;⁺-rightInd   = ⁺-rightInd
  }
```

```
retractKleeneSemigroupoid : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                          → (F : Obj₂ → Obj₁)
                          → KleeneSemigroupoid j k₁ k₂ Obj₁ → KleeneSemigroupoid j k₁ k₂ Obj₂
retractKleeneSemigroupoid F base = let open KleeneSemigroupoid base in record
  {uslSemigroupoid = retractUSLSemigroupoid F uslSemigroupoid
  ; transClosOp = retractTransClosOp F transClosOp
  }
```

14.2 Categoric.KSGC

We will add converse to Kleene semigroupoids by adding iteration to upper semi-lattice semigroupoids with converse. The resulting properties are defined in a separate module, for ease of obtaining them in encompassing structures:

```
module KSGC-Props {i j k₁ k₂ : Level} {Obj : Set i}
                  (uslsgc : USLSGC j k₁ k₂ Obj)
                  (transClosOp : TransClosOp (USLSGC.uslSemigroupoid uslsgc))
  where
  open USLSGC      uslsgc
  open TransClosOp transClosOp

  ˘-⁺-subcomm : {A : Obj} → {R : Mor A A} → (R ˘) ⁺ ⊑ (R ⁺) ˘
  ˘-⁺-subcomm {A} {R} = ⁺-isTC
      (˘-monotone (⁺-increases))
      (⊑-begin
        (R ⁺) ˘ ⨾ (R ⁺) ˘
      ≈⟨ ≈-sym ˘-involution ⟩
        (R ⁺ ⨾ R ⁺) ˘
      ⊑⟨ ˘-monotone ⁺-isTransitive ⟩
        (R ⁺) ˘
    □)
  ˘-⁺ : {A : Obj} → {R : Mor A A} → (R ⁺) ˘ ≈ (R ˘) ⁺
  ˘-⁺ {A} {R} = ⊑-antisym
    (˘-⊑-swap (⊑-begin
        R ⁺
      ≈⟨ ⁺-cong (≈-sym ˘˘) ⟩
          ((R ˘) ˘) ⁺
      ⊑⟨ ˘-⁺-subcomm ⟩
        ((R ˘) ⁺) ˘
      □)
    )
    ˘-⁺-subcomm

  ⁺-isSymmetric : {A : Obj} → {R : Mor A A} → isSymmetric R → isSymmetric (R ⁺)
  ⁺-isSymmetric {A} {R} R-isSym = ≈-begin
        (R ⁺) ˘
      ≈⟨ ˘-⁺ ⟩
        (R ˘) ⁺
      ≈⟨ ⁺-cong R-isSym ⟩
        R ⁺
      □
```

In a KSGC, box star morphisms (difunctional closures) exist and can be defined using transitive closure:

```
  boxStar :     {A B : Obj} → Mor A B → Mor A B
  boxStar R = R ⊔ (R ⨾ R ˘) ⁺ ⨾ R
```

```
boxStar-2 : {A B : Obj} → {R : Mor A B} → boxStar R ⨾ boxStar R ˘ ⊑ (R ⨾ R ˘) ⁺
boxStar-2 {R = R} = ⊑-begin
      boxStar R ⨾ boxStar R ˘
    ≈⟨ ≈-refl ⟩
      (R ⊔ (R ⨾ R ˘) ⁺ ⨾ R) ⨾ (R ⊔ (R ⨾ R ˘) ⁺ ⨾ R) ˘
    ≈⟨ ⨾-cong₂ ˘-⊔-distrib ⟩
      (R ⊔ (R ⨾ R ˘) ⁺ ⨾ R) ⨾ (R ˘ ⊔ ((R ⨾ R ˘) ⁺ ⨾ R) ˘)
    ⊑⟨ ⨾-⊔-subdistribR ⟩
      (R ⊔ (R ⨾ R ˘) ⁺ ⨾ R) ⨾ R ˘ ⊔ (R ⊔ (R ⨾ R ˘) ⁺ ⨾ R) ⨾ ((R ⨾ R ˘) ⁺ ⨾ R) ˘
    ⊑⟨ ⊔-universal
        (⊑-begin
            (R ⊔ (R ⨾ R ˘) ⁺ ⨾ R) ⨾ R ˘
          ⊑⟨ ⨾-⊔-subdistribL ⟩
            R ⨾ R ˘ ⊔ ((R ⨾ R ˘) ⁺ ⨾ R) ⨾ R ˘
          ⊑⟨ ⊔-universal ⁺-increases (⊑-begin
              ((R ⨾ R ˘) ⁺ ⨾ R) ⨾ R ˘
            ≈⟨ ⨾-assoc ⟩
              (R ⨾ R ˘) ⁺ ⨾ R ⨾ R ˘
            ⊑⟨ R⁺⨾R⊑R⁺ ⟩
              (R ⨾ R ˘) ⁺
            □) ⟩
            (R ⨾ R ˘) ⁺
        □) (⊑-begin
            (R ⊔ (R ⨾ R ˘) ⁺ ⨾ R) ⨾ ((R ⨾ R ˘) ⁺ ⨾ R) ˘
          ≈⟨ ⨾-cong₂ ˘-involution ⟩
            (R ⊔ (R ⨾ R ˘) ⁺ ⨾ R) ⨾ R ˘ ⨾ ((R ⨾ R ˘) ⁺) ˘
          ≈⟨ ⨾-cong₂₂ (˘-⁺ ⟨≈≈⟩ ⁺-cong ˘-involutionRightConv) ⟩
            (R ⊔ (R ⨾ R ˘) ⁺ ⨾ R) ⨾ R ˘ ⨾ (R ⨾ R ˘) ⁺
          ⊑⟨ ⨾-⊔-subdistribL ⟩
            R ⨾ R ˘ ⨾ (R ⨾ R ˘) ⁺ ⊔ ((R ⨾ R ˘) ⁺ ⨾ R) ⨾ R ˘ ⨾ (R ⨾ R ˘) ⁺
          ⊑⟨ ⊔-universal (⊑-begin
              R ⨾ R ˘ ⨾ (R ⨾ R ˘) ⁺
            ⊑⟨ ⨾-assocL ⟨≈⊑⟩ R⨾R⁺⊑R⁺ ⟩
              (R ⨾ R ˘) ⁺
            □) (⊑-begin
              ((R ⨾ R ˘) ⁺ ⨾ R) ⨾ R ˘ ⨾ (R ⨾ R ˘) ⁺
            ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⨾-assocL ⟩
              (R ⨾ R ˘) ⁺ ⨾ (R ⨾ R ˘) ⨾ (R ⨾ R ˘) ⁺
            ⊑⟨ ⨾-monotone₂ R⨾R⁺⊑R⁺ ⟩
              (R ⨾ R ˘) ⁺ ⨾ (R ⨾ R ˘) ⁺
            ⊑⟨ ⁺-isTransitive ⟩
              (R ⨾ R ˘) ⁺
            □) ⟩
            (R ⨾ R ˘) ⁺
        □) ⟩
      (R ⨾ R ˘) ⁺
  □

boxStarProof : {A B : Obj} (R : Mor A B) → boxStar R isBoxStarOf R
boxStarProof R = let Rb = boxStar R in record
  {isBoxStar-incl = ⊑-begin
        R
      ⊑⟨ ⊔-upper₁ ⟩
        R ⊔ (R ⨾ R ˘) ⁺ ⨾ R
    □
  ; isBoxStar-difun = ⊑-begin
        boxStar R ⨾ boxStar R ˘ ⨾ boxStar R
      ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ boxStar-2 ⟩
        (R ⨾ R ˘) ⁺ ⨾ boxStar R
```

```
        ≈⟨ ≈-refl ⟩
          (R ⨾ R ˘) ⁺ ⨾ (R ⊔ (R ⨾ R ˘) ⁺ ⨾ R)
        ⊑⟨ ⨾-⊔-subdistribR ⟩
          (R ⨾ R ˘) ⁺ ⨾ R ⊔ (R ⨾ R ˘) ⁺ ⨾ (R ⨾ R ˘) ⁺ ⨾ R
        ⊑⟨ ⊔-universal ⊑-refl (⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ ⁺-isTransitive) ⟩
          (R ⨾ R ˘) ⁺ ⨾ R
        ⊑⟨ ⊔-upper₂ ⟩
          R ⊔ (R ⨾ R ˘) ⁺ ⨾ R
        ≈⟨ ≈-refl ⟩
          boxStar R
      □
    ; isBoxStar-leftInd = λ {C} {P} {Q} P⨾R⊑Q Q⨾R˘⨾R⊑Q → ⊑-begin
          P ⨾ boxStar R
        ≈⟨ ≈-refl ⟩
          P ⨾ (R ⊔ (R ⨾ R ˘) ⁺ ⨾ R)
        ⊑⟨ ⨾-⊔-subdistribR ⟩
          P ⨾ R ⊔ P ⨾ (R ⨾ R ˘) ⁺ ⨾ R
        ≈⟨ ⊔-cong₂ (⨾-cong₂ ⁺-⨾-roll) ⟩
          P ⨾ R ⊔ P ⨾ R ⨾ (R ˘ ⨾ R) ⁺
        ⊑⟨ ⊔-monotone₂ (⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ P⨾R⊑Q) ⟩
          P ⨾ R ⊔ Q ⨾ (R ˘ ⨾ R) ⁺
        ⊑⟨ ⊔-universal P⨾R⊑Q (⁺-rightInd Q⨾R˘⨾R⊑Q) ⟩
          Q
      □
    ; isBoxStar-rightInd = λ {C} {P} {Q} R⨾P⊑Q R⨾R˘⨾Q⊑Q → ⊑-begin
          boxStar R ⨾ P
        ≈⟨ ≈-refl ⟩
          (R ⊔ (R ⨾ R ˘) ⁺ ⨾ R) ⨾ P
        ⊑⟨ ⨾-⊔-subdistribL ⟩
          R ⨾ P ⊔ ((R ⨾ R ˘) ⁺ ⨾ R) ⨾ P
        ⊑⟨ ⊔-monotone₂ (⨾-assoc ⟨≈⊑⟩ ⨾-monotone₂ R⨾P⊑Q) ⟩
          R ⨾ P ⊔ (R ⨾ R ˘) ⁺ ⨾ Q
        ⊑⟨ ⊔-universal R⨾P⊑Q (⁺-leftInd (⨾-assoc ⟨≈⊑⟩ R⨾R˘⨾Q⊑Q)) ⟩
          Q
      □
    }

record KSGC {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field uslsgc : USLSGC j k₁ k₂ Obj
        transClosOp : TransClosOp (USLSGC.uslSemigroupoid uslsgc)
  open USLSGC      uslsgc                 public
  open TransClosOp transClosOp            public
  open KSGC-Props  uslsgc transClosOp     public


retractKSGC : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
             → (F : Obj₂ → Obj₁) → KSGC j k₁ k₂ Obj₁ → KSGC j k₁ k₂ Obj₂
retractKSGC F base = let open KSGC base in record
  {uslsgc = retractUSLSGC F uslsgc
  ; transClosOp = retractTransClosOp F transClosOp
  }
```

14.3 Categoric.KleeneCategory

Kleene categories are essentially the typed Kleene algebras of Kozen (1998). We first define the Kleene star operator in the context of an upper semi-lattice category.

```
module InUSLCat {i j k₁ k₂ : Level} {Obj : Set i}
                (base : USLCategory j k₁ k₂ Obj) where
  open USLCategory base
  open FinColimits compOp using (IsInitial; IsInitial≈)

record IsStar {A : Obj} (R R* : Mor A A) : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  field
    *-recDef : R* ≈ Id ⊔ R ⊔ R* ⨾ R*
    *-leftInd : {B : Obj} {S : Mor A B} → R ⨾ S ⊑ S → R* ⨾ S ⊑ S
    *-rightInd : {B : Obj} {Q : Mor B A} → Q ⨾ R ⊑ Q → Q ⨾ R* ⊑ Q

  *-leftInd′ : {B : Obj} {P S : Mor A B} → P ⊔ R ⨾ S ⊑ S → R* ⨾ P ⊑ S
  *-leftInd′ {_} {P} {S} P⊔R⨾S⊑S = let
    P⊑S : P ⊑ S
    P⊑S = ⊑-trans ⊔-upper₁ P⊔R⨾S⊑S
    R⨾S⊑S : R ⨾ S ⊑ S
    R⨾S⊑S = ⊑-trans ⊔-upper₂ P⊔R⨾S⊑S
    in ⊑-begin
            R* ⨾ P
          ⊑⟨ ⨾-monotone₂ P⊑S ⟩
            R* ⨾ S
          ⊑⟨ *-leftInd R⨾S⊑S ⟩
            S
      □
  *-rightInd′ : {B : Obj} {P Q : Mor B A} → P ⊔ Q ⨾ R ⊑ Q → P ⨾ R* ⊑ Q
  *-rightInd′ {_} {P} {Q} P⊔Q⨾R⊑Q = let
    P⊑Q : P ⊑ Q
    P⊑Q = ⊑-trans ⊔-upper₁ P⊔Q⨾R⊑Q
    Q⨾R⊑Q : Q ⨾ R ⊑ Q
    Q⨾R⊑Q = ⊑-trans ⊔-upper₂ P⊔Q⨾R⊑Q
    in ⊑-begin
            P ⨾ R*
          ⊑⟨ ⨾-monotone₁ P⊑Q ⟩
            Q ⨾ R*
          ⊑⟨ *-rightInd Q⨾R⊑Q ⟩
            Q
      □

  *-increases : R ⊑ R*
  *-increases = ⊑-begin
          R
      ⊑⟨  ⊔-upper₁ ⟩
          R ⊔ R* ⨾ R*
      ⊑⟨  ⊔-upper₂ ⟩
          Id ⊔ R ⊔ R* ⨾ R*
      ≈⟨ ≈-sym *-recDef ⟩
          R*
    □

  *-isReflexive : Id ⊑ R*
  *-isReflexive = ⊑-begin
          Id
      ⊑⟨  ⊔-upper₁ ⟩
          Id ⊔ R ⊔ R* ⨾ R*
      ≈⟨ ≈-sym *-recDef ⟩
          R*
```

```
  □
*-isSuperidentity : isSuperidentity (R*)
*-isSuperidentity = reflexiveIsSuperidentity *-isReflexive


*-recDef₁⊑ : Id ⊔ R ⨾ R* ⊑ R*
*-recDef₁⊑ = ⊑-begin
      Id ⊔ R ⨾ R*
    ⊑⟨ ⊔-monotone₂ (⨾-monotone₁ *-increases ⟨⊑⊑⟩ ⊔-upper₂) ⟩
      Id ⊔ R ⊔ R* ⨾ R*
    ≈⟨ ≈-sym *-recDef ⟩
      R*
  □
*-recDef₁ : R* ≈ Id ⊔ R ⨾ R*
*-recDef₁ = ⊑-antisym
  (⊑-begin
      R*
    ≈⟨ ≈-sym rightId ⟩
      R* ⨾ Id
    ⊑⟨ *-leftInd′ (⊑-begin
                          Id ⊔ R ⨾ (Id ⊔ R ⨾ R*)
                        ⊑⟨ ⊔-monotone₂ (⨾-monotone₂ *-recDef₁⊑) ⟩
                          Id ⊔ R ⨾ R*
                        □) ⟩
      Id ⊔ R ⨾ R*
  □)
  *-recDef₁⊑


*-recDef₂⊑ : Id ⊔ R* ⨾ R ⊑ R*
*-recDef₂⊑ = ⊑-begin
      Id ⊔ R* ⨾ R
    ⊑⟨ ⊔-monotone₂ (⨾-monotone₂ *-increases ⟨⊑⊑⟩ ⊔-upper₂) ⟩
      Id ⊔ R ⊔ R* ⨾ R*
    ≈⟨ ≈-sym *-recDef ⟩
      R*
  □
*-recDef₂ : R* ≈ Id ⊔ R* ⨾ R
*-recDef₂ = ⊑-antisym
  (⊑-begin
      R*
    ≈⟨ ≈-sym leftId ⟩
      Id ⨾ R*
    ⊑⟨ *-rightInd′ (⊑-begin
                          Id ⊔ (Id ⊔ R* ⨾ R) ⨾ R
                        ⊑⟨ ⊔-monotone₂ (⨾-monotone₁ *-recDef₂⊑) ⟩
                          Id ⊔ R* ⨾ R
                        □) ⟩
      Id ⊔ R* ⨾ R
  □)
  *-recDef₂⊑


*-stepL : R ⨾ R* ⊑ R*
*-stepL = ⊑-begin
        R ⨾ R*
    ⊑⟨  ⨾-monotone₁ *-increases ⟩
        R* ⨾ R*
    ⊑⟨  ⊔-upper₂ ⟩
        R ⊔ R* ⨾ R*
```

```
      ⊑⟨   ⊔-upper₂ ⟩
           Id ⊔ R ⊔ R* ⨾ R*
      ≈⟨ ≈-sym *-recDef ⟩
           R*
    □

  *-stepR : R* ⨾ R ⊑ R*
  *-stepR = ⊑-begin
           R* ⨾ R
      ⊑⟨   ⨾-monotone₂ *-increases ⟩
           R* ⨾ R*
      ⊑⟨   ⊔-upper₂ ⟩
           R ⊔ R* ⨾ R*
      ⊑⟨   ⊔-upper₂ ⟩
           Id ⊔ R ⊔ R* ⨾ R*
      ≈⟨ ≈-sym *-recDef ⟩
           R*
    □


  *-isTransitive : R* ⨾ R* ⊑ R*
  *-isTransitive = ⊑-begin
           R* ⨾ R*
      ⊑⟨   ⊔-upper₂ ⟩
           R ⊔ R* ⨾ R*
      ⊑⟨   ⊔-upper₂ ⟩
           Id ⊔ R ⊔ R* ⨾ R*
      ≈⟨ ≈-sym *-recDef ⟩
           R*
    □


  *-rightInd≈ : {B : Obj} {Q : Mor B A} → Q ⨾ R ⊑ Q → Q ⨾ R* ≈ Q
  *-rightInd≈ Q⨾R⊑Q = ⊑-antisym (*-rightInd Q⨾R⊑Q) (proj₂ *-isSuperidentity)
  *-leftInd≈ : {B : Obj} {S : Mor A B} → R ⨾ S ⊑ S → R* ⨾ S ≈ S
  *-leftInd≈ R⨾S⊑S = ⊑-antisym (*-leftInd R⨾S⊑S) (proj₁ *-isSuperidentity)


  *-is-⨾-idempotent : R* ⨾ R* ≈ R*
  *-is-⨾-idempotent = ⊑-antisym *-isTransitive (≈-sym rightId ⟨≈⊑⟩ ⨾-monotone₂ *-isReflexive)


  *-isRTC : {S : Mor A A} → R ⊑ S → Id ⊑ S → S ⨾ S ⊑ S → R* ⊑ S
  *-isRTC {S} R⊑S Id⊑S S⨾S⊑S = ⊑-begin
           R*
      ≈⟨ ≈-sym leftId ⟩
           Id ⨾ R*
      ⊑⟨ ⨾-monotone₁ Id⊑S ⟩
           S ⨾ R*
      ⊑⟨ *-rightInd (⨾-monotone₂ R⊑S ⟨⊑⊑⟩ S⨾S⊑S) ⟩
           S
    □


IsStar-subst₁      : {A : Obj} {R R′ R* : Mor A A}
                → R ≈ R′ → IsStar {A} R R* → IsStar {A} R′ R*
IsStar-subst₁ {A} {R} {R′} {R*} R≈R′ isStar = let open IsStar isStar in record
  {*-recDef = ≈-begin
           R*
    ≈⟨ *-recDef ⟩
           Id ⊔ R ⊔ R* ⨾ R*
```

```
    ≈⟨ ⊔-cong₂₁ R≈R′ ⟩
          Id ⊔ R′ ⊔ R* ⨾ R*
    □
  ; *-leftInd  =  λ R′⨾S⊑S → *-leftInd (⨾-cong₁ R≈R′ ⟨≈⊑⟩ R′⨾S⊑S)
  ; *-rightInd  =  λ Q⨾R′⊑Q → *-rightInd (⨾-cong₂ R≈R′ ⟨≈⊑⟩ Q⨾R′⊑Q)
  }

IsStar-subst₂      : {A : Obj} {R R* R*′ : Mor A A}
                   → R* ≈ R*′ → IsStar {A} R R* → IsStar {A} R R*′
IsStar-subst₂ {A} {R} {R*} {R*′} R*≈R*′ isStar  =  let open IsStar isStar in record
  {*-recDef  =  ≈-begin
          R*′
    ≈˘⟨ R*≈R*′ ⟩
          R*
    ≈⟨ *-recDef ⟩
          Id ⊔ R ⊔ R* ⨾ R*
    ≈⟨ ⊔-cong₂₂ (⨾-cong R*≈R*′ R*≈R*′) ⟩
          Id ⊔ R ⊔ R*′ ⨾ R*′
    □
  ; *-leftInd    =  λ R⨾S⊑S  → ⨾-cong₁ R*≈R*′ ⟨≈˘⊑⟩ *-leftInd R⨾S⊑S
  ; *-rightInd  =  λ Q⨾R⊑Q → ⨾-cong₂ R*≈R*′ ⟨≈˘⊑⟩ *-rightInd Q⨾R⊑Q
  }
```

To show that Kozen's (14) implies our *-recDef (which, as shown above, implies Kozen's (14) and (15) as *-recDef$_1$ and *-recDef$_2$), we define:

```
mkIsStar′  :  {A : Obj} (R R* : Mor A A)
              (*-recDef₁ : Id ⊔ R ⨾ R* ⊑ R*)
              (*-leftInd : {B : Obj} {S : Mor A B} → R ⨾ S ⊑ S → R* ⨾ S ⊑ S)
              (*-rightInd : {Z : Obj} {Q : Mor Z A} → Q ⨾ R ⊑ Q → Q ⨾ R* ⊑ Q)
           → IsStar {A} R R*
mkIsStar′ {A} R R* *-recDef₁ *-leftInd *-rightInd  =  record
  {*-recDef  =  ⊑-antisym
    (⊑-begin
            R*
        ≈˘⟨ rightId ⟩
            R* ⨾ Id
        ⊑⟨ ⨾-monotone₂ Id⊑R* ⟩
            R* ⨾ R*
        ⊑⟨ ⊔-upper₂ ⟨⊑⊑⟩ ⊔-upper₂ ⟩
            Id ⊔ R ⊔ R* ⨾ R*
      □)
    (⊔-universal Id⊑R*
      (⊔-universal
        (⊑-begin
            R
        ≈˘⟨ rightId ⟩
            R ⨾ Id
        ⊑⟨ ⨾-monotone₂ Id⊑R* ⟩
            R ⨾ R*
        ⊑⟨ ⊔-upper₂ ⟨⊑⊑⟩ *-recDef₁ ⟩
            R*
        □)
        (*-leftInd (⊑-begin
            R ⨾ R*
        ⊑⟨ ⊔-upper₂ ⟨⊑⊑⟩ *-recDef₁ ⟩
            R*
        □))))
  ; *-leftInd  =  *-leftInd
```

```
  ;*-rightInd = *-rightInd
  }
  where
    Id⊑R* : Id ⊑ R*
    Id⊑R* = ⊔-upper₁ ⟨⊑⊑⟩ *-recDef₁
```

To show that furthermore Kozen's (16) and (17) imply his (18) and (19), we define:

```
mkIsStar″ : {A : Obj} (R R* : Mor A A)
            (*-recDef₁ : Id ⊔ R ⨾ R* ⊑ R*)
            (*-leftInd′  : {B : Obj} {P S : Mor A B}
                         → P ⊔ R ⨾ S ⊑ S → R* ⨾ P ⊑ S)
            (*-rightInd′ : {B : Obj} {P Q : Mor B A}
                         → P ⊔ Q ⨾ R ⊑ Q → P ⨾ R* ⊑ Q)
          → IsStar {A} R R*
mkIsStar″ {A}   R R* *-recDef₁ *-leftInd′ *-rightInd′
   = mkIsStar′  R R* *-recDef₁ *-leftInd  *-rightInd
  where
    *-leftInd : {B : Obj} {S : Mor A B} → R ⨾ S ⊑ S → R* ⨾ S ⊑ S
    *-leftInd {_} {S} R⨾S⊑S = let
        S⊔R⨾S⊑S : S ⊔ R ⨾ S ⊑ S
        S⊔R⨾S⊑S = ⊔-universal ⊑-refl R⨾S⊑S
      in *-leftInd′ S⊔R⨾S⊑S
    *-rightInd : {Z : Obj} {Q : Mor Z A} → Q ⨾ R ⊑ Q → Q ⨾ R* ⊑ Q
    *-rightInd {_} {Q} Q⨾R⊑Q = let
        Q⊔Q⨾R⊑Q : Q ⊔ Q ⨾ R ⊑ Q
        Q⊔Q⨾R⊑Q = ⊔-universal ⊑-refl Q⨾R⊑Q
      in *-rightInd′ Q⊔Q⨾R⊑Q


record LocalStarOp (A : Obj) : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  infix 20  _*
  field
    _* : Mor A A → Mor A A
    isStar : (R : Mor A A) → IsStar {A} R (R *)
  private
    module Local {R : Mor A A} where
      open IsStar (isStar R) public
  open Local public


  *-monotone : {R S : Mor A A} → R ⊑ S → R * ⊑ S *
  *-monotone {R} {S} R⊑S = ⊑-begin
        R *
      ≈⟨ *-recDef₂ ⟩
        Id ⊔ R * ⨾ R
      ⊑⟨ ⊔-universal *-isReflexive (*-leftInd′ R⊔R⨾S*⊑S*) ⟩
        S *
      □
    where
      R⊔R⨾S*⊑S* : R ⊔ R ⨾ S * ⊑ S *
      R⊔R⨾S*⊑S* = ⊑-begin
          R ⊔ R ⨾ S *
        ⊑⟨ ⊔-monotone R⊑S (⨾-monotone₁ R⊑S) ⟩
          S ⊔ S ⨾ S *
        ⊑⟨ ⊔-universal *-increases *-stepL ⟩
          S *
        □

  *-cong : {R S : Mor A A} → R ≈ S → R * ≈ S *
  *-cong R≈S = ⊑-antisym (*-monotone (⊑-reflexive R≈S)) (*-monotone (⊑-reflexive′ R≈S))
```

```
  ** :    {R : Mor A A} → (R *) * ≈ R *
  ** {R} = ⊑-antisym (*-isRTC ⊑-refl *-isReflexive *-isTransitive) *-increases


record StarOp : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  infix 20 _*
  field
    _* : {A : Obj} → Mor A A → Mor A A
    isStar : {A : Obj} (R : Mor A A) → IsStar {A} R (R *)
  localStarOp : (A : Obj) → LocalStarOp A
  localStarOp A = record {_* = _*; isStar = isStar}
  private
    module Local {A : Obj} where
      open LocalStarOp (localStarOp A) public hiding (_*; isStar)
  open Local public


  *-⨾-roll⊑ : {A B : Obj} {R : Mor A B} {S : Mor B A} → (R ⨾ S) * ⨾ R ⊑ R ⨾ (S ⨾ R) *
  *-⨾-roll⊑ {_} {_} {R} {S} = ⊑-begin
      (R ⨾ S) * ⨾ R
    ⊑⟨ proj₂ *-isSuperidentity ⟨⊑≈⟩ ⨾-assoc ⟩
      (R ⨾ S) * ⨾ R ⨾ (S ⨾ R) *
    ⊑⟨ *-leftInd (⊑-begin
        (R ⨾ S) ⨾ R ⨾ (S ⨾ R) *
      ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ ⨾-assoc ⟨≈≈⟩ ⨾-assoc ⟩
        R ⨾ (S ⨾ R) ⨾ (S ⨾ R) *
      ⊑⟨ ⨾-monotone₂ *-stepL ⟩
        R ⨾ (S ⨾ R) *
        □) ⟩
      R ⨾ (S ⨾ R) *
    □
  ⨾-*-roll⊑ : {A B : Obj} {R : Mor A B} {S : Mor B A} → R ⨾ (S ⨾ R) * ⊑ (R ⨾ S) * ⨾ R
  ⨾-*-roll⊑ {_} {_} {R} {S} = ⊑-begin
      R ⨾ (S ⨾ R) *
    ⊑⟨ proj₁ *-isSuperidentity ⟨⊑≈⟩ ⨾-assocL ⟩
      ((R ⨾ S) * ⨾ R) ⨾ (S ⨾ R) *
    ⊑⟨ *-rightInd (⊑-begin
        ((R ⨾ S) * ⨾ R) ⨾ (S ⨾ R)
      ≈⟨ ⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⨾-assocL ⟨≈≈⟩ ⨾-assocL ⟩
        ((R ⨾ S) * ⨾ (R ⨾ S)) ⨾ R
      ⊑⟨ ⨾-monotone₁ *-stepR ⟩
        (R ⨾ S) * ⨾ R
        □) ⟩
      (R ⨾ S) * ⨾ R
    □
  *-⨾-roll : {A B : Obj} {R : Mor A B} {S : Mor B A} → (R ⨾ S) * ⨾ R ≈ R ⨾ (S ⨾ R) *
  *-⨾-roll = ⊑-antisym *-⨾-roll⊑ ⨾-*-roll⊑


StarOpFromLocal : ((A : Obj) → LocalStarOp A) → StarOp
StarOpFromLocal local = record
  {_* = λ {A} R → LocalStarOp._* (local A) R
  ; isStar = λ {A} R → LocalStarOp.isStar (local A) R
  }


InitialStarOp : {I : Obj} → IsInitial I → LocalStarOp I
InitialStarOp {I} I-isInitial = record
  {_* = λ _ → Id
  ; isStar = λ R → record
```

```
    {*-recDef  =  IsInitial≈ I-isInitial
    ; *-leftInd  =  λ R⨾S⊑S → ⊑-reflexive leftId
    ; *-rightInd  =  λ Q⨾R⊑Q → ⊑-reflexive rightId
    }
  }
```

Since we want the base parameter to be explicit for the **record** types, but implicit for the associated modules, we cannot just re-export the modules directly from InUSLCat. (This is the effect we would have obtained by not factoring out InUSLCat.)

```
open InUSLCat public hiding
  (module IsStar
  ; module LocalStarOp
  ; module StarOp
  ; StarOpFromLocal
  ; IsStar-subst₁ ; IsStar-subst₂
  )
module InUSLCat′ {i j k₁ k₂ : Level} {Obj : Set i}
                 {base : USLCategory j k₁ k₂ Obj} where
  open InUSLCat public using (IsStar-subst₁ ; IsStar-subst₂ ; StarOpFromLocal)
open InUSLCat′ public
module IsStar      {i j k₁ k₂ : Level} {Obj : Set i} {base : USLCategory j k₁ k₂ Obj}
       = InUSLCat.IsStar base
module LocalStarOp {i j k₁ k₂ : Level} {Obj : Set i} {base : USLCategory j k₁ k₂ Obj}
       = InUSLCat.LocalStarOp base
module StarOp      {i j k₁ k₂ : Level} {Obj : Set i} {base : USLCategory j k₁ k₂ Obj}
       = InUSLCat.StarOp base

record KleeneCategory {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field uslCategory : USLCategory j k₁ k₂ Obj
  open USLCategory uslCategory
  field zeroMor : ZeroMor orderedSemigroupoid
  field starOp  : StarOp uslCategory
  open USLCategory uslCategory public
  open ZeroMor     zeroMor     public
  open StarOp      starOp      public

retractStarOp : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
              → (F : Obj₂ → Obj₁)
              → {base : USLCategory j k₁ k₂ Obj₁}
              → StarOp base → StarOp (retractUSLCategory F base)
retractStarOp F {base} starOp = let open StarOp {base = base} starOp in record
  {_* = _*
  ; isStar = λ {A} R → record
    {*-recDef   = *-recDef
    ; *-leftInd  = *-leftInd
    ; *-rightInd = *-rightInd
    }
  }

retractKleeneCategory : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
                      → (F : Obj₂ → Obj₁)
                      → KleeneCategory j k₁ k₂ Obj₁ → KleeneCategory j k₁ k₂ Obj₂
retractKleeneCategory F base = let open KleeneCategory base in record
  {uslCategory = retractUSLCategory F uslCategory
  ; starOp = retractStarOp F starOp
  ; zeroMor = retractZeroMor F zeroMor
  }
```

14.4 Categoric.KCC

```
module InUSLCC {i j k₁ k₂ : Level} {Obj : Set i} (uslcc : USLCC j k₁ k₂ Obj) where
  open USLCC uslcc


UnitStarOp : {U : Obj} → IsUnit U → LocalStarOp uslCategory U
UnitStarOp {U} U-isUnit = let open IsUnit U-isUnit in record
  {_* = λ _ → Id
  ; isStar = λ R → record
    {*-recDef = ≈-sym (⊑-to-⊔₁ Id-isTop)
    ; *-leftInd = λ R⨾S⊑S → ⊑-reflexive leftId
    ; *-rightInd = λ Q⨾R⊑Q → ⊑-reflexive rightId
    }
  }


open InUSLCC public
```

We add converse to Kleene categories by adding zero morphisms and Kleene star to upper semi-lattice categories with converse. We show important properties again in a separate module:

```
module KCC-Props {i j k₁ k₂ : Level} {Obj : Set i}
                 (uslcc : USLCC j k₁ k₂ Obj)
                 (starOp : StarOp (USLCC.uslCategory uslcc))
  where
  open USLCC uslcc
  open StarOp starOp


˘-*-subcomm : {A : Obj} → {R : Mor A A} → (R ˘) * ⊑ (R *) ˘
˘-*-subcomm {A} {R} = *-isRTC
    (˘-monotone (*-increases))
    (≈-sym Id˘ ⟨≈⊑⟩ ˘-monotone *-isReflexive)
    (⊑-begin
      (R *) ˘ ⨾ (R *) ˘
    ≈⟨ ≈-sym ˘-involution ⟩
      (R * ⨾ R *) ˘
    ⊑⟨ ˘-monotone *-isTransitive ⟩
      (R *) ˘
  □)
˘-* : {A : Obj} → {R : Mor A A} → (R *) ˘ ≈ (R ˘) *
˘-* {A} {R} = ⊑-antisym
  (˘-⊑-swap (⊑-begin
      R *
    ≈⟨ *-cong (≈-sym ˘˘) ⟩
        ((R ˘) ˘) *
    ⊑⟨ ˘-*-subcomm ⟩
      ((R ˘) *) ˘
    □)
  )
  ˘-*-subcomm


*-isSymmetric : {A : Obj} → {R : Mor A A} → isSymmetric R → isSymmetric (R *)
*-isSymmetric {A} {R} R-isSym = ≈-begin
      (R *) ˘
    ≈⟨ ˘-* ⟩
      (R ˘) *
```

```
      ≈⟨ *-cong R-isSym ⟩
        R *
      □
```

An exploration, with lemmas useful even without presence of the domSpread name:

```
domSpread : {A B : Obj} → (R : Mor A B) → Mor A A
domSpread R = (R ⨾ R ˘) *

domSpread-isSymmetric : {A B : Obj} → {R : Mor A B} → isSymmetric (domSpread R)
domSpread-isSymmetric {A} {B} {R} = ≈-begin
        ((R ⨾ R ˘) *) ˘ ≈⟨ ˘-* ⟩                        ((R ⨾ R ˘) ˘) *
                        ≈⟨ *-cong ˘-involutionRightConv ⟩ (R ⨾ R ˘) * □

domSpread-cong : {A B : Obj} → {R S : Mor A B} → R ≈ S → domSpread R ≈ domSpread S
domSpread-cong R≈S = *-cong (⨾-cong R≈S (˘-cong R≈S))

domSpread-idempot : {A B : Obj} → {R : Mor A B} → domSpread (domSpread R) ≈ domSpread R
domSpread-idempot {A} {B} {R} = ≈-begin
        domSpread (domSpread R)
      ≈⟨ domSpread-cong ≈-refl ⟩
        domSpread ((R ⨾ R ˘) *)
      ≈⟨ ≈-refl ⟩
        ((R ⨾ R ˘) * ⨾ ((R ⨾ R ˘) *) ˘) *
      ≈⟨ *-cong (⨾-cong₂ (˘-* ⟨≈≈⟩ *-cong ˘-involutionRightConv)) ⟩
        ((R ⨾ R ˘) * ⨾ (R ⨾ R ˘) *) *
      ≈⟨ *-cong *-is-⨾-idempotent ⟩
        ((R ⨾ R ˘) *) *
      ≈⟨ ** ⟩
        (R ⨾ R ˘) *
      ≈⟨ ≈-sym ≈-refl ⟩
        domSpread R
  □
```

In a KCC, box star morphisms (difunctional closures) exist and can be defined using Kleene star:

```
boxStar : {A B : Obj} → Mor A B → Mor A B
boxStar R = (R ⨾ R ˘) * ⨾ R

boxStarProof : {A B : Obj} (R : Mor A B) → boxStar R isBoxStarOf R
boxStarProof R = let Rb = boxStar R in record
  {isBoxStar-incl = ⊑-begin
          R
        ≈⟨ ≈-sym leftId ⟩
          Id ⨾ R
        ⊑⟨ ⨾-monotone₁ *-isReflexive ⟩
          boxStar R
      □
  ; isBoxStar-difun = ⊑-begin
          boxStar R ⨾ boxStar R ˘ ⨾ boxStar R
        ≈⟨ ≈-refl ⟩
          ((R ⨾ R ˘) * ⨾ R) ⨾ ((R ⨾ R ˘) * ⨾ R) ˘ ⨾ (R ⨾ R ˘) * ⨾ R
        ≈⟨ ⨾-cong₂₁ ˘-involution ⟨≈≈⟩ ⨾-assocL ⟩
          (((R ⨾ R ˘) * ⨾ R) ⨾ (R ˘ ⨾ (R ⨾ R ˘) * ˘)) ⨾ (R ⨾ R ˘) * ⨾ R
        ≈⟨ ⨾-cong₁ (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ⨾-assocL) ⟩
          ((R ⨾ R ˘) * ⨾ (R ⨾ R ˘) ⨾ (R ⨾ R ˘) * ˘) ⨾ (R ⨾ R ˘) * ⨾ R
        ≈⟨ ⨾-cong₁₂₂ domSpread-isSymmetric ⟩
          ((R ⨾ R ˘) * ⨾ (R ⨾ R ˘) ⨾ (R ⨾ R ˘) *) ⨾ (R ⨾ R ˘) * ⨾ R
        ⊑⟨ ⨾-monotone₁ (⨾-monotone₂ *-stepL ⟨⊑≈⟩ *-is-⨾-idempotent) ⟩
          (R ⨾ R ˘) * ⨾ (R ⨾ R ˘) * ⨾ R
        ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ *-is-⨾-idempotent ⟩
          (R ⨾ R ˘) * ⨾ R
```

```
        ≈⟨ ≈-refl ⟩
          boxStar R
      □
  ; isBoxStar-leftInd = λ {C} {P} {Q} P⨾R⊑Q Q⨾R˘⨾R⊑Q → ⊑-begin
          P ⨾ boxStar R
        ≈⟨ ≈-refl ⟩
          P ⨾ (R ⨾ R ˘) * ⨾ R
        ≈⟨ ⨾-cong₂ *-⨾-roll ⟩
          P ⨾ R ⨾ (R ˘ ⨾ R) *
        ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ P⨾R⊑Q ⟩
          Q ⨾ (R ˘ ⨾ R) *
        ⊑⟨ *-rightInd Q⨾R˘⨾R⊑Q ⟩
          Q
      □
  ; isBoxStar-rightInd = λ {C} {P} {Q} R⨾P⊑Q R⨾R˘⨾Q⊑Q → ⊑-begin
          boxStar R ⨾ P
        ≈⟨ ⨾-assoc ⟩
          (R ⨾ R ˘) * ⨾ R ⨾ P
        ⊑⟨ ⨾-monotone₂ R⨾P⊑Q ⟩
          (R ⨾ R ˘) * ⨾ Q
        ⊑⟨ *-leftInd (⨾-assoc ⟨≈⊑⟩ R⨾R˘⨾Q⊑Q) ⟩
          Q
      □
  }
```

In a KCC, we can also define equivalence closure:

```
equClos : {A : Obj} → Mor A A → Mor A A
equClos R = (R ⊔ R ˘) *

equClos-isReflexive : {A : Obj} {R : Mor A A} → isReflexive (equClos R)
equClos-isReflexive = *-isReflexive

equClos-isSymmetric : {A : Obj} {R : Mor A A} → isSymmetric (equClos R)
equClos-isSymmetric {A} {R} = *-isSymmetric {A} {R ⊔ R ˘} (≈-begin
    (R ⊔ R ˘) ˘
  ≈⟨ ˘-⊔-distrib ⟩
    R ˘ ⊔ R ˘ ˘
  ≈⟨ ⊔-cong₂ ˘˘ ⟨≈≈⟩ ⊔-commutative ⟩
    R ⊔ R ˘
  □)

equClos-isIdempotent : {A : Obj} {R : Mor A A} → isIdempotent (equClos R)
equClos-isIdempotent = *-is-⨾-idempotent

equClos-isTransitive : {A : Obj} {R : Mor A A} → isTransitive (equClos R)
equClos-isTransitive = *-isTransitive

equClos-IsSymIdempot : {A : Obj} (R : Mor A A) → IsSymIdempot (equClos R)
equClos-IsSymIdempot R = record {symmetric = equClos-isSymmetric; idempotent = equClos-isIdempotent}

EquClos : {A : Obj} (R : Mor A A) → SymIdempot
EquClos R = record {⟪_⟫ = equClos R; prop = equClos-IsSymIdempot R}
```

```
record KCC {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field uslcc : USLCC j k₁ k₂ Obj
  open USLCC uslcc

  field zeroMor : ZeroMor orderedSemigroupoid
        starOp  : StarOp uslCategory

  kleeneCategory : KleeneCategory j k₁ k₂ Obj
  kleeneCategory = record
    {uslCategory = uslCategory
    ; starOp = starOp
```

```
    ; zeroMor = zeroMor
    }
  open KCC-Props uslcc starOp public
  open StarOp    starOp  public
  open USLCC     uslcc   public
  open ZeroMor   zeroMor public


retractKCC : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
           → (F : Obj₂ → Obj₁)
           → KCC j k₁ k₂ Obj₁ → KCC j k₁ k₂ Obj₂
retractKCC F base = let open KCC base in record
  {uslcc = retractUSLCC F uslcc
  ; starOp = retractStarOp F starOp
  ; zeroMor = retractZeroMor F zeroMor
  }
```

14.5 Categoric.KleeneCollagory

A *Kleene collagory* is a collagory that also has Kleene star, but does not need to have zero morphisms. For the time being, we do not introduce intermediate theories (in particular Kleene smi-collagories), but just add iteration to collagories.

```
record KleeneCollagory {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field   collagory : Collagory j k₁ k₂ Obj
  open    Collagory collagory
  field   starOp         : StarOp uslCategory
  open InUSLCC    uslcc         public
  open KCC-Props uslcc starOp public
  open StarOp    starOp         public
```

For two mappings F and G from A to B, given a splitting for equClos (F ˘ ⨾ G), we can construct a co-equaliser (in the mapping category) for F and G.

```
  mappingCoEqualiser : {A B : Obj} (F G : Mapping A B)
                     → let V = Mapping.mor F ˘ ⨾ Mapping.mor G; W = equClos V
                       in {C : Obj} {H : Mor B C}
                     → IsSymSplitting W H
                     → CatFinColimits.CoEqualiser (MapCat occ) F G
  mappingCoEqualiser {A} {B} F G {C} {H} HsplitsW = record
    {obj = C
    ; mor = H′
    ; prop = ≈-begin
          Mapping.mor F ⨾ H
      ≈˘⟨ ⨾-cong₂ HsplitsW.leftClosed ⟩
          Mapping.mor F ⨾ W ⨾ H
      ≈⟨ ⨾-cong₁&₂₁ F⨾W≈G⨾W ⟩
          Mapping.mor G ⨾ W ⨾ H
      ≈⟨ ⨾-cong₂ HsplitsW.leftClosed ⟩
          Mapping.mor G ⨾ H
      □
    ; universal = λ {Z} {R} F⨾R≈G⨾R
          → let
            H⨾H˘⨾R⊑R : H ⨾ H ˘ ⨾ Mapping.mor R ⊑ Mapping.mor R
            H⨾H˘⨾R⊑R = ⊑-begin
                        H ⨾ H ˘ ⨾ Mapping.mor R
```

```
                ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ HsplitsW.factors ⟩
                         W ⨾ Mapping.mor R
                ⊑⟨ *-leftInd (⊑-begin
                          (V ⊔ V ˘) ⨾ Mapping.mor R
                        ⊑⟨ ⨾-⊔-subdistribL ⟨⊑≈⟩ ⊔-cong ⨾-assoc (⨾-cong₁ ˘-involutionLeftConv ⟨≈≈⟩ ⨾-assoc) ⟩
                          F₀ ˘ ⨾ G₀ ⨾ Mapping.mor R ⊔ G₀ ˘ ⨾ F₀ ⨾ Mapping.mor R
                        ≈⟨ ⊔-cong (⨾-cong₂ (≈-sym F⨾R≈G⨾R)) (⨾-cong₂ F⨾R≈G⨾R) ⟩
                          F₀ ˘ ⨾ F₀ ⨾ Mapping.mor R ⊔ G₀ ˘ ⨾ G₀ ⨾ Mapping.mor R
                        ⊑⟨ ⊔-monotone (⨾-assocL ⟨≈⊑⟩ proj₁ (Mapping.unival F))
                                      (⨾-assocL ⟨≈⊑⟩ proj₁ (Mapping.unival G)) ⟩
                          Mapping.mor R ⊔ Mapping.mor R
                        ≈⟨ ⊔-idempotent ⟩
                          Mapping.mor R
                        □)
                      ⟩
                         Mapping.mor R
                □
              H⨾H˘⨾R≈R : H ⨾ H ˘ ⨾ Mapping.mor R ≈ Mapping.mor R
              H⨾H˘⨾R≈R = ⊑-antisym H⨾H˘⨾R⊑R (proj₁ (reflexiveIsSuperidentity (≈-isReflexive HsplitsW.factors *-isReflexive)) ⟨⊑≈⟩ ⨾-asso
            in mkMapping (H ˘ ⨾ Mapping.mor R)
              (⊑-isSubidentity (⊑-begin
                      (H ˘ ⨾ Mapping.mor R) ˘ ⨾ H ˘ ⨾ Mapping.mor R
                ≈⟨ ⨾-cong₁ ˘-involutionLeftConv ⟨≈≈⟩ ⨾-assoc ⟩
                      Mapping.mor R ˘ ⨾ H ⨾ H ˘ ⨾ Mapping.mor R
                ⊑⟨ ⨾-monotone₂ H⨾H˘⨾R⊑R ⟩
                      Mapping.mor R ˘ ⨾ Mapping.mor R
                □) (Mapping.unival R)
              , ⊑-isSuperidentity (⊑-begin
                      H ˘ ⨾ H
                ⊑⟨ ⨾-monotone₂ (proj₁ (Mapping.total R) ⟨⊑≈⟩ ⨾-assoc) ⟩
                      H ˘ ⨾ Mapping.mor R ⨾ Mapping.mor R ˘ ⨾ H
                ≈˘⟨ ⨾-cong₂ ˘-involutionLeftConv ⟨≈≈⟩ ⨾-assoc ⟩
                      (H ˘ ⨾ Mapping.mor R) ⨾ (H ˘ ⨾ Mapping.mor R) ˘
                □) (isIdentity-super HsplitsW.splitId)
              )
            , ≈-sym H⨾H˘⨾R≈R
            , (λ {U′} R≈H′⨾U′ → ≈-begin
                  H ˘ ⨾ Mapping.mor R
                ≈⟨ ⨾-cong₂ R≈H′⨾U′ ⟩
                  H ˘ ⨾ H ⨾ Mapping.mor U′
                ≈⟨ ⨾-assocL ⟨≈≈⟩ proj₁ (HsplitsW.splitId) ⟩
                  Mapping.mor U′
                □)
    }
    where
      module HsplitsW = IsSymSplitting HsplitsW
      F₀ = Mapping.mor F
      G₀ = Mapping.mor G
      V = F₀ ˘ ⨾ G₀
      W = equClos V
      H′ : Mapping B C
      H′ = mkMapping H (isIdentity-sub HsplitsW.splitId
                         , reflexiveIsSuperidentity (≈-isReflexive HsplitsW.factors *-isReflexive))
      F⨾W≈G⨾W : Mapping.mor F ⨾ W ≈ Mapping.mor G ⨾ W
      F⨾W≈G⨾W = ⊑-antisym
        (⊑-begin
              Mapping.mor F ⨾ W
          ⊑⟨ proj₁ (Mapping.total G) ⟨⊑≈⟩ ⨾-assoc ⟩
              Mapping.mor G ⨾ Mapping.mor G ˘ ⨾ Mapping.mor F ⨾ W
```

```
          ⊑⟨ ⨾-monotone₂ (⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ (˘-involutionLeftConv ⟨≈˘⊑⟩ ⊔-upper₂)) ⟩
            Mapping.mor G ⨾ (V ⊔ V ˘) ⨾ W
          ⊑⟨ ⨾-monotone₂ *-stepL ⟩
            Mapping.mor G ⨾ W
        □)
      (⊑-begin
            Mapping.mor G ⨾ W
          ⊑⟨ proj₁ (Mapping.total F) ⟨⊑≈⟩ ⨾-assoc ⟩
            Mapping.mor F ⨾ Mapping.mor F ˘ ⨾ Mapping.mor G ⨾ W
          ⊑⟨ ⨾-monotone₂ (⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ ⊔-upper₁) ⟩
            Mapping.mor F ⨾ (V ⊔ V ˘) ⨾ W
          ⊑⟨ ⨾-monotone₂ *-stepL ⟩
            Mapping.mor F ⨾ W
        □)

  mappingCoEqualiser′ : {A B : Obj} (F G : Mapping A B)
                      → SymSplitting (equClos (Mapping.mor F ˘ ⨾ Mapping.mor G))
                      → CatFinColimits.CoEqualiser (MapCat occ) F G
  mappingCoEqualiser′    {A} {B} F G WSplit = let open SymSplitting WSplit in
    mappingCoEqualiser   {A} {B} F G {obj} {mor} proof


  open Collagory  collagory   public


retractKleeneCollagory : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
  → (F : Obj₂ → Obj₁)
  → KleeneCollagory j k₁ k₂ Obj₁ → KleeneCollagory j k₁ k₂ Obj₂
retractKleeneCollagory F base = let open KleeneCollagory base in record
  {collagory = retractCollagory F collagory
  ;starOp = retractStarOp F starOp
  }
```

14.6 Categoric.ActLatSemigroupoid

```
record ActLatSemigroupoid {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field kleeneSemigroupoid  : KleeneSemigroupoid j k₁ k₂ Obj
  open KleeneSemigroupoid kleeneSemigroupoid

  field meetOp     : MeetOp     orderedSemigroupoid
        leftResOp  : LeftResOp  orderedSemigroupoid
        rightResOp : RightResOp orderedSemigroupoid

  latticeSemigroupoid : LatticeSemigroupoid j k₁ k₂ Obj
  latticeSemigroupoid = record
    {orderedSemigroupoid = orderedSemigroupoid
    ;meetOp = meetOp
    ;joinOp = joinOp
    }
    -- In Agda-2.3.0, using takes too much time and memory.
    -- open LatticeSemigroupoid latticeSemigroupoid public using (lslSemigroupoid)
  lslSemigroupoid = LatticeSemigroupoid.lslSemigroupoid latticeSemigroupoid

  open MeetOp               meetOp                       public
  open HomLattice orderedSemigroupoid meetOp joinOp      public
  open LeftResOp            leftResOp                    public
  open RightResOp           rightResOp                   public
  open KleeneSemigroupoid kleeneSemigroupoid             public
```

```
retractActLatSemigroupoid : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
  → (F : Obj₂ → Obj₁) → ActLatSemigroupoid j k₁ k₂ Obj₁ → ActLatSemigroupoid j k₁ k₂ Obj₂
retractActLatSemigroupoid F base = let open ActLatSemigroupoid base in record
  {kleeneSemigroupoid = retractKleeneSemigroupoid F kleeneSemigroupoid
  ; meetOp = retractMeetOp F meetOp
  ; leftResOp = retractLeftResOp F leftResOp
  ; rightResOp = retractRightResOp F rightResOp
  }
```

14.7 Categoric.ActLatCategory

```
record ActLatCategory {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field kleeneCategory : KleeneCategory j k₁ k₂ Obj
  open KleeneCategory kleeneCategory
  field meetOp     : MeetOp     orderedSemigroupoid
        leftResOp  : LeftResOp  orderedSemigroupoid
        rightResOp : RightResOp orderedSemigroupoid
  latticeSemigroupoid : LatticeSemigroupoid j k₁ k₂ Obj
  latticeSemigroupoid = record
    {orderedSemigroupoid = orderedSemigroupoid
    ; meetOp = meetOp
    ; joinOp = joinOp
    }
    -- As of Agda-2.3.0, using is still too expensive.
    -- open LatticeSemigroupoid latticeSemigroupoid public using (lslSemigroupoid)
  lslSemigroupoid = LatticeSemigroupoid.lslSemigroupoid latticeSemigroupoid
  open MeetOp         meetOp                                    public
  open HomLattice orderedSemigroupoid meetOp joinOp             public
  open LeftResOp      leftResOp                                 public
  open RightResOp     rightResOp                                public
  open KleeneCategory kleeneCategory                            public

retractActLatCategory : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
  → (F : Obj₂ → Obj₁) → ActLatCategory j k₁ k₂ Obj₁ → ActLatCategory j k₁ k₂ Obj₂
retractActLatCategory F base = let open ActLatCategory base in record
  {kleeneCategory = retractKleeneCategory F kleeneCategory
  ; meetOp = retractMeetOp F meetOp
  ; leftResOp = retractLeftResOp F leftResOp
  ; rightResOp = retractRightResOp F rightResOp
  }
```

14.8 Categoric.DistrActAllegory

A *distributive action allegory* is an action lattice category that is also a distributive allegory, and therefore also a division allegory. For the time being, we do not introduce intermediate theories, and just add iteration to division allegories.

```
record DistrActAllegory {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊍ ℓsuc (j ⊍ k₁ ⊍ k₂)) where
  field divAllegory : DivAllegory j k₁ k₂ Obj
  open DivAllegory divAllegory
  field starOp : StarOp uslCategory
  kcc : KCC j k₁ k₂ Obj
```

```
  kcc = record
    {uslcc = uslcc
    ; starOp = starOp
    ; zeroMor = zeroMor
    }
  actLatCategory : ActLatCategory j k₁ k₂ Obj
  actLatCategory = record
    {kleeneCategory = KCC.kleeneCategory kcc
    ; meetOp = meetOp
    ; leftResOp = leftResOp
    ; rightResOp = rightResOp
    }
  open KCC-Props uslcc starOp public
  open StarOp starOp public
  open DivAllegory divAllegory public

retractDistrActAllegory : {i₁ i₂ j k₁ k₂ : Level} {Obj₁ : Set i₁} {Obj₂ : Set i₂}
  → (F : Obj₂ → Obj₁)
  → DistrActAllegory j k₁ k₂ Obj₁ → DistrActAllegory j k₁ k₂ Obj₂
retractDistrActAllegory F base = let open DistrActAllegory base in record
  {divAllegory = retractDivAllegory F divAllegory
  ; starOp = retractStarOp F starOp
  }
```

Chapter 15

Direct Sums

Direct sums are a local relation-algebraic generalisation of disjoint union; we show a fine-grained development of their theory in Sect. 15.1, and in Sect. 15.2 we use direct sums to decompose the calculation of Kleene star.

15.1 Categoric.DirectSum

```
module Categoric.DirectSum
    {i j k₁ k₂ : Level} {Obj : Set i}
    (base : USLCC j k₁ k₂ Obj)
  where
    open USLCC base
    open LeastMor orderedSemigroupoid
    open Categoric.OSGC.LeastMor osgc
```

In (Kahl, 2011a), we considered direct sums in collagories, which do not have zero morphisms. Much of the develepment there actually only requires a USLCC.

Here, while working in the context of a USLCC without zero morphisms, we formulate the definition of IsDirectSum-L in a way that would allow it to be set in a USLSGC without zero morphisms. This way we derive already some of the results of (Kahl, 2011a, Sect. 3.8), and are prepared to refactor the current module once the need arises.

The suffic “-L” stands for “in the context of assuming only a Least morphism” (as opposed to assuming zero morphisms).

```
record IsDirectSum-L {A B S : Obj} {⊥ : Mor A B}
                    (is-⊥ : isLeastMor ⊥) (ι : Mor A S) (κ : Mor B S) : Set (i ⊍ j ⊍ k₁) where
  field
    commutes : ι ⨾ κ ˘ ≈ ⊥
    jointId       : isIdentity (ι ˘ ⨾ ι ⊔ κ ˘ ⨾ κ)
    leftKernel : isIdentity (ι ⨾ ι ˘)
    rightKernel : isIdentity (κ ⨾ κ ˘)


  leftUnival : isUnivalent ι
  leftUnival = ⊑-isSubidentity ⊔-upper₁ (isIdentity-sub jointId)
  leftTotal      : isTotal ι
  leftTotal = isIdentity-super leftKernel
  leftInj     : isInjective ι
  leftInj         = isIdentity-sub leftKernel

  rightUnival : isUnivalent κ
  rightUnival = ⊑-isSubidentity ⊔-upper₂ (isIdentity-sub jointId)
  rightTotal : isTotal κ
```

```
rightTotal  =  isIdentity-super rightKernel
rightInj    : isInjective κ
rightInj    =  isIdentity-sub rightKernel
leftMapping  :  isMapping ι
leftMapping  =  leftUnival, leftTotal
rightMapping  :  isMapping κ
rightMapping  =  rightUnival, rightTotal
LeftMapping  :  Mapping A S
LeftMapping  =  mkMapping ι leftMapping
RightMapping  :  Mapping B S
RightMapping  =  mkMapping κ rightMapping

κ⨾ι˘-is-⊥  :  isLeastMor (κ ⨾ ι ˘)
κ⨾ι˘-is-⊥ R  =  ⊑-begin
             κ ⨾ ι ˘
         ≈˘⟨ ˘-involutionRightConv ⟩
             (ι ⨾ κ ˘) ˘
         ≈⟨ ˘-cong commutes ⟩
             ⊥ ˘
         ⊑⟨ ⊥-˘ is-⊥ R ⟩
             R
         □

infixr 5 _⊕_
_⊕_  :  {D : Obj} (F : Mor A D) (G : Mor B D) → Mor S D
F ⊕ G  =  ι ˘ ⨾ F ⊔ κ ˘ ⨾ G
⊕-cong    :    {C : Obj} {F1 F2 : Mor A C} {G1 G2 : Mor B C}
              → F1 ≈ F2 → G1 ≈ G2 → F1 ⊕ G1 ≈ F2 ⊕ G2
⊕-cong F1≈F2 G1≈G2  =  ⊔-cong (⨾-cong2 F1≈F2) (⨾-cong2 G1≈G2)
⊕-cong1     :    {C : Obj} {F1 F2 : Mor A C} {G : Mor B C} → F1 ≈ F2
              → F1 ⊕ G ≈ F2 ⊕ G
⊕-cong1 F1≈F2  =     ⊕-cong F1≈F2 ≈-refl
⊕-cong2     :    {C : Obj} {F : Mor A C} {G1 G2 : Mor B C} → G1 ≈ G2
              → F ⊕ G1 ≈ F ⊕ G2
⊕-cong2 G1≈G2  =     ⊕-cong ≈-refl G1≈G2
⊕-monotone     : {C : Obj} {F1 F2 : Mor A C} {G1 G2 : Mor B C}
              → F1 ⊑ F2 → G1 ⊑ G2 → F1 ⊕ G1 ⊑ F2 ⊕ G2
⊕-monotone F1⊑F2 G1⊑G2  =  ⊔-monotone (⨾-monotone2 F1⊑F2) (⨾-monotone2 G1⊑G2)
⊕-monotone1  :   {C : Obj} {F1 F2 : Mor A C} {G : Mor B C} → F1 ⊑ F2
              → F1 ⊕ G ⊑ F2 ⊕ G
⊕-monotone1 F1⊑F2  =  ⊕-monotone F1⊑F2 ⊑-refl
⊕-monotone2  :   {C : Obj} {F : Mor A C} {G1 G2 : Mor B C} → G1 ⊑ G2
              → F ⊕ G1 ⊑ F ⊕ G2
⊕-monotone2 G1⊑G2  =  ⊕-monotone ⊑-refl G1⊑G2
⊕-⊔-⊕    : {C : Obj}    {F1 F2 : Mor A C} {G1 G2 : Mor B C}
        → (F1 ⊕ G1) ⊔ (F2 ⊕ G2) ≈ (F1 ⊔ F2) ⊕ (G1 ⊔ G2)
⊕-⊔-⊕ {F1 = F1} {F2} {G1} {G2}  =    ≈-begin
          (F1 ⊕ G1) ⊔ (F2 ⊕ G2)
        ≈⟨ ⊔-transpose2 ⟩
          ((ι ˘ ⨾ F1) ⊔ (ι ˘ ⨾ F2)) ⊔ ((κ ˘ ⨾ G1) ⊔ (κ ˘ ⨾ G2))
        ≈˘⟨ ⊔-cong ⨾-⊔-distribR ⨾-⊔-distribR ⟩
          (F1 ⊔ F2) ⊕ (G1 ⊔ G2)
        □
⊕-⨾  :      {C D : Obj} {F1 : Mor A C} {F2 : Mor B C} {G : Mor C D}
        → (F1 ⊕ F2) ⨾ G      ≈ F1 ⨾ G ⊕ F2 ⨾ G
⊕-⨾ {F1 = F1} {F2} {G}  =    ≈-begin
```

```
        (F₁ ⊕ F₂) ⨾ G
      ≈⟨ ⨾-⊔-distribL ⟩
        (ι ˘ ⨾ F₁) ⨾ G ⊔ (κ ˘ ⨾ F₂) ⨾ G
      ≈⟨ ⊔-cong ⨾-assoc ⨾-assoc ⟩
        F₁ ⨾ G ⊕ F₂ ⨾ G
      □

⊕-⨾-⊑     :  {C D : Obj} {F₁ : Mor A C} {F₂ : Mor B C} {G : Mor C D} {H : Mor S D}
      →      ι ˘ ⨾ F₁ ⨾ G ⊑ H → κ ˘ ⨾ F₂ ⨾ G ⊑ H → (F₁ ⊕ F₂) ⨾ G ⊑ H
⊕-⨾-⊑ {F₁ = F₁} {F₂} {G} {H} ι˘⨾F₁⨾G⊑H κ˘⨾F₂⨾G⊑H = ⊑-begin
        (F₁ ⊕ F₂) ⨾ G
      ≈⟨ ⊕-⨾ ⟩
        ι ˘ ⨾ F₁ ⨾ G ⊔ κ ˘ ⨾ F₂ ⨾ G
      ⊑⟨ ⊔-universal ι˘⨾F₁⨾G⊑H κ˘⨾F₂⨾G⊑H ⟩
        H
      □

⊕-⨾-⊑₁     :    {C D : Obj} {F₁ : Mor A C} {F₂ : Mor B C} {G : Mor C D} {H : Mor S D}
      → (F₁ ⊕ F₂) ⨾ G ⊑ H → ι ˘ ⨾ F₁ ⨾ G ⊑ H
⊕-⨾-⊑₁ {F₁ = F₁} {F₂} {G} {H} F⨾G⊑H = ⊑-begin
        ι ˘ ⨾ F₁ ⨾ G
      ⊑⟨ ⊔-upper₁ ⟩
        ι ˘ ⨾ F₁ ⨾ G ⊔ κ ˘ ⨾ F₂ ⨾ G
      ≈˘⟨ ⊕-⨾ ⟩
        (F₁ ⊕ F₂) ⨾ G
      ⊑⟨ F⨾G⊑H ⟩
        H
      □

⊕-⨾-⊑₂     :    {C D : Obj} {F₁ : Mor A C} {F₂ : Mor B C} {G : Mor C D} {H : Mor S D}
      → (F₁ ⊕ F₂) ⨾ G ⊑ H → κ ˘ ⨾ F₂ ⨾ G ⊑ H
⊕-⨾-⊑₂ {F₁ = F₁} {F₂} {G} {H} F⨾G⊑H = ⊑-begin
        κ ˘ ⨾ F₂ ⨾ G
      ⊑⟨ ⊔-upper₂ ⟩
        ι ˘ ⨾ F₁ ⨾ G ⊔ κ ˘ ⨾ F₂ ⨾ G
      ≈˘⟨ ⊕-⨾ ⟩
        (F₁ ⊕ F₂) ⨾ G
      ⊑⟨ F⨾G⊑H ⟩
        H
      □

to-⊕ : {C : Obj} {H : Mor S C} → H ≈ ι ⨾ H ⊕ κ ⨾ H
to-⊕ {_} {H} = ≈-begin
        H
      ≈⟨ proj₁ jointId ⟨≈˘≈⟩ ⨾-⊔-distribL ⟩
        (ι ˘ ⨾ ι) ⨾ H ⊔ (κ ˘ ⨾ κ) ⨾ H
      ≈⟨ ⊔-cong ⨾-assoc ⨾-assoc ⟩
        ι ⨾ H ⊕ κ ⨾ H
      □


Id⊞ : Mor S S
Id⊞ = ι ⊕ κ

Id⊞-isIdentity : isIdentity Id⊞
Id⊞-isIdentity = jointId


ι-⨾-⊕-L      :    {D : Obj} {F : Mor A D} {G : Mor B D} → ι ⨾ (F ⊕ G) ≈ F ⊔ ⊥ ⨾ G
ι-⨾-⊕-L      {_} {F} {G} = ≈-begin
      ι ⨾ (F ⊕ G)
    ≈⟨ ⨾-⊔-distribR ⟩
      ι ⨾ ι ˘ ⨾ F ⊔ ι ⨾ κ ˘ ⨾ G
    ≈⟨ ⊔-cong (⨾-assocL ⟨≈≈⟩ proj₁ leftKernel) (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ commutes) ⟩
```

```
      F ⊔ ⊥ ⨾ G
    □
κ-⨾-Ð-L      :    {D : Obj} {F : Mor A D} {G : Mor B D} → κ ⨾ (F Ð G) ≈ G ⊔ ⊥ ˘ ⨾ F
κ-⨾-Ð-L      {_} {F} {G} = ≈-begin
      κ ⨾ (F Ð G)
    ≈⟨ ⨾-⊔-distribR ⟩
      κ ⨾ ι ˘ ⨾ F ⊔ κ ⨾ κ ˘ ⨾ G
    ≈⟨ ⊔-cong      (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (˘-involutionRightConv ⟨≈˘≈⟩ ˘-cong commutes))
                   (⨾-assocL ⟨≈≈⟩ proj₁ rightKernel) ⟩
      ⊥ ˘ ⨾ F ⊔ G
    ≈⟨ ⊔-commutative ⟩
      G ⊔ ⊥ ˘ ⨾ F
    □

ι-⨾-Ð-LU   :  {D : Obj} {F : Mor A D} {G : Mor B D} → isUnivalent G → ι ⨾ (F Ð G) ≈ F
ι-⨾-Ð-LU   {_} {F} {G} univalG = ≈-begin
      ι ⨾ (F Ð G)
    ≈⟨ ι-⨾-Ð-L ⟩
      F ⊔ ⊥ ⨾ G
    ≈⟨ ⊑-to-⊔₁ (⊥-⨾-unival is-⊥ univalG F) ⟩
      F
    □
κ-⨾-Ð-LU   :  {D : Obj} {F : Mor A D} {G : Mor B D} → isUnivalent F → κ ⨾ (F Ð G) ≈ G
κ-⨾-Ð-LU   {_} {F} {G} univalF = ≈-begin
      κ ⨾ (F Ð G)
    ≈⟨ κ-⨾-Ð-L ⟩
      G ⊔ ⊥ ˘ ⨾ F
    ≈⟨ ⊑-to-⊔₁ (⊥-⨾-unival (⊥-˘ is-⊥) univalF G) ⟩
      G
    □

univalÐ˘⨾Ð        : {D : Obj} {F : Mor A D} {G : Mor B D}
                  → isUnivalent F → isUnivalent G → (F Ð G) ˘ ⨾ (F Ð G) ≈ F ˘ ⨾ F ⊔ G ˘ ⨾ G
univalÐ˘⨾Ð {_} {F} {G} univalF univalG = ≈-begin
      (ι ˘ ⨾ F ⊔ κ ˘ ⨾ G) ˘ ⨾ (F Ð G)
    ≈⟨ ⨾-cong₁ (˘-⊔-distrib ⟨≈≈⟩ ⊔-cong ˘-involutionLeftConv ˘-involutionLeftConv) ⟩
      (F ˘ ⨾ ι ⊔ G ˘ ⨾ κ) ⨾ (F Ð G)
    ≈⟨ ⨾-⊔-distribL ⟨≈≈⟩ ⊔-cong ⨾-assoc ⨾-assoc ⟩
      F ˘ ⨾ ι ⨾ (F Ð G)      ⊔ G ˘ ⨾ κ ⨾ (F Ð G)
    ≈⟨ ⊔-cong (⨾-cong₂ (ι-⨾-Ð-LU univalG)) (⨾-cong₂ (κ-⨾-Ð-LU univalF)) ⟩
      F ˘ ⨾ F      ⊔ G ˘ ⨾ G
    □
Ð-isUnivalent       : {D : Obj} {F : Mor A D} {G : Mor B D}
                    → isUnivalent F → isUnivalent G → isUnivalent (F Ð G)
Ð-isUnivalent univalF univalG = ≈-isSubidentity (univalÐ˘⨾Ð univalF univalG) (⊔-isSubidentity univalF univalG)

Ð-isTotal       :  {D : Obj} {F : Mor A D} {G : Mor B D}
                  → isTotal F → isTotal G → isTotal (F Ð G)
Ð-isTotal       {_} {F} {G} totalF totalG = ⊑-isSuperidentity (⊑-begin
      ι ˘ ⨾ ι ⊔ κ ˘ ⨾ κ
    ⊑⟨ ⊔-monotone (⨾-monotone₂ (proj₁ totalF ⟨⊑≈⟩ ⨾-assoc)) (⨾-monotone₂ (proj₁ totalG ⟨⊑≈⟩ ⨾-assoc)) ⟩
      ι ˘ ⨾ F ⨾ F ˘ ⨾ ι ⊔ κ ˘ ⨾ G ⨾ G ˘ ⨾ κ
    ⊑⟨ ⊔-cong ⨾-assocL ⨾-assocL ⟨≈⊑⟩ ⊔-⨾-⊔-par ⟩
      (F Ð G) ⨾ ((F ˘ ⨾ ι) ⊔ (G ˘ ⨾ κ))
    ≈˘⟨ ⨾-cong₂ (˘-⊔-distrib ⟨≈≈⟩ ⊔-cong ˘-involutionLeftConv ˘-involutionLeftConv) ⟩
      (F Ð G) ⨾ (F Ð G) ˘
    □
  ) (isIdentity-super jointId)
```

```
infixr 5 _⨭_
_⨭_ : {Z : Obj} (F : Mor Z A) (G : Mor Z B) → Mor Z S
F ⨭ G = F ⨾ ι ⊔ G ⨾ κ

⨭-cong   :   {Z : Obj} {F₁ F₂ : Mor Z A} {G₁ G₂ : Mor Z B}
             → F₁ ≈ F₂ → G₁ ≈ G₂ → F₁ ⨭ G₁ ≈ F₂ ⨭ G₂
⨭-cong F₁≈F₂ G₁≈G₂ = ⊔-cong (⨾-cong₁ F₁≈F₂) (⨾-cong₁ G₁≈G₂)

⨭-cong₁   :   {Z : Obj} {F₁ F₂ : Mor Z A} {G : Mor Z B} → F₁ ≈ F₂
              → F₁ ⨭ G ≈ F₂ ⨭ G
⨭-cong₁ F₁≈F₂ =   ⨭-cong F₁≈F₂ ≈-refl

⨭-cong₂   :   {Z : Obj} {F : Mor Z A} {G₁ G₂ : Mor Z B} → G₁ ≈ G₂
              → F ⨭ G₁ ≈ F ⨭ G₂
⨭-cong₂ G₁≈G₂ =   ⨭-cong ≈-refl G₁≈G₂

⨭-monotone   : {Z : Obj} {F₁ F₂ : Mor Z A} {G₁ G₂ : Mor Z B}
             → F₁ ⊑ F₂ → G₁ ⊑ G₂ → F₁ ⨭ G₁ ⊑ F₂ ⨭ G₂
⨭-monotone F₁⊑F₂ G₁⊑G₂ = ⊔-monotone (⨾-monotone₁ F₁⊑F₂) (⨾-monotone₁ G₁⊑G₂)

⨭-monotone₁ :   {Z : Obj} {F₁ F₂ : Mor Z A} {G : Mor Z B} → F₁ ⊑ F₂
              → F₁ ⨭ G ⊑ F₂ ⨭ G
⨭-monotone₁ F₁⊑F₂ = ⨭-monotone F₁⊑F₂ ⊑-refl

⨭-monotone₂ :   {Z : Obj} {F : Mor Z A} {G₁ G₂ : Mor Z B} → G₁ ⊑ G₂
              → F ⨭ G₁ ⊑ F ⨭ G₂
⨭-monotone₂ G₁⊑G₂ = ⨭-monotone ⊑-refl G₁⊑G₂

⨭-⊔-⨭ :     {Z : Obj} {F₁ F₂ : Mor Z A} {G₁ G₂ : Mor Z B}
         → (F₁ ⨭ G₁) ⊔ (F₂ ⨭ G₂) ≈ (F₁ ⊔ F₂) ⨭ (G₁ ⊔ G₂)
⨭-⊔-⨭ {F₁ = F₁} {F₂} {G₁} {G₂} =   ≈-begin
           (F₁ ⨭ G₁) ⊔ (F₂ ⨭ G₂)
         ≈⟨ ⊔-transpose₂ ⟩
           ((F₁ ⨾ ι) ⊔ (F₂ ⨾ ι)) ⊔ ((G₁ ⨾ κ) ⊔ (G₂ ⨾ κ))
         ≈˘⟨ ⊔-cong ⨾-⊔-distribL ⨾-⊔-distribL ⟩
           (F₁ ⊔ F₂) ⨭ (G₁ ⊔ G₂)
         □

⨾-⨭ :     {Y Z : Obj} {F : Mor Y Z} {G₁ : Mor Z A} {G₂ : Mor Z B}
         → F ⨾ (G₁ ⨭ G₂)     ≈ F ⨾ G₁ ⨭ F ⨾ G₂
⨾-⨭ {F = F} {G₁} {G₂} = ≈-begin
           F ⨾ (G₁ ⨭ G₂)
         ≈⟨ ⨾-⊔-distribR ⟩
           F ⨾ G₁ ⨾ ι ⊔ F ⨾ G₂ ⨾ κ
         ≈⟨ ⊔-cong ⨾-assocL ⨾-assocL ⟩
           F ⨾ G₁ ⨭ F ⨾ G₂
         □

⨾-⨭-⊑    : {Y Z : Obj} {F : Mor Y Z} {G₁ : Mor Z A} {G₂ : Mor Z B} {H : Mor Y S}
         →      F ⨾ G₁ ⨾ ι ⊑ H → F ⨾ G₂ ⨾ κ ⊑ H → F ⨾ (G₁ ⨭ G₂) ⊑ H
⨾-⨭-⊑ {F = F} {G₁} {G₂} {H} F⨾G₁⨾ι⊑H F⨾G₂⨾κ⊑H = ⊑-begin
           F ⨾ (G₁ ⨭ G₂)
         ≈⟨ ⨾-⊔-distribR ⟩
           F ⨾ G₁ ⨾ ι ⊔ F ⨾ G₂ ⨾ κ
         ⊑⟨ ⊔-universal F⨾G₁⨾ι⊑H F⨾G₂⨾κ⊑H ⟩
           H
         □

⨾-⨭-⊑₁    :    {Y Z : Obj} {F : Mor Y Z} {G₁ : Mor Z A} {G₂ : Mor Z B} {H : Mor Y S}
         →      F ⨾ (G₁ ⨭ G₂) ⊑ H → F ⨾ G₁ ⨾ ι ⊑ H
⨾-⨭-⊑₁ {F = F} {G₁} {G₂} {H} F⨾G⊑H = ⊑-begin
           F ⨾ G₁ ⨾ ι
         ⊑⟨ ⊔-upper₁ ⟩
           F ⨾ G₁ ⨾ ι ⊔ F ⨾ G₂ ⨾ κ
         ≈˘⟨ ⨾-⊔-distribR ⟩
           F ⨾ (G₁ ⨭ G₂)
```

```
        ⊑⟨ F⨾G⊑H ⟩
          H
        □

  ⨾-⊕-⊑₂    :    {Y Z : Obj} {F : Mor Y Z} {G₁ : Mor Z A} {G₂ : Mor Z B} {H : Mor Y S}
            →    F ⨾ (G₁ ⊕ G₂) ⊑ H → F ⨾ G₂ ⨾ κ ⊑ H
  ⨾-⊕-⊑₂ {F = F} {G₁} {G₂} {H} F⨾G⊑H = ⊑-begin
          F ⨾ G₂ ⨾ κ
        ⊑⟨ ⊔-upper₂ ⟩
          F ⨾ G₁ ⨾ ι ⊔ F ⨾ G₂ ⨾ κ
        ≈˘⟨ ⨾-⊔-distribR ⟩
          F ⨾ (G₁ ⊕ G₂)
        ⊑⟨ F⨾G⊑H ⟩
          H
        □

  to-⊕ : {C : Obj} {H : Mor C S} → H ≈ H ⨾ ι ˘ ⊕ H ⨾ κ ˘
  to-⊕ {_} {H} = ≈-begin
          H
        ≈⟨ proj₂ jointId ⟨≈˘≈⟩ ⨾-⊔-distribR ⟩
          H ⨾ (ι ˘ ⨾ ι) ⊔ H ⨾ (κ ˘ ⨾ κ)
        ≈⟨ ⊔-cong ⨾-assocL ⨾-assocL ⟩
          H ⨾ ι ˘ ⊕ H ⨾ κ ˘
        □

  ⊕-˘ :    {Z : Obj} {F : Mor Z A} {G : Mor Z B} → (F ⊕ G) ˘ ≈ F ˘ ⊞ G ˘
  ⊕-˘ {_} {F} {G} = ≈-begin
          (F ⊕ G) ˘
        ≈⟨ ˘-⊔-distrib ⟩
          (F ⨾ ι) ˘ ⊔ (G ⨾ κ) ˘
        ≈⟨ ⊔-cong ˘-involution ˘-involution ⟩
          F ˘ ⊞ G ˘
        □

  ⊞-˘ :    {C : Obj} {F : Mor A C} {G : Mor B C} → (F ⊞ G) ˘ ≈ F ˘ ⊕ G ˘
  ⊞-˘ {_} {F} {G} = ≈-begin
          (F ⊞ G) ˘
        ≈⟨ ˘-⊔-distrib ⟩
          (ι ˘ ⨾ F) ˘ ⊔ (κ ˘ ⨾ G) ˘
        ≈⟨ ⊔-cong ˘-involutionLeftConv ˘-involutionLeftConv ⟩
          F ˘ ⊕ G ˘
        □


module IsDirectSum {A B S : Obj}
    {⊥ : Mor A B} (is-⊥ : isLeastMor ⊥)
    (⊥⨾-is-⊥ : {C : Obj} {R : Mor B C} → isLeastMor (⊥ ⨾ R))
    (⊥˘⨾-is-⊥ : {C : Obj} {S : Mor A C} → isLeastMor (⊥ ˘ ⨾ S))
    {ι : Mor A S} {κ : Mor B S}
    (sum : IsDirectSum-L is-⊥ ι κ)
  where
  open IsDirectSum-L sum public


  ι-⨾-⊞     : {D : Obj} {F : Mor A D} {G : Mor B D} → ι ⨾ (F ⊞ G) ≈ F
  ι-⨾-⊞     {F = F} {G} = ≈-begin
        ι ⨾ (F ⊞ G)
      ≈⟨ ι-⨾-⊞-L ⟩
        F ⊔ ⊥ ⨾ G
      ≈⟨ ⊑-to-⊔₁ (⊥⨾-is-⊥ F) ⟩
        F
      □
  κ-⨾-⊞     : {D : Obj} {F : Mor A D} {G : Mor B D} → κ ⨾ (F ⊞ G) ≈ G
```

```
κ-⨾-⊞    {F = F} {G} = ≈-begin
       κ ⨾ (F ⊞ G)
     ≈⟨ κ-⨾-⊞-L ⟩
       G ⊔ ⊥ ˘ ⨾ F
     ≈⟨ ⊑-to-⊔₁ (⊥˘⨾-is-⊥ G)     ⟩
       G
     □


⊕-⨾-ι˘     :    {Z : Obj} {F : Mor Z A} {G : Mor Z B} → (F ⊕ G) ⨾ ι ˘ ≈ F
⊕-⨾-ι˘     {F = F} {G} = ≈-begin
       (F ⊕ G) ⨾ ι ˘
     ≈⟨ ˘-involutionRightConv ⟨≈˘≈⟩ ˘-cong (⨾-cong₂ ⊕-˘) ⟩
       (ι ⨾ (F ˘ ⊞ G ˘)) ˘
     ≈⟨ ˘-cong ι-⨾-⊞      ⟨≈≈⟩ ˘˘ ⟩
       F
     □
⊕-⨾-κ˘     :    {Z : Obj} {F : Mor Z A} {G : Mor Z B} → (F ⊕ G) ⨾ κ ˘ ≈ G
⊕-⨾-κ˘     {F = F} {G} = ≈-begin
       (F ⊕ G) ⨾ κ ˘
     ≈⟨ ˘-involutionRightConv ⟨≈˘≈⟩ ˘-cong (⨾-cong₂ ⊕-˘) ⟩
       (κ ⨾ (F ˘ ⊞ G ˘)) ˘
     ≈⟨ ˘-cong κ-⨾-⊞      ⟨≈≈⟩ ˘˘ ⟩
       G
     □


⊞-⊑-⊞₁     :    {C : Obj} {F₁ G₁ : Mor A C} {F₂ G₂ : Mor B C}
            → F₁ ⊞ F₂ ⊑ G₁ ⊞ G₂ → F₁ ⊑ G₁
⊞-⊑-⊞₁ {F₁ = F₁} {G₁} {F₂} {G₂} F⊑G = ⊑-begin
         F₁
       ≈˘⟨ ι-⨾-⊞        ⟩
         ι ⨾ (F₁ ⊞ F₂)
       ⊑⟨ ⨾-monotone₂ F⊑G ⟩
         ι ⨾ (G₁ ⊞ G₂)
       ≈⟨ ι-⨾-⊞    ⟩
         G₁
       □
⊞-⊑-⊞₂     :    {C : Obj} {F₁ G₁ : Mor A C} {F₂ G₂ : Mor B C}
            → F₁ ⊞ F₂ ⊑ G₁ ⊞ G₂ → F₂ ⊑ G₂
⊞-⊑-⊞₂ {F₁ = F₁} {G₁} {F₂} {G₂} F⊑G = ⊑-begin
         F₂
       ≈˘⟨ κ-⨾-⊞        ⟩
         κ ⨾ (F₁ ⊞ F₂)
       ⊑⟨ ⨾-monotone₂ F⊑G ⟩
         κ ⨾ (G₁ ⊞ G₂)
       ≈⟨ κ-⨾-⊞    ⟩
         G₂
       □


⊕-⊑-⊕₁     :    {C : Obj} {F₁ G₁ : Mor C A} {F₂ G₂ : Mor C B}
            → F₁ ⊕ F₂ ⊑ G₁ ⊕ G₂ → F₁ ⊑ G₁
⊕-⊑-⊕₁ {F₁ = F₁} {G₁} {F₂} {G₂} F⊑G = ⊑-begin
         F₁
       ≈˘⟨ ⊕-⨾-ι˘ ⟩
         (F₁ ⊕ F₂) ⨾ ι ˘
       ⊑⟨ ⨾-monotone₁ F⊑G ⟩
         (G₁ ⊕ G₂) ⨾ ι ˘
       ≈⟨ ⊕-⨾-ι˘ ⟩
```

```
          G₁
        □
  ⋲-⊑-⋲₂   :   {C : Obj} {F₁ G₁ : Mor C A} {F₂ G₂ : Mor C B}
           → F₁ ⋲ F₂ ⊑ G₁ ⋲ G₂ → F₂ ⊑ G₂
  ⋲-⊑-⋲₂ {F₁ = F₁} {G₁} {F₂} {G₂} F⊑G = ⊑-begin
          F₂
        ≈˘⟨ ⋲-⨾-κ˘ ⟩
          (F₁ ⋲ F₂) ⨾ κ ˘
        ⊑⟨ ⨾-monotone₁ F⊑G ⟩
          (G₁ ⋲ G₂) ⨾ κ ˘
        ≈⟨ ⋲-⨾-κ˘ ⟩
          G₂
        □

  ⋲-⨾-⋺   : {Z C : Obj} {F₁ : Mor Z A} {F₂ : Mor Z B} {G₁ : Mor A C} {G₂ : Mor B C}
           → (F₁ ⋲ F₂) ⨾ (G₁ ⋺ G₂) ≈ F₁ ⨾ G₁ ⊔ F₂ ⨾ G₂
  ⋲-⨾-⋺ {F₁ = F₁} {F₂} {G₁} {G₂} = ≈-begin
        (F₁ ⋲ F₂) ⨾ (G₁ ⋺ G₂)
      ≈⟨ ⨾-⊔-distribL ⟩
        (F₁ ⨾ ι) ⨾ (G₁ ⋺ G₂) ⊔ (F₂ ⨾ κ) ⨾ (G₁ ⋺ G₂)
      ≈⟨ ⊔-cong (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ ι-⨾-⋺) (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ κ-⨾-⋺) ⟩
        F₁ ⨾ G₁ ⊔ F₂ ⨾ G₂
      □

module IsDirectSum-B  (botMor : BotMor orderedSemigroupoid)
    {A B S : Obj} {ι : Mor A S} {κ : Mor B S}
    (sum : IsDirectSum-L (BotMor.is-⊥ botMor) ι κ) where
  open BotMor botMor
  open IsDirectSum-L sum public
  open OSGC-BotMor botMor
  commutes˘ : κ ⨾ ι ˘ ≈ ⊥ {B} {A}
  commutes˘ = leastMor-≈-⊥ κ⨾ι˘-is-⊥

module IsDirectSum²-B  (botMor : BotMor orderedSemigroupoid)
    {A₁ B₁ S₁ A₂ B₂ S₂ : Obj}
    {ι₁ : Mor A₁ S₁} {κ₁ : Mor B₁ S₁} (sum₁ : IsDirectSum-L (BotMor.is-⊥ botMor) ι₁ κ₁)
    {ι₂ : Mor A₂ S₂} {κ₂ : Mor B₂ S₂} (sum₂ : IsDirectSum-L (BotMor.is-⊥ botMor) ι₂ κ₂) where
  open IsDirectSum-B botMor sum₁ using () renaming (_⋲_ to _⋲₁_; _⋺_ to _⋺₁_; ⋺-cong to ⋺₁-cong)
  open IsDirectSum-B botMor sum₂ using () renaming (_⋲_ to _⋲₂_; _⋺_ to _⋺₂_)

  ⋲-⋺ : {F₁₁ : Mor A₁ A₂} {F₁₂ : Mor A₁ B₂} {F₂₁ : Mor B₁ A₂} {F₂₂ : Mor B₁ B₂}
      → (F₁₁ ⋲₂ F₁₂) ⋺₁ (F₂₁ ⋲₂ F₂₂) ≈ (F₁₁ ⋺₁ F₂₁) ⋲₂ (F₁₂ ⋺₁ F₂₂)
  ⋲-⋺ {F₁₁ = F₁₁} {F₁₂} {F₂₁} {F₂₂} = ≈-begin
          (F₁₁ ⋲₂ F₁₂) ⋺₁ (F₂₁ ⋲₂ F₂₂)
      ≈⟨ ⊔-cong ⨾-⊔-distribR ⨾-⊔-distribR ⟨≈≈⟩ ⊔-transpose₂ ⟩
          (ι₁ ˘ ⨾ F₁₁ ⨾ ι₂ ⊔ κ₁ ˘ ⨾ F₂₁ ⨾ ι₂) ⊔ (ι₁ ˘ ⨾ F₁₂ ⨾ κ₂ ⊔ κ₁ ˘ ⨾ F₂₂ ⨾ κ₂)
      ≈⟨ ⊔-cong (⊔-cong ⨾-assocL ⨾-assocL ⟨≈≈˘⟩ ⨾-⊔-distribL) (⊔-cong ⨾-assocL ⨾-assocL ⟨≈≈˘⟩ ⨾-⊔-distribL) ⟩
          (F₁₁ ⋺₁ F₂₁) ⋲₂ (F₁₂ ⋺₁ F₂₂)
      □

  infixr 5 _⊕_
  _⊕_ : (F : Mor A₁ A₂) (G : Mor B₁ B₂) → Mor S₁ S₂
  F ⊕ G = F ⨾ ι₂ ⋺₁ G ⨾ κ₂

  ⊕-cong   :   {F₁ F₂ : Mor A₁ A₂} {G₁ G₂ : Mor B₁ B₂}
             → F₁ ≈ F₂ → G₁ ≈ G₂ →  F₁ ⊕ G₁ ≈ F₂ ⊕ G₂
```

```
⊕-cong F₁≈F₂ G₁≈G₂ = ⅅ₁-cong (⨾-cong₁ F₁≈F₂) (⨾-cong₁ G₁≈G₂)
⊕-cong₁     :    {F₁ F₂ : Mor A₁ A₂} {G : Mor B₁ B₂}
               → F₁ ≈ F₂ →    F₁ ⊕ G ≈ F₂ ⊕ G
⊕-cong₁ F₁≈F₂ = ⊕-cong F₁≈F₂ ≈-refl
⊕-cong₂     :    {F : Mor A₁ A₂} {G₁ G₂ : Mor B₁ B₂}
               → G₁ ≈ G₂ →    F ⊕ G₁ ≈ F ⊕ G₂
⊕-cong₂ G₁≈G₂ = ⊕-cong ≈-refl G₁≈G₂
```

```
module IsDirectSum-Z   (zeroMor : ZeroMor orderedSemigroupoid)
    {A B S : Obj} {ι : Mor A S} {κ : Mor B S}
    (sum : IsDirectSum-L (ZeroMor.is-⊥ zeroMor) ι κ) where
  open ZeroMor zeroMor
  open IsDirectSum-B botMor sum public using (commutes˘)
  open IsDirectSum is-⊥ ⊥⨾-is-⊥ (is-⊥-⨾ (⊥-˘ is-⊥)) sum public
```

```
module IsDirectSum²-Z   (zeroMor : ZeroMor orderedSemigroupoid)
    {A₁ B₁ S₁ A₂ B₂ S₂ : Obj}
    {ι₁ : Mor A₁ S₁} {κ₁ : Mor B₁ S₁} (sum₁ : IsDirectSum-L (ZeroMor.is-⊥ zeroMor) ι₁ κ₁)
    {ι₂ : Mor A₂ S₂} {κ₂ : Mor B₂ S₂} (sum₂ : IsDirectSum-L (ZeroMor.is-⊥ zeroMor) ι₂ κ₂) where
  open ZeroMor zeroMor
  open IsDirectSum-Z zeroMor sum₁ using () renaming (ι-⨾-ⅅ to ι-⨾-ⅅ₁ ; κ-⨾-ⅅ to κ-⨾-ⅅ₁)
  open IsDirectSum²-B botMor sum₁ sum₂ public
```

```
  ι-⨾-⊕ : {F : Mor A₁ A₂} {G : Mor B₁ B₂} → ι₁ ⨾ (F ⊕ G) ≈ F ⨾ ι₂
  ι-⨾-⊕ = ι-⨾-ⅅ₁
  κ-⨾-⊕ : {F : Mor A₁ A₂} {G : Mor B₁ B₂} → κ₁ ⨾ (F ⊕ G) ≈ G ⨾ κ₂
  κ-⨾-⊕ = κ-⨾-ⅅ₁
```

```
record DirectSum-L (A B : Obj) {⊥ : Mor A B} (is-⊥ : isLeastMor ⊥) : Set (i ⊍ j ⊍ k₁) where
  field
    obj : Obj
    ι : Mor A obj
    κ : Mor B obj
    isDirectSum : IsDirectSum-L is-⊥ ι κ
  open IsDirectSum-L isDirectSum public
```

```
DirectSum-Z : (zeroMor : ZeroMor orderedSemigroupoid) (A B : Obj) → Set (i ⊍ j ⊍ k₁)
DirectSum-Z zeroMor A B = DirectSum-L A B (ZeroMor.is-⊥ zeroMor)
module DirectSum-Z (zeroMor : ZeroMor orderedSemigroupoid) {A B : Obj} (S : DirectSum-Z zeroMor A B) where
  open DirectSum-L S public using (obj; ι; κ; isDirectSum)
  open IsDirectSum-Z zeroMor isDirectSum public
```

The suffic “-L” stands for “in the context of assuming only Least morphisms” (as opposed to assuming zero morphisms).

```
record HasDirectSum-L (botMor : BotMor orderedSemigroupoid) : Set (i ⊍ j ⊍ k₁ ⊍ k₂) where
  open BotMor botMor
  open OSGC-BotMor botMor
  field
    _⊞_ : Obj → Obj → Obj
    ι : {A B : Obj} → Mor A (A ⊞ B)
    κ : {A B : Obj} → Mor B (A ⊞ B)
    dirSum : (A B : Obj) → IsDirectSum-L {A} {B} {A ⊞ B} is-⊥ ι κ
  private
    module Local {A B : Obj} where
```

```
      open IsDirectSum-B botMor (dirSum A B) public
    module Local² {A₁ B₁ A₂ B₂ : Obj} where
      open IsDirectSum²-B botMor (dirSum A₁ B₁) (dirSum A₂ B₂) public
  open Local public
  open Local² public
```

The suffic "-L" stands for "in the context of assuming only Least morphisms" (as opposed to assuming zero morphisms).

```
 module HasDirectSum (zeroMor : ZeroMor orderedSemigroupoid)
                     (hasSum : HasDirectSum-L (ZeroMor.botMor zeroMor))
  where
  open ZeroMor zeroMor
  open HasDirectSum-L hasSum public using
    (_⊞_; ι;      κ; dirSum; commutes˘; ⊕-⊕
    ; _⊕_;    ⊕-cong;  ⊕-cong₁;  ⊕-cong₂)
  private
    module Local {A B : Obj} where
      open IsDirectSum is-⊥ ⊥⨾-is-⊥ (is-⊥-⨾ (⊥-˘ is-⊥)) (dirSum A B) public
    module Local² {A₁ B₁ A₂ B₂ : Obj} where
      open IsDirectSum²-Z zeroMor (dirSum A₁ B₁) (dirSum A₂ B₂) public using (ι-⨾-⊕; κ-⨾-⊕)
  open Local public
  open Local² public
```

15.2 Categoric.KleeneCategory.DirectSum

We generalise the construction of the star operator for 2 ×2-matrices presented for example by Kozen (1994a) to direct sums in USLCCs with zero morphisms.

```
module Categoric.KleeneCategory.DirectSum {ℓi ℓj ℓk₁ ℓk₂ : Level} {Obj : Set ℓi}
      (base : USLCC ℓj ℓk₁ ℓk₂ Obj)
      (zeroMor : ZeroMor (USLCC.orderedSemigroupoid base))
where
open USLCC base
open ZeroMor zeroMor
open OSGC-BotMor osgc botMor
open Categoric.KleeneCategory.InUSLCat uslCategory
open Categoric.KCC.InUSLCC base
open Categoric.DirectSum base


module SumStar {A B : Obj}
               (Sum : DirectSum-Z zeroMor A B)
               (StarA : LocalStarOp A)
               (StarB : LocalStarOp B)
 where
 open DirectSum-Z zeroMor Sum renaming (obj to A⊞B)
 open IsDirectSum²-Z zeroMor isDirectSum isDirectSum
 open LocalStarOp StarA using () renaming (_* to _*A)
 open LocalStarOp StarB using () renaming (_* to _*B)
 open LocalStarOp
 module Square (a : Mor A A) (b : Mor A B) (c : Mor B A) (d : Mor B B) where
   E : Mor A⊞B A⊞B
   E = (a ⊅ c) ⊄ (b ⊅ d)
   E′ : Mor A⊞B A⊞B
```

```
E′ = (a ⨭ b) ⨮ (c ⨭ d)

E≈′ : E ≈ E′
E≈′ = ≈-sym ⨭-⨮

ι-⨾-E : ι ⨾ E ≈ a ⨭ b
ι-⨾-E = ≈-begin
        ι ⨾ E
      ≈⟨ ⨾-cong₂ E≈′ ⟩
        ι ⨾ E′
      ≈⟨ ι-⨾-⨮ ⟩
        a ⨭ b
      □

d* : Mor B B
d* = d *B
f : Mor A A
f = a ⊔ b ⨾ d* ⨾ c
f* : Mor A A
f* = f *A
g : Mor A B
g = f* ⨾ b ⨾ d*
h : Mor B A
h = d* ⨾ c ⨾ f*
j : Mor B B
j = d* ⨾ c ⨾ g
k : Mor B B
k = d* ⊔ j

E*  : Mor A⊞B A⊞B
E* = (f* ⨮ h) ⨭ (g ⨮ k)


g′ : Mor A B
g′ = b ⨾ d*
h′ : Mor B A
h′ = d* ⨾ c
k′ : Mor B B
k′ = d* ⊔ h′ ⨾ g′
E*′ : Mor A⊞B A⊞B
E*′ = (Id ⨮ h′) ⨭ (g′ ⨮ k′)
E*-1 : f* ≈ Id → E* ≈ E*′
E*-1 f*≈Id = ⨭-cong (⨮-cong f*≈Id h≈h′) (⨮-cong g≈g′ k≈k′)
  where
    g≈g′ : g ≈ g′
    g≈g′ = ⨾-cong₁ f*≈Id ⟨≈≈⟩ leftId
    h≈h′ : h ≈ h′
    h≈h′ = ⨾-cong₂ (⨾-cong₂ f*≈Id ⟨≈≈⟩ rightId)
    k≈k′ : k ≈ k′
    k≈k′ = ⊔-cong₂ (⨾-cong₂₂ g≈g′ ⟨≈≈⟩ ⨾-assocL)


module LeftInd {C : Obj} {x : Mor A C} {y : Mor B C}
               (E⨾xy⊑xy : E ⨾ (x ⨮ y) ⊑ x ⨮ y)
  where


  E⨾xy′⊑xy : (a ⨮ c) ⨾ x ⊔ (b ⨮ d) ⨾ y ⊑ x ⨮ y
  E⨾xy′⊑xy = ⊑-begin
               (a ⨮ c) ⨾ x ⊔ (b ⨮ d) ⨾ y
    ≈˘⟨ ⨭-⨾-⨮ ⟩
               E ⨾ (x ⨮ y)
    ⊑⟨ E⨾xy⊑xy ⟩
```

```
          x ⊕ y
    □
  ac⨾x⊑xy : (a ⊕ c) ⨾ x ⊑ x ⊕ y
  ac⨾x⊑xy = ⊔-upper₁ ⟨⊑⊑⟩ E⨾xy′⊑xy
  bd⨾y⊑xy : (b ⊕ d) ⨾ y ⊑ x ⊕ y
  bd⨾y⊑xy = ⊔-upper₂ ⟨⊑⊑⟩ E⨾xy′⊑xy
  a⨾x⊑x : a ⨾ x ⊑ x
  a⨾x⊑x = ⊕-⊑-⊕₁ (⊕-⨾ ⟨≈˘⊑⟩ ac⨾x⊑xy)
  c⨾x⊑y : c ⨾ x ⊑ y
  c⨾x⊑y = ⊕-⊑-⊕₂ (⊕-⨾ ⟨≈˘⊑⟩ ac⨾x⊑xy)
  b⨾y⊑x : b ⨾ y ⊑ x
  b⨾y⊑x = ⊕-⊑-⊕₁ (⊕-⨾ ⟨≈˘⊑⟩ bd⨾y⊑xy)
  d⨾y⊑y : d ⨾ y ⊑ y
  d⨾y⊑y = ⊕-⊑-⊕₂ (⊕-⨾ ⟨≈˘⊑⟩ bd⨾y⊑xy)

  d*⨾y⊑y : d* ⨾ y ⊑ y
  d*⨾y⊑y = *-leftInd StarB d⨾y⊑y
  b⨾d*⨾y⊑x : b ⨾ d* ⨾ y ⊑ x
  b⨾d*⨾y⊑x = ⨾-monotone₂ d*⨾y⊑y ⟨⊑⊑⟩ b⨾y⊑x
  f⨾x⊑x : f ⨾ x ⊑ x
  f⨾x⊑x = ⊑-begin
        (a ⊔ b ⨾ d* ⨾ c) ⨾ x
    ≈⟨ ⨾-⊔-distribL ⟨≈≈⟩ ⊔-cong₂ ⨾-assoc₃₊₁ ⟩
        a ⨾ x ⊔ b ⨾ d* ⨾ c ⨾ x
    ⊑⟨ ⊔-universal a⨾x⊑x (⨾-monotone₂₂ c⨾x⊑y ⟨⊑⊑⟩ b⨾d*⨾y⊑x) ⟩
        x
    □
  f*⨾x⊑x : f* ⨾ x ⊑ x
  f*⨾x⊑x = *-leftInd StarA f⨾x⊑x
  *-leftInd⊕₁ : (f* ⊕ (d* ⨾ c ⨾ f*)) ⨾ x ⊑ x ⊕ y
  *-leftInd⊕₁ =   ⊕-⨾ ⟨≈⊑⟩     ⊕-monotone f*⨾x⊑x
                                (⨾-assoc₃₊₁ ⟨≈⊑⟩ ⨾-monotone₂₂ f*⨾x⊑x ⟨⊑⊑⟩ ⨾-monotone₂ c⨾x⊑y ⟨⊑⊑⟩ d*⨾y⊑y)

  g⨾y⊑x : g ⨾ y ⊑ x
  g⨾y⊑x = ⊑-begin
        (f* ⨾ b ⨾ d*) ⨾ y
    ⊑⟨ ⨾-assoc₃₊₁ ⟨≈⊑⟩ ⨾-monotone₂ b⨾d*⨾y⊑x ⟨⊑⊑⟩ f*⨾x⊑x ⟩
        x
    □
  j⨾y⊑y : j ⨾ y ⊑ y
  j⨾y⊑y = ⊑-begin
        (d* ⨾ c ⨾ g) ⨾ y
    ⊑⟨ ⨾-assoc₃₊₁ ⟨≈⊑⟩ ⨾-monotone₂₂ g⨾y⊑x ⟨⊑⊑⟩ ⨾-monotone₂ c⨾x⊑y ⟨⊑⊑⟩ d*⨾y⊑y ⟩
        y
    □
  *-leftInd⊕₂ : (g ⊕ (d* ⊔ j)) ⨾ y ⊑ x ⊕ y
  *-leftInd⊕₂ = ⊕-⨾ ⟨≈⊑⟩ ⊕-monotone g⨾y⊑x (⨾-⊔-distribL ⟨≈⊑⟩ ⊔-universal d*⨾y⊑y j⨾y⊑y)

  *-leftInd⊕ : E* ⨾ (x ⊕ y) ⊑ (x ⊕ y)
  *-leftInd⊕ = ⊑-begin
          E* ⨾ (x ⊕ y)
      ≈⟨ ⊕-⨾-⊕ ⟩
          (f* ⊕ h) ⨾ x    ⊔ (g ⊕ k) ⨾ y
      ⊑⟨ ⊔-universal *-leftInd⊕₁ *-leftInd⊕₂ ⟩
          x ⊕ y
      □

E*-leftInd : {C : Obj} {S : Mor A⊞B C} → E ⨾ S ⊑ S → E* ⨾ S ⊑ S
E*-leftInd {_} {S} E⨾S⊑S = ⊑-begin
```

```
      E* ⨾ S
    ≈⟨ ⨾-cong₂ to-⊕ ⟩
      E* ⨾ (ι ⨾ S ⊕ κ ⨾ S)
    ⊑⟨ *-leftInd⊕ ⟩
      ι ⨾ S ⊕ κ ⨾ S
    ≈˘⟨ to-⊕ ⟩
      S
    □
  where
    E⨾xy⊑xy : E ⨾ (ι ⨾ S ⊕ κ ⨾ S) ⊑ (ι ⨾ S ⊕ κ ⨾ S)
    E⨾xy⊑xy = ⊑-begin
            E ⨾ (ι ⨾ S ⊕ κ ⨾ S)
      ≈˘⟨ ⨾-cong₂ to-⊕ ⟩
            E ⨾ S
      ⊑⟨ E⨾S⊑S ⟩
            S
      ≈⟨ to-⊕ ⟩
            ι ⨾ S ⊕ κ ⨾ S
      □
    open LeftInd E⨾xy⊑xy      using (*-leftInd⊕)


module RightInd  {Z : Obj} {x : Mor Z A} {y : Mor Z B}
                 (xy⨾E⊑xy : (x ⊕ y) ⨾ E ⊑ x ⊕ y)
  where


  xy′⨾E⊑xy : x ⨾ (a ⊕ b) ⊔ y ⨾ (c ⊕ d) ⊑ x ⊕ y
  xy′⨾E⊑xy = ⊑-begin
           x ⨾ (a ⊕ b) ⊔ y ⨾ (c ⊕ d)
    ≈⟨ ⊕-⨾-⊕ ⟨≈˘≈˘⟩ ⨾-cong₂ E≈′ ⟩
           (x ⊕ y) ⨾ E
    ⊑⟨ xy⨾E⊑xy ⟩
           x ⊕ y
    □
  x⨾ab⊑xy : x ⨾ (a ⊕ b) ⊑ x ⊕ y
  x⨾ab⊑xy = ⊔-upper₁ ⟨⊑⊑⟩ xy′⨾E⊑xy
  y⨾cd⊑xy : y ⨾ (c ⊕ d) ⊑ x ⊕ y
  y⨾cd⊑xy = ⊔-upper₂ ⟨⊑⊑⟩ xy′⨾E⊑xy
  x⨾a⊑x : x ⨾ a ⊑ x
  x⨾a⊑x = ⊕-⊑-⊕₁ (⨾-⊕ ⟨≈˘⊑⟩ x⨾ab⊑xy)
  x⨾b⊑y : x ⨾ b ⊑ y
  x⨾b⊑y = ⊕-⊑-⊕₂ (⨾-⊕ ⟨≈˘⊑⟩ x⨾ab⊑xy)
  y⨾c⊑x : y ⨾ c ⊑ x
  y⨾c⊑x = ⊕-⊑-⊕₁ (⨾-⊕ ⟨≈˘⊑⟩ y⨾cd⊑xy)
  y⨾d⊑y : y ⨾ d ⊑ y
  y⨾d⊑y = ⊕-⊑-⊕₂ (⨾-⊕ ⟨≈˘⊑⟩ y⨾cd⊑xy)


  y⨾d*⊑y : y ⨾ d* ⊑ y
  y⨾d*⊑y = *-rightInd StarB y⨾d⊑y
  x⨾b⨾d*⊑y : x ⨾ b ⨾ d* ⊑ y
  x⨾b⨾d*⊑y = ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ x⨾b⊑y ⟨⊑⊑⟩ y⨾d*⊑y
  x⨾f⊑x : x ⨾ f ⊑ x
  x⨾f⊑x = ⊑-begin
      x ⨾ (a ⊔ b ⨾ d* ⨾ c)
    ≈⟨ ⨾-⊔-distribR ⟨≈≈˘⟩ ⊔-cong₂ ⨾-assoc₃₊₁ ⟩
      x ⨾ a ⊔ (x ⨾ b ⨾ d*) ⨾ c
    ⊑⟨ ⊔-universal x⨾a⊑x (⨾-monotone₁ x⨾b⨾d*⊑y ⟨⊑⊑⟩ y⨾c⊑x) ⟩
      x
```

```
    □
  x⨾f*⊑x : x ⨾ f* ⊑ x
  x⨾f*⊑x = *-rightInd StarA x⨾f⊑x
  x⨾g⊑y : x ⨾ g ⊑ y
  x⨾g⊑y = ⊑-begin
      x ⨾ (f* ⨾ b ⨾ d*)
    ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ x⨾f*⊑x ⟨⊑⊑⟩ x⨾b⨾d*⊑y ⟩
      y
    □
  y⨾j⊑y : y ⨾ j ⊑ y
  y⨾j⊑y = ⊑-begin
      y ⨾ (d* ⨾ c ⨾ g)
    ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ y⨾d*⊑y ⟩
      y ⨾ (c ⨾ g)
    ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ y⨾c⊑x ⟨⊑⊑⟩ x⨾g⊑y ⟩
      y
    □
  y⨾h⊑x : y ⨾ h ⊑ x
  y⨾h⊑x = ⊑-begin
      y ⨾ (d* ⨾ c ⨾ f*)
    ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ y⨾d*⊑y ⟩
      y ⨾ (c ⨾ f*)
    ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ y⨾c⊑x ⟨⊑⊑⟩ x⨾f*⊑x ⟩
      x
    □


  *-rightInd⨭₁ : x ⨾ (f* ⨭ g) ⊑ x ⨭ y
  *-rightInd⨭₁ =    ⨾-⨭ ⟨≈⊑⟩ ⨭-monotone x⨾f*⊑x x⨾g⊑y
  *-rightInd⨭₂ : y ⨾ (h ⨭ k) ⊑ x ⨭ y
  *-rightInd⨭₂ = ⨾-⨭ ⟨≈⊑⟩ ⨭-monotone y⨾h⊑x (⨾-⊔-distribR ⟨≈⊑⟩ ⊔-universal y⨾d*⊑y y⨾j⊑y)


  *-rightInd⨭ : (x ⨭ y) ⨾ E* ⊑ (x ⨭ y)
  *-rightInd⨭ = ⊑-begin
          (x ⨭ y) ⨾ E*
      ≈⟨ ⨾-cong₂ ⨭-⨮ ⟨≈˘≈⟩ ⨭-⨾-⨮ ⟩
          x ⨾ (f* ⨭ g) ⊔ y ⨾ (h ⨭ k)
      ⊑⟨ ⊔-universal *-rightInd⨭₁ *-rightInd⨭₂ ⟩
          x ⨭ y
      □


E*-rightInd : {C : Obj} {S : Mor C A⊞B} → S ⨾ E ⊑ S → S ⨾ E* ⊑ S
E*-rightInd {_} {S} S⨾E⊑S = ⊑-begin
      S ⨾ E*
    ≈⟨ ⨾-cong₁ to-⨭ ⟩
      (S ⨾ ι ˘ ⨭ S ⨾ κ ˘) ⨾ E*
    ⊑⟨ *-rightInd⨭ ⟩
      S ⨾ ι ˘ ⨭ S ⨾ κ ˘
    ≈˘⟨ to-⨭ ⟩
      S
    □
  where
    xy⨾E⊑xy : (S ⨾ ι ˘ ⨭ S ⨾ κ ˘) ⨾ E ⊑ (S ⨾ ι ˘ ⨭ S ⨾ κ ˘)
    xy⨾E⊑xy = ⊑-begin
          (S ⨾ ι ˘ ⨭ S ⨾ κ ˘) ⨾ E
      ≈˘⟨ ⨾-cong₁ to-⨭ ⟩
          S ⨾ E
      ⊑⟨ S⨾E⊑S ⟩
```

```
          S
        ≈⟨ to-⋐ ⟩
          S ⨾ ι ˘ ⋐ S ⨾ κ ˘
        □
      open RightInd xy⨾E⊑xy using (*-rightInd⋐)

  f⨾f*⊑f* : f ⨾ f* ⊑ f*
  f⨾f*⊑f* = ⊔-upper₂ ⟨⊑⊑⟩ *-recDef₁⊑ StarA
  E*-recDef₁ : Id ⊔ E ⨾ E* ⊑ E*
  E*-recDef₁ = ⊑-begin
          Id ⊔ E ⨾ E*
        ≈˘⟨ ⊔-cong (isIdentity-≈Id Id⊞-isIdentity) (⨾-cong₁ ⋐-⋑) ⟩
          (ι ⋑ κ) ⊔ ((a ⋐ b) ⋑ (c ⋐ d)) ⨾ E*
        ≈⟨ ⊔-cong₂ ⋑-⨾ ⟩
          (ι ⋑ κ) ⊔ ((a ⋐ b) ⨾ E* ⋑ (c ⋐ d) ⨾ E*)
        ≈⟨ ⋑-⊔-⋑ ⟩
          (ι ⊔ (a ⋐ b) ⨾ E*) ⋑ (κ ⊔ (c ⋐ d) ⨾ E*)
        ⊑⟨ ⋑-monotone
          (⊑-begin
                ι ⊔ (a ⋐ b) ⨾ E*
              ≈⟨ ⊔-cong to-⋐ (⨾-⋐ ⟨≈≈⟩ ⋐-cong ⋐-⨾-⋑ ⋐-⨾-⋑) ⟩
                (ι ⨾ ι ˘ ⋐ ι ⨾ κ ˘) ⊔ ((a ⨾ f* ⊔ b ⨾ h) ⋐ (a ⨾ g ⊔ b ⨾ k))
              ≈⟨ ⋐-⊔-⋐ ⟩
                (ι ⨾ ι ˘ ⊔ a ⨾ f* ⊔ b ⨾ h) ⋐ (ι ⨾ κ ˘ ⊔ a ⨾ g ⊔ b ⨾ k)
              ⊑⟨ ⋐-monotone
                (⊑-begin
                      ι ⨾ ι ˘ ⊔ a ⨾ f* ⊔ b ⨾ d* ⨾ c ⨾ f*
                    ≈⟨ ⊔-cong (isIdentity-≈Id leftKernel) (⊔-cong₂ ⨾-assoc₃₊₁ ⟨≈˘≈˘⟩ ⨾-⊔-distribL) ⟩
                      Id ⊔ f ⨾ f*
                    ⊑⟨ *-recDef₁⊑ StarA ⟩
                      f*
                    □)
                (⊑-begin
                      ι ⨾ κ ˘ ⊔ a ⨾ g ⊔ b ⨾ (d* ⊔ d* ⨾ c ⨾ g)
                    ⊑⟨ ⊔-universal (commutes ⟨≈⊑⟩ ⊥-⊑) (⊔-universal
                      (⊑-begin
                            a ⨾ f* ⨾ b ⨾ d*
                          ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ (⨾-monotone₁ ⊔-upper₁ ⟨⊑⊑⟩ *-stepL StarA) ⟩
                            f* ⨾ b ⨾ d*
                          □)
                      (⊑-begin
                            b ⨾ (d* ⊔ d* ⨾ c ⨾ g)
                          ≈⟨ ⨾-⊔-distribR ⟩
                            b ⨾ d* ⊔ b ⨾ d* ⨾ c ⨾ g
                          ⊑⟨ ⊔-universal
                            (⊑-begin
                                b ⨾ d*
                              ⊑⟨ proj₁ (*-isSuperidentity StarA) ⟩
                                f* ⨾ b ⨾ d*
                              □)
                            (⊑-begin
                                b ⨾ d* ⨾ c ⨾ g
                              ⊑⟨ ⨾-assoc₃₊₁ ⟨≈˘⊑⟩ ⨾-monotone₁ ⊔-upper₂ ⟩
                                f ⨾ f* ⨾ b ⨾ d*
                              ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ (*-stepL StarA) ⟩
                                f* ⨾ b ⨾ d*
                              □)
                          ⟩
                            g
```

```
                          □))
                        ⟩
                          g
                     □)
                 ⟩
                   f* ⋐ g
               □)
            (⊑-begin
                   κ ⊔ (c ⋐ d) ⨾ E*
               ≈⟨ ⊔-cong to-⋐ (⨾-⋐ ⟨≈≈⟩ ⋐-cong ⋐-⨾-⋑ ⋐-⨾-⋑) ⟩
                   (κ ⨾ ι ˘ ⋐ κ ⨾ κ ˘) ⊔ ((c ⨾ f* ⊔ d ⨾ h) ⋐ (c ⨾ g ⊔ d ⨾ k))
               ≈⟨ ⋐-⊔-⋐ ⟩
                   (κ ⨾ ι ˘ ⊔ c ⨾ f* ⊔ d ⨾ h) ⋐ (κ ⨾ κ ˘ ⊔ c ⨾ g ⊔ d ⨾ k)
               ⊑⟨ ⋐-monotone
                    (⊑-begin
                          κ ⨾ ι ˘ ⊔ c ⨾ f* ⊔ d ⨾ d* ⨾ c ⨾ f*
                       ⊑⟨ ⊔-universal (κ⨾ι˘-is-⊥ _) (⊔-universal
                          (proj₁ (*-isSuperidentity StarB))
                          (⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ (*-stepL StarB)))
                        ⟩
                          d* ⨾ c ⨾ f*
                       □)
                    (⊑-begin
                          κ ⨾ κ ˘ ⊔ c ⨾ g ⊔ d ⨾ (d* ⊔ d* ⨾ c ⨾ g)
                       ⊑⟨ ⊔-assocL ⟨≈⊑⟩ ⊔-universal
                         (⊔-monotone
                                (isIdentity-≈Id rightKernel ⟨≈⊑⟩ *-isReflexive StarB)
                                (proj₁ (*-isSuperidentity StarB)))
                         (⨾-⊔-distribR ⟨≈⊑⟩ ⊔-monotone
                                (*-stepL StarB)
                                (⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ (*-stepL StarB)))
                        ⟩
                          d* ⊔ d* ⨾ c ⨾ g
                       □)
                 ⟩
                   h ⋐ k
               □)
            ⟩
              (f* ⋐ g) ⋑ (h ⋐ k)
          ≈⟨ ⋐-⋑      ⟩
              E*
          □

   E*-isStar : IsStar E E*
   E*-isStar = mkIsStar′ E E* E*-recDef₁ E*-leftInd E*-rightInd
```

```
module Whole (E₀ : Mor A⊞B A⊞B) where
   a : Mor A A
   a = ι ⨾ E₀ ⨾ ι ˘
   b : Mor A B
   b = ι ⨾ E₀ ⨾ κ ˘
   c : Mor B A
   c = κ ⨾ E₀ ⨾ ι ˘
   d : Mor B B
   d = κ ⨾ E₀ ⨾ κ ˘
   open Square a b c d public using (E; E*; E*′; E*-1; E*-isStar)
   E≈E₀ : E ≈ E₀
   E≈E₀ = ≈-begin
```

```
          (a ⊅ c) ⊄ (b ⊅ d)
        ≈˘⟨ ⊄-cong ⊅-⨾ ⊅-⨾ ⟩
          (ι ⊅ κ) ⨾ E₀ ⨾ ι ˘      ⊄    (ι ⊅ κ) ⨾ E₀ ⨾ κ ˘
        ≈⟨ ⊄-cong (proj₁ jointId) (proj₁ jointId) ⟩
          E₀ ⨾ ι ˘   ⊄       E₀ ⨾ κ ˘
        ≈˘⟨ ⨾-⊄ ⟩
          E₀ ⨾ (ι ˘ ⊄ κ ˘)
        ≈⟨ proj₂ jointId ⟩
          E₀
        □
  SumStarOp : LocalStarOp A⊞B
  SumStarOp = record
    {_* = Whole.E*
    ; isStar = λ E₀ → let open Whole E₀ in IsStar-subst₁ E≈E₀ E*-isStar
    }
  UnitSumStarOp : ({R : Mor A A} → R *A ≈ Id) → LocalStarOp A⊞B
  UnitSumStarOp A*≈Id = record
    {_* = Whole.E*′
    ; isStar = λ E₀ →       let open Whole E₀
                            in IsStar-subst₂ (E*-1 A*≈Id)
                              (IsStar-subst₁ E≈E₀ E*-isStar)
    }

open SumStar public using (SumStarOp)
UnitSumStarOp : {A B : Obj} → IsUnit A → LocalStarOp B → (Sum : DirectSum-Z zeroMor A B)
                  → LocalStarOp (DirectSum-Z.obj zeroMor Sum)
UnitSumStarOp A-isUnit StarB Sum = SumStar.UnitSumStarOp Sum (UnitStarOp A-isUnit) StarB ≈-refl
```

Chapter 16

Cotabulations

16.1 Categoric.Cotabulation

```
module Categoric.Cotabulation
      {i j k₁ k₂ : Level} {Obj : Set i} (occ : OCC j k₁ k₂ Obj)
      (joinOp              : JoinOp (OCC.orderedSemigroupoid occ))
      (joinCompDistrL  : JoinCompDistrL joinOp)
      (joinCompDistrR  : JoinCompDistrR joinOp)
   where
      open OCC occ
      open JoinOp                 joinOp
      open JoinCompDistrL      joinCompDistrL
      open JoinCompDistrR      joinCompDistrR
      open RawUSLSGC-Props osgc joinOp

      record IsPreCotabulation {B C D : Obj} (R : Mor B D) (S : Mor C D) (W : Mor B C) : Set (k₁ ⊍ k₂)
         where
         field
            commutes : R ⨾ S ˘ ≈ W
            jointId : R ˘ ⨾ R ⊔ S ˘ ⨾ S ≈ Id
            leftMappingI : isMappingI R
            rightMappingI : isMappingI S

         LeftMapping : Mapping B D
         LeftMapping = mkMappingI R leftMappingI
         RightMapping : Mapping C D
         RightMapping = mkMappingI S rightMappingI
         leftUnivalent : isUnivalent R
         leftUnivalent = mappingUnivalent LeftMapping
         rightUnivalent : isUnivalent S
         rightUnivalent = mappingUnivalent RightMapping
         leftTotal : isTotal R
         leftTotal = mappingTotal LeftMapping
         rightTotal : isTotal S
         rightTotal = mappingTotal RightMapping

         subfactorLeft : W ⨾ S ⊑ R
         subfactorLeft = ⊑-begin
                  W ⨾ S
               ≈˘⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ commutes ⟩
                  R ⨾ S ˘ ⨾ S
```

```
      ⊑⟨ proj₂ rightUnivalent ⟩
        R
      □
  subfactorRight : W ˘ ⨾ R ⊑ S
  subfactorRight = ⊑-begin
        W ˘ ⨾ R
      ≈˘⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (˘-involutionRightConv ⟨≈˘≈⟩ ˘-cong commutes) ⟩
        S ⨾ R ˘ ⨾ R
      ⊑⟨ proj₂ leftUnivalent ⟩
        S
      □

  difunctional : isDifunctional W
  difunctional = ⊑-begin
        W ⨾ W ˘ ⨾ W
      ≈˘⟨ ⨾-cong₂₂ commutes ⟩
        W ⨾ W ˘ ⨾ R ⨾ S ˘
      ⊑⟨ ⨾-monotone₂ (⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ subfactorRight) ⟩
        W ⨾ S ⨾ S ˘
      ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ subfactorLeft ⟩
        R ⨾ S ˘
      ≈⟨ commutes ⟩
        W
      □

  private
    W⨾W˘⊑R⨾R˘ : W ⨾ W ˘ ⊑ R ⨾ R ˘
    W⨾W˘⊑R⨾R˘ = ⊑-begin
        W ⨾ W ˘
      ≈⟨ ⨾-cong₂ (˘-cong commutes ⟨≈˘≈⟩ ˘-involutionRightConv) ⟩
        W ⨾ S ⨾ R ˘
      ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ subfactorLeft ⟩
        R ⨾ R ˘
      □
  leftKernel⊒ : Id ⊔ W ⨾ W ˘ ⊑ R ⨾ R ˘
  leftKernel⊒ = ⊔-universal (mappingTotalI LeftMapping) W⨾W˘⊑R⨾R˘
```

For illustration, we leave the following alternative proof of W ⨾ W ˘ ⊑ R ⨾ R ˘ here for the time being.

```
  private
    R⨾R˘≈R⨾R˘⊔W⨾W˘ : R ⨾ R ˘ ≈ R ⨾ R ˘ ⊔ W ⨾ W ˘
    R⨾R˘≈R⨾R˘⊔W⨾W˘ = ≈-begin
        R ⨾ R ˘
      ≈˘⟨ ⨾-cong₂ leftId ⟩
        R ⨾ Id ⨾ R ˘
      ≈˘⟨ ⨾-cong₂₁ jointId ⟩
        R ⨾ (R ˘ ⨾ R ⊔ S ˘ ⨾ S) ⨾ R ˘
      ≈⟨ ⨾-cong₂ (⨾-⊔-distribL ⟨≈≈⟩ ⊔-cong₂ ⨾-assoc) ⟩
        R ⨾ ((R ˘ ⨾ R) ⨾ R ˘ ⊔ S ˘ ⨾ S ⨾ R ˘)
      ≈⟨ ⨾-⊔-distribR ⟨≈≈⟩
         ⊔-cong (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (mappingBiDifunctional LeftMapping)) ⨾-assocL ⟩
        R ⨾ R ˘ ⊔ (R ⨾ S ˘) ⨾ S ⨾ R ˘
      ≈⟨ ⊔-cong₂ (⨾-cong commutes (˘-involutionRightConv ⟨≈˘≈⟩ ˘-cong commutes)) ⟩
        R ⨾ R ˘ ⊔ W ⨾ W ˘
      □

  factor : {D′ : Obj} (R′ : Mor B D′) (S′ : Mor C D′) → Mor D D′
  factor R′ S′ = R ˘ ⨾ R′ ⊔ S ˘ ⨾ S′
```

```agda
record IsCotabulation {B C D : Obj} (R : Mor B D) (S : Mor C D) (W : Mor B C) : Set k₁ where
  field
    commutes : R ⨾ S ˘ ≈ W
    jointId : R ˘ ⨾ R ⊔ S ˘ ⨾ S ≈ Id
    leftKernel : R ⨾ R ˘ ≈ Id ⊔ W ⨾ W ˘
    rightKernel : S ⨾ S ˘ ≈ Id ⊔ W ˘ ⨾ W


  leftMappingI : isMappingI R
  leftMappingI = (⊔-upper₁ ⟨⊑≈⟩ jointId), (⊔-upper₁ ⟨⊑≈˘⟩ leftKernel)
  rightMappingI : isMappingI S
  rightMappingI = (⊔-upper₂ ⟨⊑≈⟩ jointId), (⊔-upper₁ ⟨⊑≈˘⟩ rightKernel)
  isPreCotabulation : IsPreCotabulation R S W
  isPreCotabulation = record
    {commutes       = commutes
    ; jointId       = jointId
    ; leftMappingI  = leftMappingI
    ; rightMappingI = rightMappingI
    }
  open IsPreCotabulation isPreCotabulation public
    hiding (commutes; jointId; leftMappingI; rightMappingI)


  factorLeft :   {D′ : Obj} {R′ : Mor B D′} {S′ : Mor C D′}
             →   W ⨾ S′ ⊑ R′ → W ˘ ⨾ R′ ⊑ S′ → R ⨾ factor R′ S′ ≈ R′
  factorLeft {_} {R′} {S′} W⨾S′⊑R′ W˘⨾R′⊑S′ = ≈-begin
        R ⨾ factor R′ S′
      ≈⟨ ⨾-⊔-distribR ⟩
        R ⨾ R ˘ ⨾ R′ ⊔ R ⨾ S ˘ ⨾ S′
      ≈⟨ ⊔-cong (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ leftKernel) (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ commutes) ⟩
        (Id ⊔ W ⨾ W ˘) ⨾ R′ ⊔ W ⨾ S′
      ≈⟨ ⊔-cong₁ (⨾-⊔-distribL ⟨≈≈⟩ ⊔-cong leftId ⨾-assoc) ⟨≈≈⟩ ⊔-assoc ⟩
        R′ ⊔ (W ⨾ (W ˘ ⨾ R′) ⊔ W ⨾ S′)
      ≈⟨ ⊑-to-⊔₁ (⊑-begin
          W ⨾ (W ˘ ⨾ R′) ⊔ W ⨾ S′
        ⊑⟨ ⊔-monotone₁ (⨾-monotone₂ W˘⨾R′⊑S′) ⟩
          W ⨾ S′ ⊔ W ⨾ S′
        ≈⟨ ⊔-idempotent ⟩
          W ⨾ S′
        ⊑⟨ W⨾S′⊑R′ ⟩
          R′
        □) ⟩
        R′
      □
  factorRight : {D′ : Obj} {R′ : Mor B D′} {S′ : Mor C D′}
             →   W ⨾ S′ ⊑ R′ → W ˘ ⨾ R′ ⊑ S′ → S ⨾ factor R′ S′ ≈ S′
  factorRight {_} {R′} {S′} W⨾S′⊑R′ W˘⨾R′⊑S′ = ≈-begin
        S ⨾ factor R′ S′
      ≈⟨ ⨾-⊔-distribR ⟩
        S ⨾ R ˘ ⨾ R′ ⊔ S ⨾ S ˘ ⨾ S′
      ≈⟨ ⊔-cong (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (˘-involutionRightConv ⟨≈˘≈⟩ ˘-cong commutes))
                (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ rightKernel) ⟩
        W ˘ ⨾ R′ ⊔ (Id ⊔ W ˘ ⨾ W) ⨾ S′
      ≈⟨ ⊔-cong₂ (⨾-⊔-distribL ⟨≈≈⟩ ⊔-cong leftId ⨾-assoc ⟨≈≈⟩ ⊔-commutative) ⟨≈≈⟩ ⊔-assocL ⟩
        (W ˘ ⨾ R′ ⊔ W ˘ ⨾ W ⨾ S′) ⊔ S′
      ≈⟨ ⊑-to-⊔₂ (⊑-begin
          W ˘ ⨾ R′ ⊔ W ˘ ⨾ W ⨾ S′
        ⊑⟨ ⊔-monotone₂ (⨾-monotone₂ W⨾S′⊑R′) ⟩
          W ˘ ⨾ R′ ⊔ W ˘ ⨾ R′
        ≈⟨ ⊔-idempotent ⟩
```

```
          W ˘ ⨾ R′
        ⊑⟨ W˘⨾R′⊑S′ ⟩
          S′
        □) ⟩
        S′
      □


factorUnivalentI   :  {D′ : Obj} {R′ : Mor B D′} {S′ : Mor C D′}
                   → W ⨾ S′ ⊑ R′ → W ˘ ⨾ R′ ⊑ S′
                   → isUnivalentI R′ → isUnivalentI S′ → isUnivalentI (factor R′ S′)
factorUnivalentI   {_} {R′} {S′} W⨾S′⊑R′ W˘⨾R′⊑S′ univalR′ univalS′  =  ⊑-begin
        (R ˘ ⨾ R′ ⊔ S ˘ ⨾ S′) ˘ ⨾ factor R′ S′
      ≈⟨ ⨾-cong₁ (˘-⊔-distrib ⟨≈≈⟩ ⊔-cong ˘-involutionLeftConv ˘-involutionLeftConv) ⟩
        (R′ ˘ ⨾ R ⊔ S′ ˘ ⨾ S) ⨾ factor R′ S′
      ≈⟨ ⨾-⊔-distribL ⟨≈≈⟩ ⊔-cong ⨾-assoc ⨾-assoc ⟩
        R′ ˘ ⨾ R ⨾ factor R′ S′ ⊔ S′ ˘ ⨾ S ⨾ factor R′ S′
      ≈⟨ ⊔-cong (⨾-cong₂ (factorLeft W⨾S′⊑R′ W˘⨾R′⊑S′)) (⨾-cong₂ (factorRight W⨾S′⊑R′ W˘⨾R′⊑S′)) ⟩
        R′ ˘ ⨾ R′ ⊔         S′ ˘ ⨾ S′
      ⊑⟨ ⊔-monotone univalR′ univalS′ ⟨⊑≈⟩ ⊔-idempotent ⟩
        Id
      □

factorUnivalent  :        {D′ : Obj} {R′ : Mor B D′} {S′ : Mor C D′}
                 →        W ⨾ S′ ⊑ R′ → W ˘ ⨾ R′ ⊑ S′
                 → isUnivalent R′ → isUnivalent S′ → isUnivalent (factor R′ S′)
factorUnivalent W⨾S′⊑R′ W˘⨾R′⊑S′ univalR′ univalS′  =  isUnivalent-from-I
  (factorUnivalentI W⨾S′⊑R′ W˘⨾R′⊑S′ (isUnivalent-to-I univalR′) (isUnivalent-to-I univalS′))


factorTotalI₀   :  {D′ : Obj} {R′ : Mor B D′} {S′ : Mor C D′}
                → W ⨾ S′ ⊑ R′ → W ˘ ⨾ R′ ⊑ S′
                → isTotal R′ → isTotal S′ → isTotalI (factor R′ S′)
factorTotalI₀   {_} {R′} {S′} W⨾S′⊑R′ W˘⨾R′⊑S′ totalR′ totalS′  =  ⊑-begin
        Id
      ≈˘⟨ jointId ⟩
        R ˘ ⨾ R ⊔ S ˘ ⨾ S
      ⊑⟨ ⊔-monotone (⨾-monotone₂ (proj₁ totalR′ ⟨⊑≈⟩ ⨾-assoc))
                    (⨾-monotone₂ (proj₁ totalS′ ⟨⊑≈⟩ ⨾-assoc)) ⟩
        R ˘ ⨾ R′ ⨾ R′ ˘ ⨾ R ⊔ S ˘ ⨾ S′ ⨾ S′ ˘ ⨾ S
      ⊑⟨ ⊔-cong ⨾-assocL ⨾-assocL ⟨≈⊑⟩ ⊔-⨾-⊔-par ⟩
        factor R′ S′ ⨾ ((R′ ˘ ⨾ R) ⊔ (S′ ˘ ⨾ S))
      ≈˘⟨ ⨾-cong₂ (˘-⊔-distrib ⟨≈≈⟩ ⊔-cong ˘-involutionLeftConv ˘-involutionLeftConv) ⟩
        factor R′ S′ ⨾ (factor R′ S′) ˘
      □

factorTotalI :   {D′ : Obj} {R′ : Mor B D′} {S′ : Mor C D′}
             →   W ⨾ S′ ⊑ R′ → W ˘ ⨾ R′ ⊑ S′
             → isTotalI R′ → isTotalI S′ → isTotalI (factor R′ S′)
factorTotalI W⨾S′⊑R′ W˘⨾R′⊑S′ totalR′ totalS′
  =  factorTotalI₀ W⨾S′⊑R′ W˘⨾R′⊑S′ (isTotal-from-I totalR′) (isTotal-from-I totalS′)

factorTotal  :  {D′ : Obj} {R′ : Mor B D′} {S′ : Mor C D′}
             → W ⨾ S′ ⊑ R′ → W ˘ ⨾ R′ ⊑ S′
             → isTotal R′ → isTotal S′ → isTotal (factor R′ S′)
factorTotal W⨾S′⊑R′ W˘⨾R′⊑S′ totalR′ totalS′
  =  isTotal-from-I (factorTotalI₀ W⨾S′⊑R′ W˘⨾R′⊑S′ totalR′ totalS′)
```

Chapter 17

Relations between PER Quotients

We now define derived semigroupoids and categories with partial equivalence relations (PERs) as objects, and with morphisms compatible with the PERs as morphisms. Moving from the objects of the base semigroupoid or category to PERs can be seen as moving to simultaneous quotients and subobjects determined by these PERs.

Since each PER makes up its own identity morphism, the base semigorupoid of the PER quotient construction does not need identity morphisms, wich is particularly important for applying these constructions to the relation semigroupoids of Chapter 18 to obtain semigroupoids and categories of relations between setoids, see the discussion about the definition of identity morphisms in Sect. 18.1.

Since symmetry is one of the key propertie of PERs, we need at least converse in the base semigroupoids. In Sect. 11.4, we showed that PERs defined as symmetric, transitive, and codifunctional morphisms in an OSGC are exactly the symmetric idempotents that can be defined already in the underlying semigroupoid with converse. Therefore, we base the development here on the symmetric idempotents from Sect. 3.15.

17.1 Categoric.PERQ

This module only re-exports its imports:

```
open import Categoric.PERQ.ConvSemigroupoid public   -- Sect. 17.2
open import Categoric.PERQ.ConvCategory     public   -- Sect. 17.3
open import Categoric.PERQ.OSGC             public   -- Sect. 17.4
open import Categoric.PERQ.OCC              public   -- Sect. 17.5
open import Categoric.PERQ.USLCC            public   -- Sect. 17.6
open import Categoric.PERQ.KSGC             public   -- Sect. 17.7
open import Categoric.PERQ.KCC              public   -- Sect. 17.8
open import Categoric.PERQ.DistrLatCC       public   -- Sect. 17.9
open import Categoric.PERQ.DistrAllegory    public   -- Sect. 17.10
open import Categoric.PERQ.DivAllegory      public   -- Sect. 17.11
open import Categoric.PERQ.DistrActAllegory public   -- Sect. 17.12
```

17.2 Categoric.PERQ.ConvSemigroupoid

Given a base ConvSemigroupoid, we can construct a derived ConvSemigroupoid that has as objects the PERs ("partial equivalence repations") of the base, and as morphisms those base morphisms that are "closed", i.e., invariant under composition with the adjacent PERs. Ignoring the "partial" aspects of PERs, this construction can be understood by interpreting a PER object E as the quotient of the domain of E by E; for this reason we call the resulting semigroupoid a "PERQ semigroupoid".

The construction here is a minor variant of the construction by Freyd and Scedrov (1990, 1.28) of the category $\mathit{Split}(\mathcal{E})$ for some class $\mathcal{E}$ of idempotents (no symmetry assumed there). Freyd and Scedrov restrict the morphisms via Lclosed and Rclosed.

For easy access to basic entities of PERQ semigroupoids from within the context of the base semigroupoid, we define the following module:

```
module PERQCSG {i j k : Level} {Obj : Set i}
               (Base : ConvSemigroupoid {i} j k Obj) where
  open ConvSemigroupoid Base
  open SymIdempot
  record PERQMor (A B : SymIdempot) : Set (j ⊍ k) where
    field mor : Mor (obj A) (obj B)
          closed : ⟪ A ⟫ ⨾ mor ⨾ ⟪ B ⟫ ≈ mor
    Lclosed : ⟪ A ⟫ ⨾ mor ≈ mor
    Lclosed = ≈-begin
          ⟪ A ⟫ ⨾ mor
      ≈˘⟨ ⨾-cong₂ closed ⟩
          ⟪ A ⟫ ⨾ ⟪ A ⟫ ⨾ mor ⨾ ⟪ B ⟫
      ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (idempotent A) ⟩
          ⟪ A ⟫ ⨾ mor ⨾ ⟪ B ⟫
      ≈⟨ closed ⟩
        mor   □
    ˘Lclosed : ⟪ A ⟫ ˘ ⨾ mor ≈ mor
    ˘Lclosed = ≈-begin
          ⟪ A ⟫ ˘ ⨾ mor
      ≈⟨ ⨾-cong₁ (symmetric A) ⟩
          ⟪ A ⟫ ⨾ mor
      ≈⟨ Lclosed ⟩
          mor □
    Rclosed : mor ⨾ ⟪ B ⟫ ≈ mor
    Rclosed = ≈-begin
        mor ⨾ ⟪ B ⟫
      ≈⟨ ⨾-cong₁ closed ⟨≈˘≈⟩ ⨾-assoc₃₊₁ ⟩
          ⟪ A ⟫ ⨾ mor ⨾ ⟪ B ⟫ ⨾ ⟪ B ⟫
      ≈⟨ ⨾-cong₂₂ (idempotent B) ⟩
          ⟪ A ⟫ ⨾ mor ⨾ ⟪ B ⟫
      ≈⟨ closed ⟩
        mor   □
    ˘Rclosed : mor ⨾ ⟪ B ⟫ ˘ ≈ mor
    ˘Rclosed = ≈-begin
        mor ⨾ ⟪ B ⟫ ˘
      ≈⟨ ⨾-cong₂ (symmetric B) ⟩
        mor ⨾ ⟪ B ⟫
      ≈⟨ Rclosed ⟩
          mor □
  open PERQMor
  infixr    9 _⨾PERQ_
  infix     4 _≈PERQ_
  _≈PERQ_ = λ {A B : SymIdempot} (F G : PERQMor A B) → mor F ≈ mor G
  ≈PERQMor-isEquivalence : {A B : SymIdempot} → IsEquivalence (_≈PERQ_ {A} {B})
  ≈PERQMor-isEquivalence = record
          {refl  = ≈-refl
          ;sym   = ≈-sym
          ;trans = ≈-trans
          }
  _⨾PERQ_ : {A B C : SymIdempot} → PERQMor A B → PERQMor B C → PERQMor A C
  _⨾PERQ_ {A} {B} {C} F G = record
    {mor = mor F ⨾ mor G
    ;closed = ≈-begin
```

```
          ⟪ A ⟫ ⨾ (mor F ⨾ mor G) ⨾ ⟪ C ⟫
        ≈⟨ ⨾-cong₂ (⨾-assoc ⟨≈≈⟩ ⨾-cong₂ (Rclosed G)) ⟩
          ⟪ A ⟫ ⨾ mor F ⨾ mor G
        ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (Lclosed F) ⟩
          mor F ⨾ mor G
        □
    }
  ⨾PERQ-cong  :  {A B C : SymIdempot} {F₁ F₂ : PERQMor A B} {G₁ G₂ : PERQMor B C}
              → F₁ ≈PERQ F₂ → G₁ ≈PERQ G₂ → F₁ ⨾PERQ G₁ ≈PERQ F₂ ⨾PERQ G₂
  ⨾PERQ-cong  = ⨾-cong
  ⨾PERQ-assoc :  {A B C D : SymIdempot}
              → {F : PERQMor A B} {G : PERQMor B C} {H : PERQMor C D}
              → (F ⨾PERQ G) ⨾PERQ H ≈PERQ F ⨾PERQ (G ⨾PERQ H)
  ⨾PERQ-assoc = ⨾-assoc

  PERQHom : LocalSetoid SymIdempot (j ⊍ k) k
  PERQHom = λ A B → record
      {Carrier        = PERQMor A B
      ; _≈_           = _≈PERQ_
      ; isEquivalence = ≈PERQMor-isEquivalence
      }

  PERQCompOp : CompOp PERQHom
  PERQCompOp = record
      {    _⨾_  = _⨾PERQ_
      ; ⨾-cong  = λ {A B C P Q R S} P≈Q R≈S → ⨾PERQ-cong {F₁ = P} {Q} {R} {S} P≈Q R≈S
      ; ⨾-assoc = λ {A B C D Q R S} → ⨾PERQ-assoc {F = Q} {R} {S}
      }

  _˘PERQ : {A B : SymIdempot} → PERQMor A B → PERQMor B A
  _˘PERQ {A} {B} F = record
    {mor = (mor F) ˘
    ; closed = ≈-begin
          ⟪ B ⟫ ⨾ (mor F) ˘ ⨾ ⟪ A ⟫
        ≈˘⟨ ⨾-cong (symmetric B) (⨾-cong₂ (symmetric A)) ⟩
          ⟪ B ⟫ ˘ ⨾ (mor F) ˘ ⨾ ⟪ A ⟫ ˘
      ≈˘⟨ ⨾-cong₂ ˘-involution ⟩
          ⟪ B ⟫ ˘ ⨾ (⟪ A ⟫ ⨾ mor F) ˘
      ≈⟨ ⨾-cong₂ (˘-cong (Lclosed F)) ⟩
          ⟪ B ⟫ ˘ ⨾ (mor F) ˘
      ≈˘⟨ ˘-involution ⟩
          (mor F ⨾ ⟪ B ⟫) ˘
        ≈⟨ ˘-cong (Rclosed F) ⟩
          mor F ˘
        □
    }
```

Defining the complete PERQ Semigroupoid then just collects these entities together into the corresponding records:

```
PERQSemigroupoid :  {i j k : Level} {Obj : Set i}
                  → (Base : ConvSemigroupoid {i} j k Obj)
                  → Semigroupoid {i ⊍ j ⊍ k} (j ⊍ k) k (ConvSemigroupoid.SymIdempot Base)
PERQSemigroupoid {i} {j} {k} {Obj} Base = let
  open ConvSemigroupoid Base using (module SymIdempot)
  open SymIdempot
  open PERQCSG Base
  in record
  {Hom = PERQHom
  ; compOp = PERQCompOp
  }

PERQConvSemigroupoid :  {i j k : Level} {Obj : Set i}
```

```
                              → (Base : ConvSemigroupoid {i} j k Obj)
                              → ConvSemigroupoid {i ⊍ j ⊍ k} (j ⊍ k) k (ConvSemigroupoid.SymIdempot Base)
PERQConvSemigroupoid Base = let
  open ConvSemigroupoid Base
  open PERQCSG Base
  in record
  {semigroupoid = PERQSemigroupoid Base
  ; convOp = record
    {_˘              = _˘PERQ
    ; ˘-cong         = ˘-cong
    ; ˘˘             = ˘˘
    ; ˘-involution   = ˘-involution
    }
  }
```

Symmetric idempotents in a PERQConvSemigroupoid have splittings:

```
module PERQ-Quotient {i j k : Level} {Obj : Set i}
                     (Base : ConvSemigroupoid {i} j k Obj) where
  open PERQCSG Base
  open ConvSemigroupoid Base
  open ConvSemigroupoid (PERQConvSemigroupoid Base) using ()
    renaming
      (Mor to Mor′
      ; module SymIdempot to SymIdempot′
      ; SymIdempot to SymIdempot′
      ; SymIdempotSplitting to SymIdempotSplitting′
      )
  quotientSplitting : (SI : SymIdempot′) → SymIdempotSplitting′ SI
  quotientSplitting SI = let
      open SymIdempot′ SI using () renaming
        (obj to A; ⟪_⟫ to Ξ; symmetric to Ξ-sym; idempotent to Ξ-idempot)
      open SymIdempot A using () renaming (obj to A₀; ⟪_⟫ to Ξ₀)
      Q = record
        {obj = A₀
        ; ⟪_⟫ = PERQMor.mor Ξ
        ; prop = record {symmetric = Ξ-sym; idempotent = Ξ-idempot}
        }
      q : Mor′ A Q
      q = record
        {mor = PERQMor.mor Ξ
        ; closed = ≈-begin
              Ξ₀ ⨾ PERQMor.mor Ξ ⨾ PERQMor.mor Ξ
            ≈⟨ ⨾-cong₂ Ξ-idempot ⟩
              Ξ₀ ⨾ PERQMor.mor Ξ
            ≈⟨ PERQMor.Lclosed Ξ ⟩
              PERQMor.mor Ξ
            □
        }
    in record
    {obj = Q
    ; mor = q
    ; factors = ≈-begin
              PERQMor.mor q ⨾ PERQMor.mor q ˘
            ≈⟨ ⨾-cong₂ Ξ-sym ⟩
              PERQMor.mor Ξ ⨾ PERQMor.mor Ξ
            ≈⟨ Ξ-idempot ⟩
              PERQMor.mor Ξ
            □
```

```
  ; splitId = let
          QQ : PERQMor.mor q ˘ ⨾ PERQMor.mor q ≈ PERQMor.mor Ξ
          QQ = ≈-begin
                 PERQMor.mor q ˘ ⨾ PERQMor.mor q
             ≈⟨ ⨾-cong₁ Ξ-sym ⟩
                 PERQMor.mor Ξ ⨾ PERQMor.mor Ξ
             ≈⟨ Ξ-idempot ⟩
                 PERQMor.mor Ξ
             □
    in ((λ {_} {R} →      ≈-begin
             (PERQMor.mor q ˘ ⨾ PERQMor.mor q) ⨾ PERQMor.mor R
          ≈⟨ ⨾-cong₁ QQ ⟩
             PERQMor.mor Ξ ⨾ PERQMor.mor R
          ≈⟨ PERQMor.Lclosed R ⟩
             PERQMor.mor R
          □)), ((λ {_} {R} →    ≈-begin
             PERQMor.mor R ⨾ (PERQMor.mor q ˘ ⨾ PERQMor.mor q)
          ≈⟨ ⨾-cong₂ QQ ⟩
             PERQMor.mor R ⨾ PERQMor.mor Ξ
          ≈⟨ PERQMor.Rclosed R ⟩
             PERQMor.mor R
          □))
  }
```

17.3 Categoric.PERQ.ConvCategory

```
PERQCategory : {i j k : Level} {Obj : Set i}
             → (Base : ConvSemigroupoid {i} j k Obj)
             → Category {i ⊍ j ⊍ k} (j ⊍ k) k (ConvSemigroupoid.SymIdempot Base)
PERQCategory Base = let
  open ConvSemigroupoid Base
  open PERQCSG Base
  open SymIdempot
  open PERQMor
  in record
  {semigroupoid = PERQSemigroupoid Base
  ; idOp = record
    {Id         = λ {A} → record {mor = ⟪ A ⟫
                                 ; closed = ≈-begin
                                      ⟪ A ⟫ ⨾ ⟪ A ⟫ ⨾ ⟪ A ⟫
                                    ≈⟨ ⨾-cong₂ (idempotent A) ⟩
                                      ⟪ A ⟫ ⨾ ⟪ A ⟫
                                    ≈⟨ idempotent A ⟩
                                      ⟪ A ⟫ □}
    ; leftId    = λ {A} {B} {F} → Lclosed F
    ; rightId   = λ {A} {B} {F} → Rclosed F
    }
  }

PERQConvCategory : {i j k : Level} {Obj : Set i}
                 → (Base : ConvSemigroupoid {i} j k Obj)
                 → ConvCategory {i ⊍ j ⊍ k} (j ⊍ k) k (ConvSemigroupoid.SymIdempot Base)
PERQConvCategory Base = record
  {convSemigroupoid = PERQConvSemigroupoid Base
  ; idOp = Category.idOp (PERQCategory Base)
  }
```

17.4 Categoric.PERQ.OSGC

```
PERQOrderedSemigroupoid : {i j k₁ k₂ : Level} {Obj : Set i}
  → (Base : OSGC {i} j k₁ k₂ Obj)
  → OrderedSemigroupoid {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (OSGC.SymIdempot Base)
PERQOrderedSemigroupoid Base = let
  open OSGC Base
  open SymIdempot
  open PERQCSG convSemigroupoid
  open PERQMor
  in record
  {Hom     = λ A B → record
    {Carrier = PERQMor A B
    ; _≈_ = _≈PERQ_
    ; _≤_ = λ F G → mor F ⊑ mor G
    ; isPartialOrder = record
      {isPreorder = record
        {isEquivalence = ≈PERQMor-isEquivalence
        ; reflexive = ⊑-reflexive
        ; trans = ⊑-trans
        }
      ; antisym = ⊑-antisym
      }
    }
  ; compOp = PERQCompOp
  ; locOrd = record
    {⨾-monotone = λ {A} {B} {C} {F} {F'} {G} {G'} leqF leqG → ⨾-monotone leqF leqG
    }
  }
```

```
PERQOSGC : {i j k₁ k₂ : Level} {Obj : Set i}
  → (Base : OSGC {i} j k₁ k₂ Obj)
  → OSGC {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (OSGC.SymIdempot Base)
PERQOSGC Base = record
  {OSGC_Base = let open OSGC Base in record
    {orderedSemigroupoid = PERQOrderedSemigroupoid Base
    ; convOp = ConvSemigroupoid.convOp (PERQConvSemigroupoid convSemigroupoid)
    ; ˘-monotone = ˘-monotone
    }
  }
```

17.5 Categoric.PERQ.OCC

```
PERQOrderedCategory : {i j k₁ k₂ : Level} {Obj : Set i}
                    → (Base : OSGC {i} j k₁ k₂ Obj)
                    → OrderedCategory {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (OSGC.SymIdempot Base)
PERQOrderedCategory Base = let open OSGC Base in record
  {orderedSemigroupoid = PERQOrderedSemigroupoid Base
  ; idOp = Category.idOp (PERQCategory convSemigroupoid)
  }
```

```
PERQOCC-Base : {i j k₁ k₂ : Level} {Obj : Set i}
             → (Base : OSGC {i} j k₁ k₂ Obj)
             → OCC-Base {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (OSGC.SymIdempot Base)
```

```
PERQOCC-Base Base = let open OSGC Base in record
  {osgc = PERQOSGC Base
  ; idOp = Category.idOp (PERQCategory convSemigroupoid)
  }


PERQOCC : {i j k₁ k₂ : Level} {Obj : Set i}
        → (Base : OSGC {i} j k₁ k₂ Obj)
        → OCC {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (OSGC.SymIdempot Base)
PERQOCC Base = record {OCC_Base = PERQOCC-Base Base}
```

17.6 Categoric.PERQ.USLCC

For join preservation, we do not need any additional material beyond adding joins to the base OCC (whereas meet preservation requires modal rules, i.e., (semi-)allegories).

```
PERQJoinOp : {i j k₁ k₂ : Level} {Obj : Set i}
           → (Base : USLSGC {i} j k₁ k₂ Obj)
           → JoinOp (PERQOrderedSemigroupoid (USLSGC.osgc Base))
PERQJoinOp Base = let open USLSGC Base in record
  {join = λ {A} {B} R S → let
    open SymIdempot
    open PERQCSG convSemigroupoid
    open PERQMor
    in record
      {value = record
            {mor = mor R ⊔ mor S
            ; closed = ≈-begin
                ⟪ A ⟫ ⨾ (mor R ⊔ mor S) ⨾ ⟪ B ⟫
              ≈⟨ ⨾-cong₂ ⨾-⊔-distribL ⟩
                ⟪ A ⟫ ⨾ (mor R ⨾ ⟪ B ⟫ ⊔ mor S ⨾ ⟪ B ⟫)
              ≈⟨ ⨾-⊔-distribR ⟩
                ⟪ A ⟫ ⨾ mor R ⨾ ⟪ B ⟫ ⊔ ⟪ A ⟫ ⨾ mor S ⨾ ⟪ B ⟫
              ≈⟨ ⊔-cong (closed R) (closed S) ⟩
                mor R ⊔ mor S
              □}
      ; proof = record
            {bound₁ = ⊔-upper₁
            ; bound₂ = ⊔-upper₂
            ; universal = ⊔-universal
            }
      }
  }


PERQJoinCompDistrL : {i j k₁ k₂ : Level} {Obj : Set i}
                   → (Base : USLSGC {i} j k₁ k₂ Obj)
                   → JoinCompDistrL (PERQJoinOp Base)
PERQJoinCompDistrL Base = let
  open JoinCompDistrL (USLSGC.joinCompDistrL Base)
  open PERQCSG (USLSGC.convSemigroupoid Base)
  open PERQMor
  in record
    {⨾-⊔-subdistribL = λ {A B C R₁ R₂ S} → ⨾-⊔-subdistribL {R₁ = mor R₁} {mor R₂} {mor S}}


PERQJoinCompDistrR : {i j k₁ k₂ : Level} {Obj : Set i}
                   → (Base : USLSGC {i} j k₁ k₂ Obj)
```

```
                          → JoinCompDistrR (PERQJoinOp Base)
PERQJoinCompDistrR Base = let
  open JoinCompDistrR (USLSGC.joinCompDistrR Base)
  open PERQCSG (USLSGC.convSemigroupoid Base)
  open PERQMor
  in record
    {⨾-⊔-subdistribR = λ {A B C R S₁ S₂} → ⨾-⊔-subdistribR {R = mor R} {mor S₁} {mor S₂}}


PERQUSLSemigroupoid : {i j k₁ k₂ : Level} {Obj : Set i}
                    → (Base : USLSGC {i} j k₁ k₂ Obj)
                    → USLSemigroupoid {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (USLSGC.SymIdempot Base)
PERQUSLSemigroupoid Base = let open USLSGC Base in record
  {orderedSemigroupoid = PERQOrderedSemigroupoid osgc
  ; joinOp              = PERQJoinOp                Base
  ; joinCompDistrL      = PERQJoinCompDistrL        Base
  ; joinCompDistrR      = PERQJoinCompDistrR        Base
  }


PERQUSLSGC : {i j k₁ k₂ : Level} {Obj : Set i}
           → (Base : USLSGC {i} j k₁ k₂ Obj)
           → USLSGC {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (USLSGC.SymIdempot Base)
PERQUSLSGC Base = let open USLSGC Base in record
  {osgc           = PERQOSGC           osgc
  ; joinOp         = PERQJoinOp         Base
  ; joinCompDistrL = PERQJoinCompDistrL Base
  ; joinCompDistrR = PERQJoinCompDistrR Base
  }


PERQUSLCategory : {i j k₁ k₂ : Level} {Obj : Set i}
                → (Base : USLSGC {i} j k₁ k₂ Obj)
                → USLCategory {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (USLSGC.SymIdempot Base)
PERQUSLCategory Base = let open USLSGC Base in record
  {orderedCategory = PERQOrderedCategory osgc
  ; joinOp          = PERQJoinOp          Base
  ; joinCompDistrL  = PERQJoinCompDistrL  Base
  ; joinCompDistrR  = PERQJoinCompDistrR  Base
  }


PERQUSLCC : {i j k₁ k₂ : Level} {Obj : Set i}
          → (Base : USLSGC {i} j k₁ k₂ Obj)
          → USLCC {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (USLSGC.SymIdempot Base)
PERQUSLCC Base = let open USLSGC Base in record
  {occ            = PERQOCC            osgc
  ; joinOp         = PERQJoinOp         Base
  ; joinCompDistrL = PERQJoinCompDistrL Base
  ; joinCompDistrR = PERQJoinCompDistrR Base
  }


isLeastMorPERQ : {i j k₁ k₂ : Level} {Obj : Set i}
               → (Base : OSGC j k₁ k₂ Obj)
               → {A B : OSGC.SymIdempot Base}
               → let OSG = PERQOrderedSemigroupoid Base; open OSGC Base in
                 {R : OrderedSemigroupoid.Mor OSG A B}
               → LeastMor.isLeastMor orderedSemigroupoid (PERQCSG.PERQMor.mor R)
               → LeastMor.isLeastMor OSG R
isLeastMorPERQ Base {A} {B} {R} isLeast-mor-R S = isLeast-mor-R (PERQCSG.PERQMor.mor S)
```

The converse implication is not to be expected.

Bottom morphisms are closed automatically only if zero laws are assumed:

```
PERQBotMor₀ : {i j k₁ k₂ : Level} {Obj : Set i}
  →         (Base : OSGC {i} j k₁ k₂ Obj)
  →         (botMor : BotMor (OSGC.orderedSemigroupoid Base))
  →         (leftZeroLaw : LeftZeroLaw botMor)
  →         (rightZeroLaw : RightZeroLaw botMor)
  →         BotMor (PERQOrderedSemigroupoid Base)
PERQBotMor₀ Base botMor leftZeroLaw rightZeroLaw = let
                open OSGC Base
                open BotMor botMor
                open LeftZeroLaw leftZeroLaw
                open RightZeroLaw rightZeroLaw
                open SymIdempot
                open PERQCSG convSemigroupoid
                open PERQMor
  in record
  {leastMor = λ {A} {B} → record
    {mor = record
      {mor = ⊥
      ;closed = ≈-begin
          ⟪ A ⟫ ⨾ ⊥ ⨾ ⟪ B ⟫
        ≈⟨ ⨾-cong₂ leftZero ⟩
          ⟪ A ⟫ ⨾ ⊥
        ≈⟨ rightZero ⟩
              ⊥ □}
    ;proof = λ R → ⊥-⊑
    }
  }
```

In general, we still obtain least PERQ morphisms by closing the underlying least morphism:

```
PERQBotMor : {i j k₁ k₂ : Level} {Obj : Set i}
           → (Base : OSGC {i} j k₁ k₂ Obj)
           → (botMor : BotMor (OSGC.orderedSemigroupoid Base))
           → BotMor (PERQOrderedSemigroupoid Base)
PERQBotMor Base botMor = let open BotMor botMor in record
  {leastMor = λ {A} {B} → let
    open OSGC Base
    open SymIdempot
    open PERQCSG convSemigroupoid
    open PERQMor
    in record
      {mor = record
        {mor = ⟪ A ⟫ ⨾ ⊥ ⨾ ⟪ B ⟫
        ;closed = ≈-begin
            ⟪ A ⟫ ⨾ (⟪ A ⟫ ⨾ ⊥ ⨾ ⟪ B ⟫) ⨾ ⟪ B ⟫
          ≈⟨ ⨾-cong₂ (⨾-assoc₃₊₁ ⟨≈≈⟩ ⨾-cong₂₂ (idempotent B)) ⟩
            ⟪ A ⟫ ⨾ ⟪ A ⟫ ⨾ ⊥ ⨾ ⟪ B ⟫
          ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (idempotent A) ⟩
            ⟪ A ⟫ ⨾ ⊥ ⨾ ⟪ B ⟫ □}
      ;proof = λ R → ⊑-begin
            ⟪ A ⟫ ⨾ ⊥ ⨾ ⟪ B ⟫
          ⊑⟨ ⨾-monotone₂₁ ⊥-⊑ ⟩
            ⟪ A ⟫ ⨾ mor R ⨾ ⟪ B ⟫
          ≈⟨ closed R ⟩
            mor R □
      }
  }
```

Intermediate constructions might use only one zero law.

As long as we do not need to consider the zero laws separately, we derive them here only for $\mathsf{PERQBotMor}_0$

```
PERQLeftZeroLaw₀ : {i j k₁ k₂ : Level} {Obj : Set i}
  → (Base : OSGC {i} j k₁ k₂ Obj)
  → {botMor : BotMor (OSGC.orderedSemigroupoid Base)}
  → (leftZeroLaw : LeftZeroLaw botMor)
  → (rightZeroLaw : RightZeroLaw botMor)
  → LeftZeroLaw (PERQBotMor₀ Base botMor leftZeroLaw rightZeroLaw)
PERQLeftZeroLaw₀ Base leftZeroLaw rightZeroLaw = let open LeftZeroLaw leftZeroLaw in
  record {leftZero⊑ = leftZero⊑}


PERQRightZeroLaw₀ : {i j k₁ k₂ : Level} {Obj : Set i}
  → (Base : OSGC {i} j k₁ k₂ Obj)
  → {botMor : BotMor (OSGC.orderedSemigroupoid Base)}
  → (leftZeroLaw : LeftZeroLaw botMor)
  → (rightZeroLaw : RightZeroLaw botMor)
  → RightZeroLaw (PERQBotMor₀ Base botMor leftZeroLaw rightZeroLaw)
PERQRightZeroLaw₀ Base leftZeroLaw rightZeroLaw = let open RightZeroLaw rightZeroLaw in
  record {rightZero⊑ = rightZero⊑}
```

These functions are useful for the common case of a ZeroMor:

```
PERQZeroMor : {i j k₁ k₂ : Level} {Obj : Set i}
            → (Base : OSGC {i} j k₁ k₂ Obj)
            → (zroMor : ZeroMor (OSGC.orderedSemigroupoid Base))
            → ZeroMor (PERQOrderedSemigroupoid Base)
PERQZeroMor Base zeroMor = let open ZeroMor zeroMor in record
  {botMor       = PERQBotMor₀ Base botMor leftZeroLaw rightZeroLaw
  ; leftZeroLaw  = PERQLeftZeroLaw₀ Base leftZeroLaw rightZeroLaw
  ; rightZeroLaw = PERQRightZeroLaw₀ Base leftZeroLaw rightZeroLaw
  }
```

17.7 Categoric.PERQ.KSGC

```
PERQTransClosOp : {i j k₁ k₂ : Level} {Obj : Set i}
                → (Base : USLSGC {i} j k₁ k₂ Obj)
                → (transClosOp : TransClosOp (USLSGC.uslSemigroupoid Base))
                → TransClosOp (PERQUSLSemigroupoid Base)
PERQTransClosOp Base transClosOp = let
  open USLSGC Base
  open TransClosOp transClosOp
  open SymIdempot
  open PERQCSG convSemigroupoid
  open PERQMor
  in record
    {_⁺ = λ {A} R → record
      {mor = (mor R) ⁺
      ; closed = ≈-begin
          ⟪ A ⟫ ⨾ (mor R) ⁺ ⨾ ⟪ A ⟫
        ≈⟨ ⨾-cong₂₁ (⁺-cong (≈-sym (Lclosed R))) ⟩
          ⟪ A ⟫ ⨾ (⟪ A ⟫ ⨾ mor R) ⁺ ⨾ ⟪ A ⟫
        ≈⟨ ⨾-cong₂ ⁺-⨾-roll ⟩
          ⟪ A ⟫ ⨾ ⟪ A ⟫ ⨾ (mor R ⨾ ⟪ A ⟫) ⁺
        ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong (idempotent A) (⁺-cong (Rclosed R)) ⟩
```

```
          ⟪ A ⟫ ⨾ (mor R) ⁺
        ≈⟨ ⨾-cong₂ ⁺-recDef₁ ⟩
          ⟪ A ⟫ ⨾ (mor R ⊔ mor R ⨾ (mor R) ⁺)
        ≈⟨ ⨾-⊔-distribR ⟩
          ⟪ A ⟫ ⨾ mor R ⊔ ⟪ A ⟫ ⨾ mor R ⨾ (mor R) ⁺
        ≈⟨ ⊔-cong (Lclosed R) (⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (Lclosed R)) ⟩
          mor R ⊔ mor R ⨾ (mor R) ⁺
        ≈⟨ ≈-sym ⁺-recDef₁ ⟩
          (mor R) ⁺
        □
      }
    ; ⁺-recDef₁     = ⁺-recDef₁
    ; ⁺-recDef₂     = ⁺-recDef₂
    ; ⁺-leftInd     = ⁺-leftInd
    ; ⁺-rightInd    = ⁺-rightInd
    }
```

```
PERQKleeneSemigroupoid : {i j k₁ k₂ : Level} {Obj : Set i}
  → (Base : KSGC {i} j k₁ k₂ Obj)
  → KleeneSemigroupoid {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (KSGC.SymIdempot Base)
PERQKleeneSemigroupoid Base = let open KSGC Base in record
  {uslSemigroupoid = PERQUSLSemigroupoid uslsgc
  ; transClosOp     = PERQTransClosOp      uslsgc transClosOp
  }
```

```
PERQKSGC : {i j k₁ k₂ : Level} {Obj : Set i}
  → (Base : KSGC {i} j k₁ k₂ Obj)
  → KSGC {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (KSGC.SymIdempot Base)
PERQKSGC Base = let open KSGC Base in record
  {uslsgc       = PERQUSLSGC      uslsgc
  ; transClosOp = PERQTransClosOp uslsgc transClosOp
  }
```

17.8 Categoric.PERQ.KCC

[WK: *This should use a general construction of a StarOp from a TransClosOp and an IdOp.* **]**

```
PERQStarOp : {i j k₁ k₂ : Level} {Obj : Set i}
           → (Base : KSGC {i} j k₁ k₂ Obj)
           → StarOp (PERQUSLCategory (KSGC.uslsgc Base))
PERQStarOp Base = let
  open KSGC Base
  open SymIdempot
  open PERQCSG convSemigroupoid
  open PERQMor
  open KSGC (PERQKSGC Base) using () renaming
             (Mor to Mor′
             ; _⁺ to _⁺′
             )
  _*′ : {A : SymIdempot} → Mor′ A A → Mor′ A A
  _*′ {A} R = let R⁺ = R ⁺′ in record
             {mor = ⟪ A ⟫ ⊔ mor R⁺
             ; closed = ≈-begin
                 ⟪ A ⟫ ⨾ (⟪ A ⟫ ⊔ mor R⁺) ⨾ ⟪ A ⟫
               ≈⟨ ⨾-cong₂ (⨾-⊔-distribL ⟨≈≈⟩ ⊔-cong₁ (idempotent A)) ⟩
```

```
                ⟪ A ⟫ ⨾ (⟪ A ⟫ ⊔ mor R⁺ ⨾ ⟪ A ⟫)
              ≈⟨ ⨾-⊔-distribR ⟨≈≈⟩ ⊔-cong₁ (idempotent A) ⟩
                ⟪ A ⟫ ⊔ ⟪ A ⟫ ⨾ mor R⁺ ⨾ ⟪ A ⟫
              ≈⟨ ⊔-cong₂ (closed R⁺) ⟩
                ⟪ A ⟫ ⊔ mor R⁺
              □
            }
  in record
    {_* = _*′
    ; isStar = λ {A} R → record
      {*-recDef = let R⁺ = R ⁺′; R* = R *′ in ≈-sym (≈-begin
                ⟪ A ⟫ ⊔ mor R ⊔ mor R* ⨾ mor R*
              ≈⟨ ≈-refl ⟩
                ⟪ A ⟫ ⊔ mor R ⊔ mor R* ⨾ (⟪ A ⟫ ⊔ mor R⁺)
              ≈⟨ ⊔-cong₂₂ (⨾-⊔-distribR ⟨≈≈⟩ ⊔-cong₁ (Rclosed R*)) ⟩
                ⟪ A ⟫ ⊔ mor R ⊔ mor R* ⊔ mor R* ⨾ mor R⁺
              ≈⟨ ⊔-assocL₃₊₁ ⟩
                (⟪ A ⟫ ⊔ mor R ⊔ mor R*) ⊔ mor R* ⨾ mor R⁺
              ≈⟨ ⊔-cong (≈-begin
                      ⟪ A ⟫ ⊔ mor R ⊔ mor R*
                    ≈⟨ ≈-refl ⟩
                      ⟪ A ⟫ ⊔ mor R ⊔ ⟪ A ⟫ ⊔ mor R⁺
                    ≈⟨ ⊔-assocL₃₊₁ ⟨≈≈⟩ ⊔-cong₁₂ ⊔-commutative ⟩
                      (⟪ A ⟫ ⊔ ⟪ A ⟫ ⊔ mor R) ⊔ mor R⁺
                    ≈⟨ ⊔-cong₁ (⊔-assocL ⟨≈≈⟩ ⊔-cong₁ ⊔-idempotent) ⟩
                      (⟪ A ⟫ ⊔ mor R) ⊔ mor R⁺
                  □) (≈-begin
                      mor R* ⨾ mor R⁺
                    ≈⟨ ≈-refl ⟩
                      (⟪ A ⟫ ⊔ mor R⁺) ⨾ mor R⁺
                    ≈⟨ ⨾-⊔-distribL ⟨≈≈⟩ ⊔-cong₁ (Lclosed R⁺) ⟩
                      mor R⁺ ⊔ mor R⁺ ⨾ mor R⁺
                  □) ⟩
                (((⟪ A ⟫ ⊔ mor R) ⊔ mor R⁺) ⊔ mor R⁺ ⊔ mor R⁺ ⨾ mor R⁺
              ≈⟨ ⊔-assoc ⟨≈≈⟩ ⊔-cong₂ (⊔-assocL ⟨≈≈⟩ ⊔-cong₁ ⊔-idempotent) ⟩
                (⟪ A ⟫ ⊔ mor R) ⊔ mor R⁺ ⊔ mor R⁺ ⨾ mor R⁺
              ≈⟨ ⊔-assoc ⟨≈≈⟩ ⊔-cong₂ (⊔-assocL ⟨≈≈⟩ ⊔-cong₁ ⊔-commutative) ⟩
                ⟪ A ⟫ ⊔ (mor R⁺ ⊔ mor R) ⊔ mor R⁺ ⨾ mor R⁺
              ≈⟨ ⊔-cong₂ (⊔-assoc ⟨≈≈⟩ ⊔-cong₂ (≈-sym ⁺-recDef)) ⟩
                ⟪ A ⟫ ⊔ mor R⁺ ⊔ mor R⁺
              ≈⟨ ⊔-cong₂ ⊔-idempotent ⟩
                ⟪ A ⟫ ⊔ mor R⁺ □)
      ; *-leftInd = λ {B} {S} R⨾S⊑S → let R⁺ = R ⁺′; R* = R *′ in ⊑-begin
                mor R* ⨾ mor S
              ≈⟨ ≈-refl ⟩
                (⟪ A ⟫ ⊔ mor R⁺) ⨾ mor S
              ⊑⟨ ⨾-⊔-subdistribL ⟨⊑≈⟩ ⊔-cong₁ (Lclosed S) ⟩
                mor S ⊔ mor R⁺ ⨾ mor S
              ⊑⟨ ⊔-monotone₂ (⁺-leftInd R⨾S⊑S) ⟩
                mor S ⊔ mor S
              ≈⟨ ⊔-idempotent ⟩
                mor S □
      ; *-rightInd        = λ {B} {Q} Q⨾R⊑Q → let R⁺ = R ⁺′; R* = R *′ in ⊑-begin
                mor Q ⨾ mor R*
              ≈⟨ ≈-refl ⟩
                mor Q ⨾ (⟪ A ⟫ ⊔ mor R⁺)
              ⊑⟨ ⨾-⊔-subdistribR ⟨⊑≈⟩ ⊔-cong₁ (Rclosed Q) ⟩
                mor Q ⊔ mor Q ⨾ mor R⁺
              ⊑⟨ ⊔-monotone₂ (⁺-rightInd Q⨾R⊑Q) ⟩
```

```
                mor Q ⊔ mor Q
              ≈⟨ ⊔-idempotent ⟩
                mor Q □
      }
    }
```

For the time being, we only provide Kleene categories based on PERQBotMor$_0$, i.e., assuming zero laws in the underlying KSGC.

```
PERQKleeneCategory₀ : {i j k₁ k₂ : Level} {Obj : Set i}
                    → (Base : KSGC {i} j k₁ k₂ Obj)
                    → (zeroMor : ZeroMor (KSGC.orderedSemigroupoid Base))
                    → KleeneCategory {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (KSGC.SymIdempot Base)
PERQKleeneCategory₀ Base zeroMor = let open KSGC Base in record
  {uslCategory = PERQUSLCategory uslsgc
  ;zeroMor     = PERQZeroMor     osgc zeroMor
  ;starOp      = PERQStarOp      Base
  }


PERQKCC₀ : {i j k₁ k₂ : Level} {Obj : Set i}
         → (Base : KSGC {i} j k₁ k₂ Obj)
         → (zeroMor : ZeroMor (KSGC.orderedSemigroupoid Base))
         → KCC {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (KSGC.SymIdempot Base)
PERQKCC₀ Base zeroMor = let open KSGC Base in record
  {uslcc    = PERQUSLCC   uslsgc
  ;zeroMor  = PERQZeroMor osgc zeroMor
  ;starOp   = PERQStarOp  Base
  }
```

17.9 Categoric.PERQ.DistrLatCC

Meet preservation requires modal rules, so we need to start from semi-allegories and semi-collagories for instantiating the homset (lower semi-)lattice theories.

```
PERQMeetOp : {i j k₁ k₂ : Level} {Obj : Set i}
           → (Base : SemiAllegory {i} j k₁ k₂ Obj)
           → MeetOp (PERQOrderedSemigroupoid (SemiAllegory.osgc Base))
PERQMeetOp Base = let open SemiAllegory Base in record
  {meet = λ {A} {B} R S → let
    open SymIdempot
    open PERQCSG convSemigroupoid
    open PERQMor
    in record
      {value = record
            {mor = mor R ⊓ mor S
            ;closed = ⊑-antisym (⊑-begin
                ⟪ A ⟫ ⨾ (mor R ⊓ mor S) ⨾ ⟪ B ⟫
              ⊑⟨ ⨾-monotone₂ ⨾-⊓-subdistribL ⟩
                ⟪ A ⟫ ⨾ (mor R ⨾ ⟪ B ⟫ ⊓ mor S ⨾ ⟪ B ⟫)
              ⊑⟨ ⨾-⊓-subdistribR ⟩
                ⟪ A ⟫ ⨾ mor R ⨾ ⟪ B ⟫ ⊓ ⟪ A ⟫ ⨾ mor S ⨾ ⟪ B ⟫
              ≈⟨ ⊓-cong (closed R) (closed S) ⟩
                mor R ⊓ mor S
              □) (⊑-begin
                mor R ⊓ mor S
              ≈˘⟨ ⊓-cong₁ (Lclosed R) ⟩
```

```
                  ⟪ A ⟫ ⨾ mor R ⊓ mor S
                ⊑⟨ modal₁ ⟩
                  ⟪ A ⟫ ⨾ (mor R ⊓ ⟪ A ⟫ ˘ ⨾ mor S)
                ≈⟨ ⨾-cong₂ (⊓-cong₂ (˘Lclosed S)) ⟩
                  ⟪ A ⟫ ⨾ (mor R ⊓ mor S)
                ⊑⟨ ⨾-monotone₂ (⊑-begin
                                  mor R ⊓ mor S
                                ≈˘⟨ ⊓-cong₁ (Rclosed R) ⟩
                                  mor R ⨾ ⟪ B ⟫ ⊓ mor S
                                ⊑⟨ modal₂ ⟩
                                  (mor R ⊓ mor S ⨾ ⟪ B ⟫ ˘) ⨾ ⟪ B ⟫
                                ≈⟨ ⨾-cong₁ (⊓-cong₂ (˘Rclosed S)) ⟩
                                  (mor R ⊓ mor S) ⨾ ⟪ B ⟫
                                □) ⟩
                  ⟪ A ⟫ ⨾ (mor R ⊓ mor S) ⨾ ⟪ B ⟫
                □)}
      ; proof = record
            {bound₁ = ⊓-lower₁
            ; bound₂ = ⊓-lower₂
            ; universal = λ X⊑R X⊑S → ⊓-universal (X⊑R) (X⊑S)
            }
      }
  }

PERQLSLSemigroupoid : {i j k₁ k₂ : Level} {Obj : Set i}
                    → (Base : SemiAllegory {i} j k₁ k₂ Obj)
                    → LSLSemigroupoid {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (SemiAllegory.SymIdempot Base)
PERQLSLSemigroupoid Base = let open SemiAllegory Base in record
  {orderedSemigroupoid = PERQOrderedSemigroupoid osgc
  ; meetOp              = PERQMeetOp Base
  }

PERQLatticeSemigroupoid : {i j k₁ k₂ : Level} {Obj : Set i}
                        → (Base : SemiCollagory {i} j k₁ k₂ Obj)
                        → LatticeSemigroupoid {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (SemiCollagory.SymIdempot Base)
PERQLatticeSemigroupoid Base = let open SemiCollagory Base in record
  {orderedSemigroupoid = PERQOrderedSemigroupoid osgc
  ; meetOp              = PERQMeetOp              semiAllegory
  ; joinOp              = PERQJoinOp              uslsgc
  }

PERQHomLatticeDistr : {i j k₁ k₂ : Level} {Obj : Set i}
                    → (Base : SemiCollagory {i} j k₁ k₂ Obj)
                    → HomLatticeDistr (PERQLatticeSemigroupoid Base)
PERQHomLatticeDistr Base = let
  open SemiCollagory Base
  open PERQCSG convSemigroupoid
  open PERQMor
  in record
    {⊓-⊔-subdistribR = λ {A} {B} {Q} {R} {S} → ⊓-⊔-subdistribR {Q = mor Q} {mor R} {mor S}}

PERQDistrLatSemigroupoid : {i j k₁ k₂ : Level} {Obj : Set i}
                         → (Base : SemiCollagory {i} j k₁ k₂ Obj)
                         → DistrLatSemigroupoid {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (SemiCollagory.SymIdempot Base)
PERQDistrLatSemigroupoid Base = let open SemiCollagory Base in record
  {latticeSemigroupoid = PERQLatticeSemigroupoid Base
  ; homLatDistr        = PERQHomLatticeDistr       Base
```

```
  ; joinCompDistrL      = PERQJoinCompDistrL      uslsgc
  ; joinCompDistrR      = PERQJoinCompDistrR      uslsgc
  }
```

```
PERQDistrLatSGC : {i j k₁ k₂ : Level} {Obj : Set i}
                → (Base : SemiCollagory {i} j k₁ k₂ Obj)
                → DistrLatSGC {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (SemiCollagory.SymIdempot Base)
PERQDistrLatSGC Base = let open SemiCollagory Base in record
  {osgc              = PERQOSGC              osgc
  ; meetOp           = PERQMeetOp            semiAllegory
  ; joinOp           = PERQJoinOp            uslsgc
  ; joinCompDistrL   = PERQJoinCompDistrL uslsgc
  ; joinCompDistrR   = PERQJoinCompDistrR uslsgc
  ; homLatDistr      = PERQHomLatticeDistr Base
  }
```

```
PERQDistrLatCC : {i j k₁ k₂ : Level} {Obj : Set i}
               → (Base : SemiCollagory {i} j k₁ k₂ Obj)
               → DistrLatCC {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (SemiCollagory.SymIdempot Base)
PERQDistrLatCC Base = let open SemiCollagory Base in record
  {occ               = PERQOCC               osgc
  ; meetOp           = PERQMeetOp            semiAllegory
  ; joinOp           = PERQJoinOp            uslsgc
  ; homLatDistr      = PERQHomLatticeDistr Base
  ; joinCompDistrL   = PERQJoinCompDistrL uslsgc
  ; joinCompDistrR   = PERQJoinCompDistrR uslsgc
  }
```

17.10 Categoric.PERQ.DistrAllegory

```
PERQAllegory : {i j k₁ k₂ : Level} {Obj : Set i}
  → (Base : SemiAllegory {i} j k₁ k₂ Obj)
  → Allegory {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (SemiAllegory.SymIdempot Base)
PERQAllegory Base = let open SemiAllegory Base in record
  {occ         = PERQOCC      osgc
  ; meetOp     = PERQMeetOp Base
  ; Dedekind   = Dedekind
  }
```

```
PERQCollagory : {i j k₁ k₂ : Level} {Obj : Set i}
  → (Base : SemiCollagory {i} j k₁ k₂ Obj)
  → Collagory {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (SemiCollagory.SymIdempot Base)
PERQCollagory Base = let open SemiCollagory Base in record
  {allegory          = PERQAllegory          semiAllegory
  ; joinOp           = PERQJoinOp            uslsgc
  ; homLatDistr      = PERQHomLatticeDistr Base
  ; joinCompDistrL   = PERQJoinCompDistrL uslsgc
  ; joinCompDistrR   = PERQJoinCompDistrR uslsgc
  }
```

For the time being, we only provide distributive allegories based on $\mathsf{PERQBotMor}_0$, i.e., assuming zero laws in the underlying semiallegory.

```
PERQDistrAllegory₀ : {i j k₁ k₂ : Level} {Obj : Set i}
                   → (Base : DistrSemiAllegory {i} j k₁ k₂ Obj)
```

```
                          → DistrAllegory {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (DistrSemiAllegory.SymIdempot Base)
PERQDistrAllegory₀ Base = let open DistrSemiAllegory Base in record
  {collagory = PERQCollagory semiCollagory
  ; zeroMor  = PERQZeroMor osgc zeroMor
  }
```

17.11 Categoric.PERQ.DivAllegory

```
PERQLeftResOp : {i j k₁ k₂ : Level} {Obj : Set i}
  → (Base : OSGC {i} j k₁ k₂ Obj)
  → (leftResOp : LeftResOp (OSGC.orderedSemigroupoid Base))
  → LeftResOp (PERQOrderedSemigroupoid Base)
PERQLeftResOp Base leftResOp = let
    open OSGC Base
    open LeftResOp leftResOp
    open SymIdempot
    open PERQCSG convSemigroupoid
    open PERQMor
    _/′_ : {A B C : SymIdempot} (S : PERQMor A C) (R : PERQMor B C) → PERQMor A B
    _/′_ {A} {B} {C} S R = record
          {mor = ⟪ A ⟫ ⨾ (mor S / mor R) ⨾ ⟪ B ⟫
          ; closed = ≈-begin
                     ⟪ A ⟫ ⨾ (⟪ A ⟫ ⨾ (mor S / mor R) ⨾ ⟪ B ⟫) ⨾ ⟪ B ⟫
                   ≈⟨ ⨾-cong₂ (⨾-assoc₃₊₁ ⟨≈≈⟩ ⨾-cong₂₂ (idempotent B)) ⟩
                     ⟪ A ⟫ ⨾ ⟪ A ⟫ ⨾ (mor S / mor R) ⨾ ⟪ B ⟫
                   ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (idempotent A) ⟩
                     ⟪ A ⟫ ⨾ (mor S / mor R) ⨾ ⟪ B ⟫ □
          }
  in record
    {_/_ =     _/′_
    ; /-cancel-outer = λ {A B C S R} → ⊑-begin
                     mor (S /′ R) ⨾ mor R
                   ≈⟨ ≈-refl ⟩
                     (⟪ A ⟫ ⨾ (mor S / mor R) ⨾ ⟪ B ⟫) ⨾ mor R
                   ≈⟨ ⨾-assoc₃₊₁ ⟨≈≈⟩ ⨾-cong₂₂ (Lclosed R) ⟩
                     ⟪ A ⟫ ⨾ (mor S / mor R) ⨾ mor R
                   ⊑⟨ ⨾-monotone₂ /-cancel-outer ⟩
                     ⟪ A ⟫ ⨾ mor S
                   ≈⟨ Lclosed S ⟩
                     mor S        □
    ; /-universal = λ {A B C S R Q} Q⨾R⊑S → ⊑-begin
                     mor Q
                   ≈⟨ ≈-sym (closed Q) ⟩
                     ⟪ A ⟫ ⨾ mor Q ⨾ ⟪ B ⟫
                   ⊑⟨ ⨾-monotone₂₁ (/-universal Q⨾R⊑S) ⟩
                     ⟪ A ⟫ ⨾ (mor S / mor R) ⨾ ⟪ B ⟫
                   ≈⟨ ≈-refl ⟩
                     mor (S /′ R) □
    }
```

```
PERQRightResOp : {i j k₁ k₂ : Level} {Obj : Set i}
  → (Base : OSGC {i} j k₁ k₂ Obj)
  → (rightResOp : RightResOp (OSGC.orderedSemigroupoid Base))
  → RightResOp (PERQOrderedSemigroupoid Base)
PERQRightResOp Base rightResOp = let
    open OSGC Base
```

```
    open RightResOp rightResOp
    open SymIdempot
    open PERQCSG convSemigroupoid
    open PERQMor
    _\′_ : {A B C : SymIdempot} (Q : PERQMor A B) (S : PERQMor A C) → PERQMor B C
    _\′_ {A} {B} {C} Q S = record
         {mor = ⟪ B ⟫ ⨾ (mor Q \ mor S) ⨾ ⟪ C ⟫
         ; closed = ≈-begin
             ⟪ B ⟫ ⨾ (⟪ B ⟫ ⨾ (mor Q \ mor S) ⨾ ⟪ C ⟫) ⨾ ⟪ C ⟫
           ≈⟨ ⨾-cong₂ (⨾-assoc₃₊₁ ⟨≈≈⟩ ⨾-cong₂₂ (idempotent C)) ⟩
             ⟪ B ⟫ ⨾ ⟪ B ⟫ ⨾ (mor Q \ mor S) ⨾ ⟪ C ⟫
           ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (idempotent B) ⟩
             ⟪ B ⟫ ⨾ (mor Q \ mor S) ⨾ ⟪ C ⟫ □
         }
  in record
    {_\_ = _\′_
    ; \-cancel-outer = λ {A B C S Q} → ⊑-begin
             mor Q ⨾ ⟪ B ⟫ ⨾ (mor Q \ mor S) ⨾ ⟪ C ⟫
           ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (Rclosed Q) ⟩
             mor Q ⨾ (mor Q \ mor S) ⨾ ⟪ C ⟫
           ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ \-cancel-outer ⟩
             mor S ⨾ ⟪ C ⟫
           ≈⟨ Rclosed S ⟩
             mor S        □
    ; \-universal = λ {A B C S Q R} Q⨾R⊑S → ⊑-begin
             mor R
           ≈⟨ ≈-sym (closed R) ⟩
             ⟪ B ⟫ ⨾ mor R ⨾ ⟪ C ⟫
           ⊑⟨ ⨾-monotone₂₁ (\-universal Q⨾R⊑S) ⟩
             ⟪ B ⟫ ⨾ (mor Q \ mor S) ⨾ ⟪ C ⟫
           ≈⟨ ≈-refl ⟩
             mor (Q \′ S) □
    }

PERQSyqOp : {i j k₁ k₂ : Level} {Obj : Set i}
             → (Base : OSGC {i} j k₁ k₂ Obj)
             → (syqOp : SyqOp Base)
             → SyqOp (PERQOSGC Base)
PERQSyqOp Base syqOp = let
    open OSGC Base
    open SyqOp syqOp
    open SymIdempot
    open PERQCSG convSemigroupoid
    open PERQMor
    _〳′_ : {A B C : SymIdempot} (Q : PERQMor A B) (S : PERQMor A C) → PERQMor B C
    _〳′_ {A} {B} {C} Q S = record
      {mor = ⟪ B ⟫ ⨾ (mor Q 〳 mor S) ⨾ ⟪ C ⟫
      ; closed = ≈-begin
           ⟪ B ⟫ ⨾ (⟪ B ⟫ ⨾ (mor Q 〳 mor S) ⨾ ⟪ C ⟫) ⨾ ⟪ C ⟫
        ≈⟨ ⨾-cong₂ (⨾-assoc₃₊₁ ⟨≈≈⟩ ⨾-cong₂₂ (idempotent C)) ⟩
           ⟪ B ⟫ ⨾ ⟪ B ⟫ ⨾ (mor Q 〳 mor S) ⨾ ⟪ C ⟫
        ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (idempotent B) ⟩
           ⟪ B ⟫ ⨾ (mor Q 〳 mor S) ⨾ ⟪ C ⟫    □
      }
  in record
    {_〳_ = _〳′_
    ; 〳-cong = λ {A B C Q₁ Q₂ S₁ S₂} Q₁≈Q₂ S₁≈S₂ → ≈-begin
           ⟪ B ⟫ ⨾ (mor Q₁ 〳 mor S₁) ⨾ ⟪ C ⟫
        ≈⟨ ⨾-cong₂₁ (〳-cong Q₁≈Q₂ S₁≈S₂) ⟩
```

```
          ⟪ B ⟫ ⨾ (mor Q₂ ╳ mor S₂) ⨾ ⟪ C ⟫ □
  ; ╳-cancel-left  =  λ {A B C Q S} → ⊑-begin
          mor Q ⨾ ⟪ B ⟫ ⨾ (mor Q ╳ mor S) ⨾ ⟪ C ⟫
       ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (Rclosed Q) ⟩
          mor Q ⨾ (mor Q ╳ mor S) ⨾ ⟪ C ⟫
       ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ ╳-cancel-left ⟩
          mor S ⨾ ⟪ C ⟫
       ≈⟨ Rclosed S ⟩
          mor S          □
  ; ╳-cancel-right  =  λ {A B C Q S} → ⊑-begin
          mor (Q ╳′ S) ⨾ mor S ˘
       ≈⟨ ≈-refl ⟩
          (⟪ B ⟫ ⨾ (mor Q ╳ mor S) ⨾ ⟪ C ⟫) ⨾ mor S ˘
       ≈⟨ ⨾-assoc₃₊₁ ⟨≈≈⟩ ⨾-cong₂₂ (˘-involutionRightConv ⟨≈˘≈⟩ ˘-cong (˘Rclosed S)) ⟩
          ⟪ B ⟫ ⨾ (mor Q ╳ mor S) ⨾ mor S ˘
       ⊑⟨ ⨾-monotone₂ ╳-cancel-right ⟩
          ⟪ B ⟫ ⨾ mor Q ˘
       ≈⟨ ˘-involutionRightConv ⟨≈˘≈⟩ ˘-cong (˘Rclosed Q) ⟩
          mor Q ˘        □
  ; ╳-universal  =  λ       {A B C Q S R} Q⨾R⊑S R⨾S˘⊑Q˘ → ⊑-begin
          mor R
       ≈⟨ ≈-sym (closed R) ⟩
          ⟪ B ⟫ ⨾ mor R ⨾ ⟪ C ⟫
       ⊑⟨ ⨾-monotone₂₁ (╳-universal Q⨾R⊑S R⨾S˘⊑Q˘) ⟩
          ⟪ B ⟫ ⨾ (mor Q ╳ mor S) ⨾ ⟪ C ⟫
       ≈⟨ ≈-refl ⟩
          mor (Q ╳′ S) □
  }

PERQDivAllegory :  {i j k₁ k₂ : Level} {Obj : Set i}
                   → (Base : DivSemiAllegory {i} j k₁ k₂ Obj)
                   → DivAllegory {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (DivSemiAllegory.SymIdempot Base)
PERQDivAllegory Base  =  let open DivSemiAllegory Base in record
  {distrAllegory  =  PERQDistrAllegory₀ distrSemiAllegory
  ; leftResOp     =  PERQLeftResOp      osgc leftResOp
  ; rightResOp    =  PERQRightResOp     osgc rightResOp
  ; syqOp         =  PERQSyqOp          osgc syqOp
  }
```

17.12 Categoric.PERQ.DistrActAllegory

```
PERQDistrActAllegory :  {i j k₁ k₂ : Level} {Obj : Set i}
                        → (Base : DivSemiAllegory {i} j k₁ k₂ Obj)
                        → (transClosOp : TransClosOp (DivSemiAllegory.uslSemigroupoid Base))
                        → DistrActAllegory {i ⊍ j ⊍ k₁} (j ⊍ k₁) k₁ k₂ (DivSemiAllegory.SymIdempot Base)
PERQDistrActAllegory Base transClosOp  =  let open DivSemiAllegory Base using (uslsgc) in record
  {divAllegory     =  PERQDivAllegory Base
  ; starOp         =  PERQStarOp     (record {uslsgc  =  uslsgc; transClosOp  =  transClosOp})
  }
```

Part III

Concrete Relations

Chapter 18

Properties of Concrete Relations

In Sect. 18.2, we define a fully universe-polymorphic type of concrete relations that is compatible with the standard library, which does not provide typical relation-algebraic operations and laws.

We then define standard relation-algebraic operations and prove their properties, which is necessarily very similar to the corresponding part of the AoPA library of Mu et al. (2009), which however supports heterogeneous binary relations only at the levels 0 and, to a lesser degree, 1.

The main use of the material in this chapter will be to prove, in Chapter 19, that our concrete relations are models of the abstract relation-algebraic theories defined in chapters 3–13.

18.1 Relation.Binary.Heterogeneous

This module only re-exports all the material below it, to provide a single interface for import.

```
open import Relation.Binary.Heterogeneous.Base                    public
open import Relation.Binary.Heterogeneous.Props.Inclusion         public
open import Relation.Binary.Heterogeneous.Props.Equivalence       public
open import Relation.Binary.Heterogeneous.Props.Poset             public
open import Relation.Binary.Heterogeneous.Props.Meet              public
open import Relation.Binary.Heterogeneous.Props.Join              public
open import Relation.Binary.Heterogeneous.Props.Converse          public
open import Relation.Binary.Heterogeneous.Props.Composition       public
open import Relation.Binary.Heterogeneous.Props.Residuals         public
open import Relation.Binary.Heterogeneous.Props.RestrResiduals    public
open import Relation.Binary.Heterogeneous.Props                   public
open import Relation.Binary.Heterogeneous.Props.Props             public
open import Relation.Binary.Heterogeneous.Props.Domain            public
open import Relation.Binary.Heterogeneous.Props.Range             public
open import Relation.Binary.Heterogeneous.Props.Plus              public

open import Relation.Binary.Heterogeneous.GenPropEq               public
open import Relation.Binary.Heterogeneous.Props.Identity          public
open import Relation.Binary.Heterogeneous.Props.GenPropEq         public
open import Relation.Binary.Heterogeneous.Props.PropEqProps       public
open import Relation.Binary.Heterogeneous.Props.Star              public
```

18.2 Relation.Binary.Heterogeneous.Base

We define fully level-polymorphic heterogeneous binary relations with a different argument order than the standard library (which uses the identifier REL), but otherwise in the same way, so we have:

$$\mathcal{R}el\ \mathsf{k}\ \mathsf{A}\ \mathsf{B}\ =\ \mathsf{REL}\ \mathsf{A}\ \mathsf{B}\ \mathsf{k}$$

Via this equality, our 𝓡eℓ-based library is fully interoperable with the standard library.

However, with our argument order, we more directly obtain the homset function for categories, allegories, etc. of concrete of relations at level k as 𝓡eℓ k.

```
𝓡eℓ : (k : Level) → {i j : Level} → Set i → Set j → Set (i ⊍ j ⊍ ℓsuc k)
𝓡eℓ k A B = A → B → Set k
```

We also introduce some alternative infix and mixfix symbols:

```
infixr 8 _⟷⟨_⟩_ _⟷_ _⟷₀_ _⟷₁_
_⟷⟨_⟩_ : ∀ {i j} → Set i → (k : Level) → Set j → Set (i ⊍ j ⊍ ℓsuc k)
A ⟷⟨ k ⟩ B = 𝓡eℓ k A B
_⟷_ : {k i j : Level} → Set i → Set j → Set (i ⊍ j ⊍ ℓsuc k)
_⟷_ {k} = 𝓡eℓ k
_⟷₀_ : {i j : Level} → Set i → Set j → Set (ℓsuc ℓ₀ ⊍ i ⊍ j)
_⟷₀_ = 𝓡eℓ ℓ₀
_⟷₁_ : {i j : Level} → Set i → Set j → Set (ℓsuc (ℓsuc ℓ₀) ⊍ i ⊍ j)
_⟷₁_ = 𝓡eℓ (ℓsuc ℓ₀)
```

We now define basic predicates and operations on binary relations.

Inclusion

In contrast with _⇒_ from the standard library, we make the arguments x and y *explicit* arguments. This makes more involved proofs involving relations much easier; for example, the proofs in Sect. 18.23 would require far more explicit specification of the relations arguments involved in many of the proof steps there if inclusion were to take x and y as implicit arguments.

Inclusion is a relation on relations — the equivalent view of a binary predicate on relations is given as a comment in the type of _⊆_:

```
infixr 4 _⊆_
_⊆_ : {i j k₁ k₂ : Level} {A : Set i} {B : Set j}
    -- → 𝓡eℓ k₁ A B → 𝓡eℓ k₂ A B → Set (i ⊍ j ⊍ k₁ ⊍ k₂)
  → 𝓡eℓ (i ⊍ j ⊍ k₁ ⊍ k₂) (𝓡eℓ k₁ A B) (𝓡eℓ k₂ A B)
P ⊆ Q = ∀ x y → P x y → Q x y
```

Union

```
infixr 7 _∪_ _∩_
_∪_ : ∀ {i j k k'} {A : Set i} {B : Set j}
    → (R : 𝓡eℓ k A B)
    → (S : 𝓡eℓ k' A B)
    → 𝓡eℓ (k ⊍ k') A B
(R ∪ S) a b = R a b ⊎ S a b
```

Intersection

```
_∩_ : ∀ {i j k k'} {A : Set i} {B : Set j}
    → (R : 𝓡eℓ k A B)
    → (S : 𝓡eℓ k' A B)
    → 𝓡eℓ (k ⊍ k') A B
(R ∩ S) a b = R a b × S a b
```

Relation Equivalence

```
infixr 4 _≑_
_≑_ : ∀ {i j k₁ k₂} → {A : Set i} → {B : Set j} → 𝓡el k₁ A B → 𝓡el k₂ A B → Set _
R ≑ S = (R ⊆ S) × (S ⊆ R)
```

Composition

```
_⨾_ : ∀ {i j l k k'} {A : Set i} {B : Set j} {C : Set l}
     → (R : 𝓡el k {i} {j} A B)
     → (S : 𝓡el k' {j} {l} B C)
     → 𝓡el (j ⊍ k ⊍ k') {i} {l} A C
(R ⨾ S) a c = ∃ (λ b → R a b × S b c)
```

Converse

```
_˘ : ∀ {i j k} → {A : Set i} → {B : Set j} → 𝓡el k A B → 𝓡el k B A
_˘ = flip
```

Functions as Relations

Without generalised propositional equality we cannot achieve full Level polymorphism:

```
fun : {i j : Level} {A : Set i} {B : Set j} → (A → B) → 𝓡el j A B
fun f a b = (f a ≡ b)
```

Domain

(Producing a subidentity):

```
dom : ∀ {i j k} {A : Set i} {B : Set j}
     → (R : 𝓡el k A B)
     → 𝓡el (i ⊍ j ⊍ k) A A
dom {k = k} R a a′ = ∃ (λ b → R a b) × a ≡ a′
```

(Here, and in ran, additional level polymorphism would be possible by replacing ≡ with ≡≡ and making its level a separate argument.)

Range

```
ran : ∀ {i j k} {A : Set i} {B : Set j}
    → (R : 𝓡el k A B)
    → 𝓡el (i ⊍ j ⊍ k) B B
ran {k = k} R b b′ = ∃ (λ a → R a b) × b ≡ b′
```

Residuals

The residuals of composition translate into universally quantified implications:

Left residual:

```
_/_ : ∀ {i j l k k'} {A : Set i} {B : Set j} {C : Set l}
    → (S : 𝓡𝑒ℓ k  A C)
    → (R : 𝓡𝑒ℓ k' B C)
    → 𝓡𝑒ℓ (l ⊍ k ⊍ k') A B
(S / R) a b = ∀ c → R b c → S a c
```

Right residual:

```
infixl 9 _/_ _⌿_
infixr 9 _\_ _⍀_
_\_ : ∀ {i j l k k'} {A : Set i} {B : Set j} {C : Set l}
    → (Q : 𝓡𝑒ℓ k A B)
    → (S : 𝓡𝑒ℓ k' A C)
    → 𝓡𝑒ℓ (i ⊍ k ⊍ k') B C
(Q \ S) b c = ∀ a → Q a b → S a c
```

Restricted Residuals

The restricted residuals Kahl (2008) add an existence statement:

Restricted left residual:

```
_⌿_ : ∀ {i j l k k'} {A : Set i} {B : Set j} {C : Set l}
    → (S : 𝓡𝑒ℓ k  A C)
    → (R : 𝓡𝑒ℓ k' B C)
    → 𝓡𝑒ℓ (l ⊍ k ⊍ k') A B
(S ⌿ R) a b = (∀ c → R b c → S a c) × ∃ (λ c → R b c)
```

Restricted right residual:

```
_⍀_ : ∀ {i j l k k'} {A : Set i} {B : Set j} {C : Set l}
    → (Q : 𝓡𝑒ℓ k A B)
    → (S : 𝓡𝑒ℓ k' A C)
    → 𝓡𝑒ℓ (i ⊍ k ⊍ k') B C
(Q ⍀ S) b c = (∀ a → Q a b → S a c) × ∃ (λ a → Q a b)
```

18.3 Relation.Binary.PropositionalEquality.Generalised

We define a fully Level-polymorphic variant of propositional equality in order to be able to define fully Level-polymorphic identity relations. (Thorsten Altenkirch[1] and others have expressed reservations whether such a definition at levels k lower than a should be legal.)

```
infix 4 _≡≡_
data _≡≡_ {k a : Level} {A : Set a} (x : A) : A → Set k where
  ≡≡-refl : x ≡≡ x
≡≡-sym : {k a : Level} {A : Set a} → Symmetric {a} {k} (_≡≡_ {A = A})
≡≡-sym ≡≡-refl = ≡≡-refl
```

[1] at AIM XIII, April 2011

```
≡≡-trans : {k a : Level} {A : Set a} → Transitive {a} {k} (_≡≡_ {A = A})
≡≡-trans ≡≡-refl ≡≡-refl = ≡≡-refl

≡≡-subst : {k a p : Level} {A : Set a} → Substitutive {a} {k} (_≡≡_ {A = A}) p
≡≡-subst P ≡≡-refl p = p

≡≡-resp₂ : {k a ℓ : Level} {A : Set a} (~ : Rel A ℓ) → ~ Respects₂ (_≡≡_ {k} {A = A})
≡≡-resp₂ _~_ = subst⟶resp₂ _~_ ≡≡-subst
```

We also lift the "inspect idiom" from the standard library:

```
data Inspect≡≡ k {a} {A : Set a} (x : A) : Set (a ⊍ k) where
  _with-≡≡_ : (y : A) (eq : _≡≡_ {k} x y) → Inspect≡≡ k x

inspect≡≡ : ∀ k {a} {A : Set a} (x : A) → Inspect≡≡ k x
inspect≡≡ k x = x with-≡≡ ≡≡-refl
```

18.4 Relation.Binary.Heterogeneous.Props.Inclusion

Relation inclusion ⊆ is a preorder.

For the reflexivity and transitivity proofs, we include versions ⊆-Refl and ⊆-Trans with explicit relation arguments, since using those, e.g. as ⊆-Refl Q, is notationally easier than explicitly supplying the last implicit argument, e.g. as ⊆-refl {R = Q}, where that implicit argument cannot be inferred.

```
⊆-Refl : {i j k : Level} {A : Set i} {B : Set j} → (R : 𝓡ℯ𝓁 k A B) → R ⊆ R
⊆-Refl _ x y p = p

⊆-refl : {i j k : Level} {A : Set i} {B : Set j} → {R : 𝓡ℯ𝓁 k A B} → R ⊆ R
⊆-refl x y p = p

⊆-reflexive : {i j k : Level} {A : Set i} {B : Set j} → _≡_ ⇒ _⊆_ {i} {j} {k} {k} {A} {B}
⊆-reflexive {i} {j} {k} {A} {B} {R} {.R} ≡-refl = ⊆-Refl R

⊆-Trans : {i j k₁ k₂ k₃ : Level} {A : Set i} {B : Set j}
        → (Q : 𝓡ℯ𝓁 k₁ A B) → (R : 𝓡ℯ𝓁 k₂ A B) → (S : 𝓡ℯ𝓁 k₃ A B)
        → Q ⊆ R → R ⊆ S → Q ⊆ S
⊆-Trans _ _ _ qr rs x y p = rs x y (qr x y p)   -- rs (qr p)

⊆-trans : {i j k₁ k₂ k₃ : Level} {A : Set i} {B : Set j}
        → {Q : 𝓡ℯ𝓁 k₁ A B} → {R : 𝓡ℯ𝓁 k₂ A B} → {S : 𝓡ℯ𝓁 k₃ A B}
        → Q ⊆ R → R ⊆ S → Q ⊆ S
⊆-trans qr rs x y p = rs x y (qr x y p)   -- rs (qr p)

⊆-IsPreorder : {i j k : Level} {A : Set i} {B : Set j} → IsPreorder _≡_ (_⊆_ {i} {j} {k} {k} {A} {B})
⊆-IsPreorder {i} {j} {k} {A} {B} = record
  {isEquivalence = PropEq.isEquivalence
  ; reflexive    = ⊆-reflexive
  ; trans        = ⊆-trans
  }

⊆-Preorder : {i j k : Level} (A : Set i) (B : Set j) → Preorder (i ⊍ j ⊍ ℓsuc k) (i ⊍ j ⊍ ℓsuc k) (i ⊍ j ⊍ k)
⊆-Preorder {i} {j} {k} A B = record
  {Carrier = 𝓡ℯ𝓁 k A B
  ; _≈_ = _≡_
  ; _∼_ = _⊆_
  ; isPreorder = ⊆-IsPreorder {i} {j} {k} {A} {B}
  }
```

18.5 Relation.Binary.Heterogeneous.Props.Equivalence

For any two sets A and B, relation equivalence ≑ on relations from A to B is indeed an equivalence, and we use that to define relation setoids.

```
≑-refl : {i j k : Level} {A : Set i} {B : Set j} → {R : 𝓡eℓ k A B} → R ≑ R
≑-refl {i} {j} {k} {A} {B} {R} = ⊆-Refl R, ⊆-Refl R

≑-sym : {i j k₁ k₂ : Level} {A : Set i} {B : Set j}
      → {R : 𝓡eℓ k₁ A B} → {S : 𝓡eℓ k₂ A B} → R ≑ S → S ≑ R
≑-sym (p, q) = q, p

≑-trans : {i j k₁ k₂ k₃ : Level} {A : Set i} {B : Set j}
        → {R : 𝓡eℓ k₁ A B} → {S : 𝓡eℓ k₂ A B} → {T : 𝓡eℓ k₃ A B} → R ≑ S → S ≑ T → R ≑ T
≑-trans {i} {j} {k₁} {k₂} {k₃} {A} {B} {R} {S} {T} (rs, sr) (st, ts)
  = ⊆-Trans R S T rs st
  , ⊆-Trans T S R ts sr

≑-equiv : {i j : Level} (k : Level) (A : Set i) (B : Set j) → IsEquivalence {i ⊍ j ⊍ ℓsuc k} {i ⊍ j ⊍ k} {𝓡eℓ k A B} _≑_
≑-equiv k A B = record
  {refl = ≑-refl
  ; sym = ≑-sym
  ; trans = ≑-trans
  }

≑-Setoid : {i j : Level} (k : Level) (A : Set i) (B : Set j) → Setoid (i ⊍ j ⊍ ℓsuc k) (i ⊍ j ⊍ k)
≑-Setoid k A B = record
  {Carrier = 𝓡eℓ k A B
  ; _≈_ = _≑_
  ; isEquivalence = ≑-equiv k A B
  }
```

18.6 Relation.Binary.Heterogeneous.Props.Poset

We now show that relation inclusion ⊆ is an order if relation equivalence ≑ is considered as the underlying equality, and define the resulting Posets of relations from A to B.

```
⊆-≑-reflexive : {i j k : Level} {A : Set i} {B : Set j} → _≑_ ⇒ _⊆_ {i} {j} {k} {k} {A} {B}
⊆-≑-reflexive {i} {j} {k} {A} {B} {R} {S} = proj₁

⊆-≑-resp₂ : {i j k : Level} {A : Set i} {B : Set j} → (_⊆_ {i} {j} {k} {k} {A} {B}) Respects₂ _≑_
⊆-≑-resp₂ = (λ r=s q⊆r x y q → proj₁ r=s x y (q⊆r x y q))
          , (λ r=s r⊆q x y s → r⊆q x y (proj₂ r=s x y s))

⊆-≑-IsPreorder : (k : Level) {i j : Level} (A : Set i) (B : Set j)
               → IsPreorder _≑_ (_⊆_ {i} {j} {k} {k} {A} {B})
⊆-≑-IsPreorder k A B = record
  {isEquivalence = ≑-equiv k A B
  ; reflexive     = ⊆-≑-reflexive
  ; trans         = ⊆-trans
  }

⊆-≑-antisym : {i j k₁ k₂ : Level} {A : Set i} {B : Set j}
            → {Q : 𝓡eℓ k₁ A B} → {R : 𝓡eℓ k₂ A B}
            → Q ⊆ R → R ⊆ Q → Q ≑ R
⊆-≑-antisym = _,_

⊆-IsPartialOrder : (k : Level) {i j : Level} (A : Set i) (B : Set j)
                 → IsPartialOrder {i ⊍ j ⊍ ℓsuc k} {i ⊍ j ⊍ k} {i ⊍ j ⊍ k} _≑_ _⊆_
⊆-IsPartialOrder k A B = record
  {isPreorder = ⊆-≑-IsPreorder k A B
  ; antisym = ⊆-≑-antisym
  }

⊆-Poset : (k : Level) {i j : Level} (A : Set i) (B : Set j) → Poset (i ⊍ j ⊍ ℓsuc k) (i ⊍ j ⊍ k) (i ⊍ j ⊍ k)
⊆-Poset k A B = record
  {Carrier = 𝓡eℓ k A B
  ; _≈_ = _≑_
  ; _≤_ = _⊆_
```

```
 ; isPartialOrder = ⊆-IsPartialOrder k A B
 }
```

18.7 Relation.Binary.Heterogeneous.Props.Meet

Intersection returns the greatest lower bound of its arguments:

```
∩-lower₁ : {i₁ i₂ k₁ k₂ : Level}
         → {A : Set i₁} {B : Set i₂}
         → {R₁ : 𝓡𝑒𝓁 k₁ A B}
         → {R₂ : 𝓡𝑒𝓁 k₂ A B}
         → R₁ ∩ R₂ ⊆ R₁
∩-lower₁ {R₁ = R₁} {R₂} x y (Rxy, Sxy) = Rxy
∩-lower₂ : {i₁ i₂ k₁ k₂ : Level}
         → {A : Set i₁} {B : Set i₂}
         → {R₁ : 𝓡𝑒𝓁 k₁ A B}
         → {R₂ : 𝓡𝑒𝓁 k₂ A B}
         → R₁ ∩ R₂ ⊆ R₂
∩-lower₂ {R₁ = R₁} {R₂} x y (Rxy, Sxy) = Sxy
∩-universal : {i₁ i₂ k₁ k₂ k₃ : Level}
            → {A : Set i₁} {B : Set i₂}
            → {R₁ : 𝓡𝑒𝓁 k₁ A B}
            → {R₂ : 𝓡𝑒𝓁 k₂ A B}
            → {X : 𝓡𝑒𝓁 k₃ A B}
            → X ⊆ R₁
            → X ⊆ R₂
            → X ⊆ R₁ ∩ R₂
∩-universal {R₁ = R₁} {R₂} {X} X⊆R₁ X⊆R₂ x y Xxy = X⊆R₁ x y Xxy, X⊆R₂ x y Xxy
```

18.8 Relation.Binary.Heterogeneous.Props.Join

Union returns the least upper bound of its arguments:

```
∪-upper₁ : {i₁ i₂ k₁ k₂ : Level}
         → {A : Set i₁} {B : Set i₂}
         → {R₁ : 𝓡𝑒𝓁 k₁ A B}
         → {R₂ : 𝓡𝑒𝓁 k₂ A B}
         → R₁ ⊆ R₁ ∪ R₂
∪-upper₁ {R₁ = R₁} {R₂} x y Rxy = inj₁ Rxy
∪-upper₂ : {i₁ i₂ k₁ k₂ : Level}
         → {A : Set i₁} {B : Set i₂}
         → {R₁ : 𝓡𝑒𝓁 k₁ A B}
         → {R₂ : 𝓡𝑒𝓁 k₂ A B}
         → R₂ ⊆ R₁ ∪ R₂
∪-upper₂ {R₁ = R₁} {R₂} x y Sxy = inj₂ Sxy
∪-universal : {i₁ i₂ k₁ k₂ k₃ : Level}
            → {A : Set i₁} {B : Set i₂}
            → {R₁ : 𝓡𝑒𝓁 k₁ A B}
            → {R₂ : 𝓡𝑒𝓁 k₂ A B}
            → {X : 𝓡𝑒𝓁 k₃ A B}
            → R₁ ⊆ X
            → R₂ ⊆ X
            → R₁ ∪ R₂ ⊆ X
∪-universal {R₁ = R₁} {R₂} {X} R₁⊆X R₂⊆X x y (inj₁ R₁xy) = R₁⊆X x y R₁xy
∪-universal {R₁ = R₁} {R₂} {X} R₁⊆X R₂⊆X x y (inj₂ R₂xy) = R₂⊆X x y R₂xy
```

18.9 Relation.Binary.Heterogeneous.Props.Converse

The converse operation is monotone, self-inverse, and involutory with respect to composition.

```
˘-Monotone : {i₁ i₂ k₁ k₂ : Level}
           → {A : Set i₁} {B : Set i₂}
           → (R₁ : 𝓡ℯℓ k₁ A B)
           → (R₂ : 𝓡ℯℓ k₂ A B)
           → R₁ ⊆ R₂ → R₁ ˘ ⊆ R₂ ˘
˘-Monotone _ _ leqR y x xy = leqR x y xy

˘-monotone : {i₁ i₂ k₁ k₂ : Level}
           → {A : Set i₁} {B : Set i₂}
           → {R₁ : 𝓡ℯℓ k₁ A B}
           → {R₂ : 𝓡ℯℓ k₂ A B}
           → R₁ ⊆ R₂ → R₁ ˘ ⊆ R₂ ˘
˘-monotone leqR y x xy = leqR x y xy

˘-cong : {i₁ i₂ k₁ k₂ : Level}
           → {A : Set i₁} {B : Set i₂}
           → {R₁ : 𝓡ℯℓ k₁ A B}
           → {R₂ : 𝓡ℯℓ k₂ A B}
           → R₁ ≑ R₂ → R₁ ˘ ≑ R₂ ˘
˘-cong {R₁ = R₁} {R₂} (leqR, geqR) = ˘-Monotone R₁ R₂ leqR
                                    , ˘-Monotone R₂ R₁ geqR

˘˘-Increasing  : {i₁ i₂ k : Level}
               → {A : Set i₁} {B : Set i₂}
               → (R : 𝓡ℯℓ k A B)
               → R ⊆ (R ˘) ˘
˘˘-Increasing R x y xy = ⊆-Refl R x y xy

˘˘-Decreasing : {i₁ i₂ k : Level}
               → {A : Set i₁} {B : Set i₂}
               → (R : 𝓡ℯℓ k A B)
               → (R ˘) ˘ ⊆ R
˘˘-Decreasing R x y xy = ⊆-Refl R x y xy

˘˘-Id : {i₁ i₂ k : Level}
               → {A : Set i₁} {B : Set i₂}
               → (R : 𝓡ℯℓ k A B)
               → (R ˘) ˘ ≑ R
˘˘-Id R =  ˘˘-Decreasing R
        ,  ˘˘-Increasing R

˘-SubInvolute : {i₁ i₂ i₃ k₁ k₂ : Level}
           → {A : Set i₁} {B : Set i₂} {C : Set i₃}
           → (R : 𝓡ℯℓ k₁ A B)
           → (S : 𝓡ℯℓ k₂ B C)
           → (R ⨾ S) ˘ ⊆ S ˘ ⨾ R ˘
˘-SubInvolute R S z x (y, (xy, yz)) = (y, (yz, xy))

˘-SupInvolute : {i₁ i₂ i₃ k₁ k₂ : Level}
           → {A : Set i₁} {B : Set i₂} {C : Set i₃}
           → (R : 𝓡ℯℓ k₁ A B)
           → (S : 𝓡ℯℓ k₂ B C)
           → S ˘ ⨾ R ˘ ⊆ (R ⨾ S) ˘
˘-SupInvolute R S z x (y, (yz, xy)) = (y, (xy, yz))

˘-Involute :   {i₁ i₂ i₃ k₁ k₂ : Level}
           → {A : Set i₁} {B : Set i₂} {C : Set i₃}
           → (R : 𝓡ℯℓ k₁ A B)
           → (S : 𝓡ℯℓ k₂ B C)
           → (R ⨾ S) ˘ ≑ S ˘ ⨾ R ˘
˘-Involute R S = ˘-SubInvolute R S
```

```
        , ˘-SupInvolute R S
```

18.10 Relation.Binary.Heterogeneous.Props.Composition

Relation composition ⨾ is monotone with respect to inclusion ⊆ in both arguments, preserves equivalence ≑, and is associative.

The main use of these properties is to populate the records in Chapter 19 proving that concrete relations form models of the abstract theories of chapters 3–12. Therefore, we use the signatures that will be needed in Chapter 19, although the relation arguments would be more naturally left explicit here, since relation composition is not a constructor, and therefore does not communicate its arguments to the Agda type checker.

```
⨾-monotone : {i₁ i₂ i₃ k₁ k₂ k₃ k₄ : Level}
           → {A : Set i₁} {B : Set i₂} {C : Set i₃}
           → {R₁ : 𝓡𝓮𝓵 k₁ A B}
           → {R₂ : 𝓡𝓮𝓵 k₂ A B}
           → {S₁ : 𝓡𝓮𝓵 k₃ B C}
           → {S₂ : 𝓡𝓮𝓵 k₄ B C}
           → R₁ ⊆ R₂ → S₁ ⊆ S₂
           → R₁ ⨾ S₁ ⊆ R₂ ⨾ S₂
⨾-monotone leqR leqS x z (y, (xy, yz)) = y, (leqR x y xy, leqS y z yz)
⨾-cong : {i₁ i₂ i₃ k₁ k₂ k₃ k₄ : Level}
           → {A : Set i₁} {B : Set i₂} {C : Set i₃}
           → {R₁ : 𝓡𝓮𝓵 k₁ A B}
           → {R₂ : 𝓡𝓮𝓵 k₂ A B}
           → {S₁ : 𝓡𝓮𝓵 k₃ B C}
           → {S₂ : 𝓡𝓮𝓵 k₄ B C}
           → R₁ ≑ R₂ → S₁ ≑ S₂
           → R₁ ⨾ S₁ ≑ R₂ ⨾ S₂
⨾-cong {R₁ = R₁} {R₂} {S₁} {S₂} (leqR, geqR) (leqS, geqS)
           = ⨾-monotone {R₁ = R₁} {R₂} {S₁} {S₂} leqR leqS
           , ⨾-monotone {R₁ = R₂} {R₁} {S₂} {S₁} geqR geqS
⨾-assocR : {i₁ i₂ i₃ i₄ k₁ k₂ k₃ : Level}
         → {A : Set i₁} {B : Set i₂} {C : Set i₃} {D : Set i₄}
         → {R : 𝓡𝓮𝓵 k₁ A B}
         → {S : 𝓡𝓮𝓵 k₂ B C}
         → {T : 𝓡𝓮𝓵 k₃ C D}
         → ((R ⨾ S) ⨾ T) ⊆ (R ⨾ (S ⨾ T))
⨾-assocR a d (c, (b, abR, bcS), cdT) = b, (abR, c, bcS, cdT)
⨾-assocL : {i₁ i₂ i₃ i₄ k₁ k₂ k₃ : Level}
         → {A : Set i₁} {B : Set i₂} {C : Set i₃} {D : Set i₄}
         → {R : 𝓡𝓮𝓵 k₁ A B}
         → {S : 𝓡𝓮𝓵 k₂ B C}
         → {T : 𝓡𝓮𝓵 k₃ C D}
         → (R ⨾ (S ⨾ T)) ⊆ ((R ⨾ S) ⨾ T)
⨾-assocL a d (b, (abR, c, bcS, cdT)) = (c, (b, abR, bcS), cdT)
⨾-assoc  : {i₁ i₂ i₃ i₄ k₁ k₂ k₃ : Level}
         → {A : Set i₁} {B : Set i₂} {C : Set i₃} {D : Set i₄}
         → {R : 𝓡𝓮𝓵 k₁ A B}
         → {S : 𝓡𝓮𝓵 k₂ B C}
         → {T : 𝓡𝓮𝓵 k₃ C D}
         → ((R ⨾ S) ⨾ T) ≑ (R ⨾ (S ⨾ T))
⨾-assoc {R = R} {S} {T}
  = ⨾-assocR {R = R} {S} {T}
  , ⨾-assocL {R = R} {S} {T}
```

18.11 Relation.Binary.Heterogeneous.Props.Residuals

```
/-cancel-outer : {i₁ i₂ i₃ k₁ k₂ : Level}
               → {A : Set i₁} {B : Set i₂} {C : Set i₃}
               → {S : 𝓡𝓮𝓵 k₁ A C}
               → {R : 𝓡𝓮𝓵 k₂ B C}
               → (S / R) ⨾ R ⊆ S
/-cancel-outer {R = R} x z (y, res, Ryz) = res z Ryz
/-universal :  {i₁ i₂ i₃ k₁ k₂ k₃ : Level}
            →  {A : Set i₁} {B : Set i₂} {C : Set i₃}
            →  {S : 𝓡𝓮𝓵 k₁ A C}
            →  {R : 𝓡𝓮𝓵 k₂ B C}
            →  {Q : 𝓡𝓮𝓵 k₃ A B}
            →  Q ⨾ R ⊆ S → Q ⊆ S / R
/-universal Q⨾R⊆S x y Qxy = λ z Ryz → Q⨾R⊆S x z (y, Qxy, Ryz)
```

```
\-cancel-outer : {i₁ i₂ i₃ k₁ k₂ : Level}
               → {A : Set i₁} {B : Set i₂} {C : Set i₃}
               → {S : 𝓡𝓮𝓵 k₁ A C}
               → {Q : 𝓡𝓮𝓵 k₂ A B}
               → Q ⨾ (Q \ S) ⊆ S
\-cancel-outer {Q = Q} x z (y, Qxy, res) = res x Qxy
\-universal :  {i₁ i₂ i₃ k₁ k₂ k₃ : Level}
            →  {A : Set i₁} {B : Set i₂} {C : Set i₃}
            →  {S : 𝓡𝓮𝓵 k₁ A C}
            →  {Q : 𝓡𝓮𝓵 k₂ A B}
            →  {R : 𝓡𝓮𝓵 k₃ B C}
            →  Q ⨾ R ⊆ S → R ⊆ Q \ S
\-universal Q⨾R⊆S y z Ryz = λ x Qxy → Q⨾R⊆S x z (y, Qxy, Ryz)
```

18.12 Relation.Binary.Heterogeneous.Props.RestrResiduals

The following are the basic properties of restricted residuals required in Sect. 13.4.

```
⌿-cancel-outer :  {i₁ i₂ i₃ k₁ k₂ : Level} {A : Set i₁} {B : Set i₂} {C : Set i₃}
               → {S : 𝓡𝓮𝓵 k₁ A C} {R : 𝓡𝓮𝓵 k₂ B C}
               → (S ⌿ R) ⨾ R ⊆ S
⌿-cancel-outer {R = R} x z (y, (resU, resE), Ryz) = resU z Ryz
⌿-restr :  {i₁ i₂ i₃ k₁ k₂ : Level} {A : Set i₁} {B : Set i₂} {C : Set i₃}
        → {S : 𝓡𝓮𝓵 k₁ A C} {R : 𝓡𝓮𝓵 k₂ B C}
        → ran (S ⌿ R) ⊆ dom R
⌿-restr {R = R} y .y ((x, (resU, (z, Ryz))), ≡-refl) = ((z, Ryz), ≡-refl)
⌿-universal :  {i₁ i₂ i₃ k₁ k₂ k₃ : Level} {A : Set i₁} {B : Set i₂} {C : Set i₃}
            →  {S : 𝓡𝓮𝓵 k₁ A C} {R : 𝓡𝓮𝓵 k₂ B C} {Q : 𝓡𝓮𝓵 k₃ A B}
            →  Q ⨾ R ⊆ S → ran Q ⊆ dom R → Q ⊆ S ⌿ R
⌿-universal Q⨾R⊆S ranQ⊆domR x y Qxy
  = (λ z Ryz → Q⨾R⊆S x z (y, Qxy, Ryz))
  ,  proj₁ (ranQ⊆domR y y ((x, Qxy), ≡-refl))
```

```
⍀-cancel-outer :  {i₁ i₂ i₃ k₁ k₂ : Level} {A : Set i₁} {B : Set i₂} {C : Set i₃}
               → {S : 𝓡𝓮𝓵 k₁ A C} {Q : 𝓡𝓮𝓵 k₂ A B}
               → Q ⨾ (Q ⍀ S) ⊆ S
⍀-cancel-outer {Q = Q} x z (y, Qxy, (resU, resE)) = resU x Qxy
⍀-restr :  {i₁ i₂ i₃ k₁ k₂ : Level} {A : Set i₁} {B : Set i₂} {C : Set i₃}
```

```
        → {S : 𝓡𝓮𝓵 k₁ A C} {Q : 𝓡𝓮𝓵 k₂ A B}
        → dom (Q ⍀ S) ⊆ ran Q
⍀-restr {Q = Q} y .y ((z, (resU, (x, Qxy))), ≡-refl) = ((x, Qxy), ≡-refl)
⍀-universal : {i₁ i₂ i₃ k₁ k₂ k₃ : Level} {A : Set i₁} {B : Set i₂} {C : Set i₃}
            → {S : 𝓡𝓮𝓵 k₁ A C} {Q : 𝓡𝓮𝓵 k₂ A B} {R : 𝓡𝓮𝓵 k₃ B C}
            → Q ⨾ R ⊆ S → dom R ⊆ ran Q → R ⊆ Q ⍀ S
⍀-universal Q⨾R⊆S domR⊆ranQ y z Ryz
  = (λ x Qxy → Q⨾R⊆S x z (y, Qxy, Ryz))
  , proj₁ (domR⊆ranQ y y ((z, Ryz), ≡-refl))
```

18.13 Relation.Binary.Heterogeneous.Props

For relation-algebraic versions of properties involving generalised propositional equality, in particular via identity relations, see Sect. 18.20.

Reflexivity

The predicate-logic definition of reflexivity:

```
reflexive : {i k : Level} {A : Set i} → 𝓡𝓮𝓵 k A A → Set (i ⊍ k)
reflexive {k = k} {A} R = {x : A} → R x x
```

Coreflexivity

The predicate-logic definition of coreflexivity, without generalised propositional equality:

```
coreflexive : {i k : Level} {A : Set i} → 𝓡𝓮𝓵 k A A → Set (i ⊍ k)
coreflexive {k = k} {A} R = (x y : A) → R x y → x ≡ y
```

Symmetry

The relation-algebraic definition of symmetry:

```
Symmetric : {i k : Level} {A : Set i} → 𝓡𝓮𝓵 k A A → Set (i ⊍ k)
Symmetric {k = k} R = R ˘ ⊆ R
```

The predicate-logic definition of symmetry:

```
symmetric : {i k : Level} {A : Set i} → 𝓡𝓮𝓵 k A A → Set (i ⊍ k)
symmetric {k = k} {A} R = (x y : A) → R y x → R x y
```

Again, these define the same function:

```
Symmetric≡symmetric : {i j k : Level} {A : Set i} → Symmetric {i} {k} {A} ≡ symmetric {i} {k} {A}
Symmetric≡symmetric = ≡-refl
```

CompIn

CompIn is generalised from the standard library's Trans.

```
CompIn : {i₁ i₂ i₃ k₁ k₂ k₃ : Level} {A : Set i₁} {B : Set i₂} {C : Set i₃}
       → (P : A ⟷⟨ k₁ ⟩ B) → (Q : B ⟷⟨ k₂ ⟩ C) → (R : A ⟷⟨ k₃ ⟩ C)
```

```
         → Set _
CompIn P Q R = ∀ {x y z} → P x y → Q y z → R x z
```

CompIn P Q R is equivalent to P ⨾ Q ⊆ R:

```
CompIn-expand : {i₁ i₂ i₃ k₁ k₂ k₃ : Level} {A : Set i₁} {B : Set i₂} {C : Set i₃}
  → {P : A ⟷⟨ k₁ ⟩ B} → {Q : B ⟷⟨ k₂ ⟩ C} → {R : A ⟷⟨ k₃ ⟩ C}
  → CompIn P Q R → P ⨾ Q ⊆ R
CompIn-expand incomp a c (b, (aPb, bQc)) = incomp {a} {b} {c} aPb bQc
CompIn-contract : {i₁ i₂ i₃ k₁ k₂ k₃ : Level}
  → {A : Set i₁} {B : Set i₂} {C : Set i₃}
  → {P : A ⟷⟨ k₁ ⟩ B}
  → {Q : B ⟷⟨ k₂ ⟩ C}
  → {R : A ⟷⟨ k₃ ⟩ C}
  → P ⨾ Q ⊆ R → CompIn P Q R
CompIn-contract P⨾Q⊆R {a} {b} {c} aPb bQc = P⨾Q⊆R a c (b, (aPb, bQc))
```

Transitivity

The relation-algebraic definition of transitivity:

```
Transitive : {i k : Level} {A : Set i} → 𝓡𝓮𝓵 k A A → Set (i ⊍ k)
Transitive {k = k} R = R ⨾ R ⊆ R
```

The predicate-logic definition of transitivity:

```
transitive : {i k : Level} {A : Set i} → 𝓡𝓮𝓵 k A A → Set (i ⊍ k)
transitive {k = k} {A} R = {x y z : A} → R x y → R y z → R x z
```

Their equivalence follows from the fact that transitive R = CompIn R R R:

```
Transitive-transitive : {i j k : Level} {A : Set i} → (R : 𝓡𝓮𝓵 k A A) → Transitive R → transitive R
Transitive-transitive R = CompIn-contract
transitive-Transitive : {i j k : Level} {A : Set i} → (R : 𝓡𝓮𝓵 k A A) → transitive R → Transitive R
transitive-Transitive R = CompIn-expand
```

Cotransitivity

The relation-algebraic definition of cotransitivity:

```
Cotransitive : {i k : Level} {A : Set i} → 𝓡𝓮𝓵 k A A → Set (i ⊍ k)
Cotransitive {k = k} R = R ⊆ R ⨾ R
```

The predicate-logic definition of cotransitivity:

```
cotransitive : {i k : Level} {A : Set i} → 𝓡𝓮𝓵 k A A → Set (i ⊍ k)
cotransitive {k = k} {A} R = (x z : A) → R x z → Σ [y : A] R x y × R y z
```

These define the same function:

```
Cotransitive≡cotransitive : {i k : Level} {A : Set i} → Cotransitive {i} {k} {A} ≡ cotransitive {i} {k} {A}
Cotransitive≡cotransitive = ≡-refl
```

Idempotence

Idempotence is just the conjunction of cotransitivity and transitivity:

```
Idempotent : {i k : Level} {A : Set i} → 𝓡ℯℓ k A A → Set (i ⊍ k)
Idempotent {k = k} R = R ⨾ R ≑ R
```

Univalence

The predicate-logic definition of univalence:

```
univalent : {i j k : Level} {A : Set i} {B : Set j} → 𝓡ℯℓ k A B → Set (i ⊍ j ⊍ k)
univalent {A = A} {B} R = (x : A) (y₁ y₂ : B) → R x y₁ → R x y₂ → y₁ ≡ y₂
```

Totality

The predicate-logic definition of totality:

```
relTotal : {i j k : Level} {A : Set i} {B : Set j} → 𝓡ℯℓ k A B → Set (i ⊍ j ⊍ k)
relTotal {A = A} {B} R = (a : A) → Σ [ b : B ] R a b
```

Univalent and total relations are called *mapping*s:

```
relIsMapping : {i j k : Level} {A : Set i} {B : Set j} → 𝓡ℯℓ k A B → Set (i ⊍ j ⊍ k)
relIsMapping R = univalent R × relTotal R
```

18.14 Relation.Binary.Heterogeneous.Props.Props

Coreflexive relations are symmetric:

```
coreflexive-symmetric : {i k : Level} {A : Set i} {R : 𝓡ℯℓ k A A}
                      → coreflexive R → symmetric R
coreflexive-symmetric coreflR x y Ryx with coreflR y x Ryx
coreflexive-symmetric coreflR x .x Ryx | refl = Ryx
```

Coreflexive relations are transitive and cotransitive, and therefore idempotent:

```
coreflexive-Transitive   : {i k : Level} {A : Set i} {R : 𝓡ℯℓ k A A}
  →                        coreflexive R → Transitive R
coreflexive-Transitive coreflR x z (y, Rxy, Ryz) with coreflR x y Rxy
coreflexive-Transitive coreflR x z (.x, Rxy, Ryz) | refl = Ryz
coreflexive-Cotransitive : {i k : Level} {A : Set i} {R : 𝓡ℯℓ k A A}
  →                        coreflexive R → Cotransitive R
coreflexive-Cotransitive coreflR x y Rxy with coreflR x y Rxy
coreflexive-Cotransitive coreflR x .x Rxx | refl = (x, Rxx, Rxx)
coreflexive-Idempotent : {i k : Level} {A : Set i} {R : 𝓡ℯℓ k A A}
  →                        coreflexive R → Idempotent R
coreflexive-Idempotent coreflR
  = coreflexive-Transitive coreflR
  , coreflexive-Cotransitive coreflR
```

Coreflexive relations are also subidentities with respect to composition:

```
coreflexive-IsLeftSubidentity : {i k₁ : Level} {A : Set i} {R : 𝓡el k₁ A A}
                              → coreflexive R
                              → {j k₂ : Level} {B : Set j} {S : 𝓡el k₂ A B}
                              → R ⨾ S ⊆ S
coreflexive-IsLeftSubidentity {R = R} coreflR {S = S} x z (y, Rxy, Syz) with coreflR x y Rxy
coreflexive-IsLeftSubidentity {R = R} coreflR {S = S} x z (.x, Rxx, Sxz) | refl = Sxz
coreflexive-IsRightSubidentity : {i k₁ : Level} {A : Set i} {R : 𝓡el k₁ A A}
                               → coreflexive R
                               → {j k₂ : Level} {B : Set j} {Q : 𝓡el k₂ B A}
                               → Q ⨾ R ⊆ Q
coreflexive-IsRightSubidentity {R = R} coreflR {Q = Q} x z (y, Qxy, Ryz) with coreflR y z Ryz
coreflexive-IsRightSubidentity {R = R} coreflR {Q = Q} x z (.z, Qxz, Rzz) | refl = Qxz
```

Conversely, subidentities are also coreflexive — to be able to use this in Sect. 19.6, we only ask for the (left) subidentity property in composition with relations at the same universe level (i ⊍ k) — due to the use of propositional equality on A, the level of relations needs to be at least i, but need not be equal to i; we supply an additional level parameter k to allow more generality, and since it cannot be derived from any other parts of the type, we make it explicit:

```
leftSubidentity-coreflexive : (k : Level) {i : Level} {A : Set i} {R : 𝓡el (i ⊍ k) A A}
                            → ({B : Set i} {S : 𝓡el (i ⊍ k) A B} → R ⨾ S ⊆ S)
                            → coreflexive R
leftSubidentity-coreflexive k {i} leftSubIdR x y Rxy = lower (leftSubIdR {S = λ x y → Lift {i} {k} (x ≡ y)} x y (y, Rxy, lift refl))
```

The predicate-logic univalence condition univalent is equivalent to the relation-algebraic conditions — again we take care to allow universe-monomorphic subidentity-proofs as arguments:

```
univalent-from-isUnivalentL : (k : Level) {i : Level} {A : Set i} {B : Set i} {R : 𝓡el (i ⊍ k) A B}
                            → ({C : Set i} {S : 𝓡el (i ⊍ k) B C} → (R ˘ ⨾ R) ⨾ S ⊆ S)
                            → univalent R
univalent-from-isUnivalentL k {A} {B} {R} RunivL x y₁ y₂ xRy₁ xRy₂ = leftSubidentity-coreflexive k RunivL y₁ y₂ (x, xRy₁, xRy₂)
```

Relations created from functions via fun (Sect. 18.2) are univalent and total:

```
fun-univalent : {i j : Level} {A : Set i} {B : Set j} → (f : A → B) → univalent (fun f)
fun-univalent f x y y' xy xy' = trans (sym xy) xy'
fun-total : {i j : Level} {A : Set i} {B : Set j} → (f : A → B) → relTotal (fun f)
fun-total f x = (f x, refl)
```

A totality proof for a relation induces a total function contained in that relation:

```
totalChoiceFunction : {i j k : Level} {A : Set i} {B : Set j} (R : A ⟷⟨ k ⟩ B) → relTotal R → (A → B)
totalChoiceFunction R total a = proj₁ (total a)
totalChoiceFunction-⊆ : {i j k : Level} {A : Set i} {B : Set j}
   → (R : A ⟷⟨ k ⟩ B)
   → (total : relTotal R)
   → fun (totalChoiceFunction R total) ⊆ R
totalChoiceFunction-⊆ R total a ∘ (totalChoiceFunction R total a) refl = proj₂ (total a)
```

If a relation R is both univalent and total, then R ≑ fun (totalChoiceFunction R total).

```
univalent-totalChoiceFunction : {i j k : Level} {A : Set i} {B : Set j}
   → (R : A ⟷⟨ k ⟩ B)
   → univalent R
   → (total : relTotal R)
   → fun (totalChoiceFunction R total) ≑ R
univalent-totalChoiceFunction R unival total =
```

```
   totalChoiceFunction-⊆ R total
  , (λ a b → unival a (proj₁ (total a)) b (proj₂ (total a)))

mappingToFunction : {i j k : Level} {A : Set i} {B : Set j}
                    → (R : A ⟷⟨ k ⟩ B)
                    → relIsMapping R
                    → Σ [ f : (A → B) ] fun f ≑ R
mappingToFunction {A = A} {B} R (unival, total) =
   totalChoiceFunction R total
  , univalent-totalChoiceFunction R unival total
```

Equivalent mappings with possibly different totality proofs therefore give rise to equivalent functions:

```
mappingToFunction-cong : {i j k : Level} {A : Set i} {B : Set j}
                         → (R S : A ⟷⟨ k ⟩ B)
                         → (mapR : relIsMapping R)
                         → (mapS : relIsMapping S)
                         → R ≑ S
                         → (x : A)
                         → proj₁ (mappingToFunction R mapR) x ≡ proj₁ (mappingToFunction S mapS) x
mappingToFunction-cong R S mapR mapS R≑S x = let
  y = proj₁ (mappingToFunction R mapR) x
  xRy : R x y
  xRy = proj₁ (proj₂ (mappingToFunction R mapR)) x y refl
  xSy : S x y
  xSy = proj₁ R≑S x y xRy
  in sym (proj₂ (proj₂ (mappingToFunction S mapS)) x y xSy)
```

Relation composition of two function-relations is the function-relation arising from the corresponding function composition:

```
fun-⨾-fun-⊆ : {i₁ i₂ i₃ : Level} {A : Set i₁} {B : Set i₂} {C : Set i₃}
              → (f : A → B) → (g : B → C)
              → fun f ⨾ fun g ⊆ fun (g ∘ f)
fun-⨾-fun-⊆ f g a ∘ (g (f a)) (∘ (f a), (refl, refl)) = refl

fun-∘-⊆ : {i₁ i₂ i₃ : Level} {A : Set i₁} {B : Set i₂} {C : Set i₃}
          → (f : A → B) → (g : B → C)
          → fun (g ∘ f) ⊆ fun f ⨾ fun g
fun-∘-⊆ f g a ∘ (g (f a)) refl = (f a, (refl, refl))

fun-⨾-fun : {i₁ i₂ i₃ : Level} {A : Set i₁} {B : Set i₂} {C : Set i₃}
            → (f : A → B) → (g : B → C)
            → fun f ⨾ fun g ≑ fun (g ∘ f)
fun-⨾-fun f g = fun-⨾-fun-⊆ f g, fun-∘-⊆ f g
```

18.15 Relation.Binary.Heterogeneous.Props.SubSupId

We collect here some properties concerning super- and sub-identities that do not require generalise propositional equality.

```
refl-leftSupId  : {i k₁ : Level} {A : Set i} {P : 𝓡𝓮𝓵 k₁ A A} (refl : {x : A} → P x x)
                  → {j k₂ : Level} {B : Set j} {R : 𝓡𝓮𝓵 k₂ A B} → R ⊆ (P ⨾ R)
refl-leftSupId refl x y xRy = (x, (refl, xRy))
refl-rightSupId : {j k₂ : Level} {B : Set j} {P : 𝓡𝓮𝓵 k₂ B B} (refl : {x : B} → P x x)
                  → {i k₁ : Level} {A : Set i} {R : 𝓡𝓮𝓵 k₁ A B} → R ⊆ (R ⨾ P)
refl-rightSupId refl x y xRy = (y, (xRy, refl))
```

18.16 Relation.Binary.Heterogeneous.Props.Domain

The following are the properties required for domain in Sect. 10.2.

```
dom-coreflexive : {i j k : Level} {A : Set i} {B : Set j}
                → {R : 𝓡𝓮𝓵 k A B}
                → coreflexive (dom R)
dom-coreflexive x .x (_, ≡-refl) = ≡-refl
dom-⨾-Idempotent : {i j k : Level} {A : Set i} {B : Set j}
                 → {R : 𝓡𝓮𝓵 k A B}
                 → Idempotent (dom R)
dom-⨾-Idempotent = coreflexive-Idempotent dom-coreflexive
domPreserves⊆ : {i j k : Level} {A : Set i} {B : Set j}
              → {Q R : 𝓡𝓮𝓵 k A B}
              → Q ⊆ R → Q ⊆ dom R ⨾ Q
domPreserves⊆ Q⊆R x y Qxy = (x, ((y, Q⊆R x y Qxy), ≡-refl), Qxy)
domLeastPreserver : {i j k₁ k₂ : Level} {A : Set i} {B : Set j}
                  → {R : 𝓡𝓮𝓵 k₁ A B}
                  → {d : 𝓡𝓮𝓵 k₂ A A}
                  → coreflexive d
                  → R ⊆ d ⨾ R
                  → dom R ⊆ d
domLeastPreserver {d = d} corefl-d R⊆d⨾R x .x ((y, Rxy), ≡-refl) with R⊆d⨾R x y Rxy
... | (x′, dxx′, Rx′y) = ≡-subst (d x) (≡-sym (corefl-d x x′ dxx′)) dxx′
domLocality : {i j l k₁ k₂ : Level} {A : Set i} {B : Set j} {C : Set l}
            → {R : 𝓡𝓮𝓵 k₁ A B}
            → {S : 𝓡𝓮𝓵 k₂ B C}
            → dom (R ⨾ dom S) ⊆ dom (R ⨾ S)
domLocality x .x ((y, (.y, Rxy, ((z, Syz), ≡-refl))), ≡-refl)
  = ((z, (y, Rxy, Syz)), ≡-refl)
```

18.17 Relation.Binary.Heterogeneous.Props.Range

The following are the properties required for range in Sect. 10.2.

```
ran-coreflexive : {i j k : Level} {A : Set i} {B : Set j}
                → {R : 𝓡𝓮𝓵 k A B}
                → coreflexive (ran R)
ran-coreflexive {R = R} y .y (_, ≡-refl) = ≡-refl
ran-⨾-Idempotent : {i j k : Level} {A : Set i} {B : Set j}
                 → {R : 𝓡𝓮𝓵 k A B}
                 → Idempotent (ran R)
ran-⨾-Idempotent {R = R} = coreflexive-Idempotent (ran-coreflexive {R = R})
ranPreserves⊆ : {i j k : Level} {A : Set i} {B : Set j}
              → {Q R : 𝓡𝓮𝓵 k A B}
              → Q ⊆ R → Q ⊆ Q ⨾ ran R
ranPreserves⊆ Q⊆R x y Qxy = (y, Qxy, ((x, Q⊆R x y Qxy), ≡-refl))
ranLeastPreserver : {i j k₁ k₂ : Level} {A : Set i} {B : Set j}
                  → {R : 𝓡𝓮𝓵 k₁ A B}
                  → {d : 𝓡𝓮𝓵 k₂ B B}
                  → coreflexive d
                  → R ⊆ R ⨾ d
                  → ran R ⊆ d
ranLeastPreserver {d = d} corefl-d R⊆R⨾d y .y ((x, Rxy), ≡-refl) with R⊆R⨾d x y Rxy
... | (y′, Rxy′, dy′y) = ≡-subst (λ y′ → d y′ y) (corefl-d y′ y dy′y) dy′y
```

```
ranLocality : {i j l k₁ k₂ : Level} {A : Set i} {B : Set j} {C : Set l}
            → {R : 𝓡𝓮𝓵 k₁ A B}
            → {S : 𝓡𝓮𝓵 k₂ B C}
            → ran (ran R ⨾ S) ⊆ ran (R ⨾ S)
ranLocality {R = R} {S} z .z ((y, (.y, ((x, Rxy), ≡-refl), Syz)), ≡-refl)
  = ((x, (y, Rxy, Syz)), ≡-refl)
```

18.18 Relation.Binary.Heterogeneous.Props.Plus

The following are the properties required for the transitive closure operator in Sect. 14.1.

```
Plus-isIncreasing : {i k : Level} {A : Set i} {R : 𝓡𝓮𝓵 k A A} → R ⊆ Plus R
Plus-isIncreasing x y Rxy = [Rxy]

Plus-recDef₁⊆ : {i k : Level} {A : Set i} {R : 𝓡𝓮𝓵 k A A} → Plus R ⊆ R ∪ R ⨾ Plus R
Plus-recDef₁⊆ x z [Rxz] = inj₁ Rxz
Plus-recDef₁⊆ x z (Rxy :: RPyz) = inj₂ (_, Rxy, RPyz)

Plus-recDef₁⊇ : {i k : Level} {A : Set i} {R : 𝓡𝓮𝓵 k A A} → R ∪ R ⨾ Plus R ⊆ Plus R
Plus-recDef₁⊇ x z (inj₁ Rxz) = [Rxz]
Plus-recDef₁⊇ x z (inj₂ (y, Rxy, RPyz)) = Rxy :: RPyz

Plus-recDef₁ : {i k : Level} {A : Set i} {R : 𝓡𝓮𝓵 k A A} → Plus R ≑ R ∪ R ⨾ Plus R
Plus-recDef₁ = ⊆-≑-antisym Plus-recDef₁⊆ Plus-recDef₁⊇

Plus-recDef₂⊆ : {i k : Level} {A : Set i} {R : 𝓡𝓮𝓵 k A A} → Plus R ⊆ R ∪ Plus R ⨾ R
Plus-recDef₂⊆ x z [Rxz] = inj₁ Rxz
Plus-recDef₂⊆ x z (Rxy :: RPyz) with Plus-recDef₂⊆ _ _ RPyz
... | inj₁ Ryz = inj₂ (_, [Rxy], Ryz)
... | inj₂ (u, RPyu, Ruz) = inj₂ (u, Rxy :: RPyu, Ruz)

snoc : {i k : Level} {A : Set i} {R : 𝓡𝓮𝓵 k A A} {x y z : A} → Plus R x y → R y z → Plus R x z
snoc [Rxy] Ryz = Rxy :: [Ryz]
snoc (Rxu :: RPuy) Ryz = Rxu :: snoc RPuy Ryz

Plus-recDef₂⊇ : {i k : Level} {A : Set i} {R : 𝓡𝓮𝓵 k A A} → R ∪ Plus R ⨾ R ⊆ Plus R
Plus-recDef₂⊇ x z (inj₁ Rxz) = [Rxz]
Plus-recDef₂⊇ x z (inj₂ (y, RPxy, Ryz)) = snoc RPxy Ryz

Plus-recDef₂ : {i k : Level} {A : Set i} {R : 𝓡𝓮𝓵 k A A} → Plus R ≑ R ∪ Plus R ⨾ R
Plus-recDef₂ = ⊆-≑-antisym Plus-recDef₂⊆ Plus-recDef₂⊇

Plus-leftInd : {i k₁ : Level} {A : Set i} {R : 𝓡𝓮𝓵 k₁ A A}
             → {j k₂ : Level} {B : Set j} {S : 𝓡𝓮𝓵 k₂ A B}
             → R ⨾ S ⊆ S → Plus R ⨾ S ⊆ S
Plus-leftInd R⨾S⊆S x z (y, [Rxy], Syz) = R⨾S⊆S x z (y, Rxy, Syz)
Plus-leftInd R⨾S⊆S x z (y, _::_ {.x} {x'} {.y} Rxx' x'PlusRy, Syz)
  = R⨾S⊆S x z (x', Rxx', Plus-leftInd R⨾S⊆S x' z (y, x'PlusRy, Syz))

Plus-rightInd : {i k₁ : Level} {A : Set i} {R : 𝓡𝓮𝓵 k₁ A A}
              → {j k₂ : Level} {B : Set j} {Q : 𝓡𝓮𝓵 k₂ B A}
              → Q ⨾ R ⊆ Q → Q ⨾ Plus R ⊆ Q
Plus-rightInd {A = A} {R = R} {Q = Q} Q⨾R⊆Q x z (y, Qxy, Ryz) = h y Qxy Ryz
  where
    h : (y : A) → Q x y → Plus R y z → Q x z
    h y Qxy [Ryz] = Q⨾R⊆Q x z (y, Qxy, Ryz)
    h y Qxy (_::_ {.y} {y'} {.z} Ryy' y'PlusRz)
      = h y' (Q⨾R⊆Q x y' (y, Qxy, Ryy')) y'PlusRz
```

18.19 Relation.Binary.Heterogeneous.GenPropEq

Some aspects of fully level-polymorphic heterogeneous binary relations can only be realised using the fully level-polymorphic propositional equality of Sect. 18.3.

Identity relations use generalised propositional equality (Sect. 18.3) to achieve full Level polymorphism:

```
idR : {k i : Level} {A : Set i} → 𝓡ℯℓ k A A
idR = _≡≡_
```

Functions as relations — using generalised propositional equality, we can achieve full Level polymorphism, unlike fun in Sect. 18.2:

```
fun′ : {k i j : Level} {A : Set i} {B : Set j} → (A → B) → 𝓡ℯℓ k A B
fun′ f a b = (f a ≡≡ b)
```

18.20 Relation.Binary.Heterogeneous.Props.GenPropEq

Reflexivity

The relation-algebraic definition of reflexivity:

```
Reflexive : {i k : Level} {A : Set i} → 𝓡ℯℓ k A A → Set (i ⊍ k)
Reflexive {k = k} R = idR {k} ⊆ R
```

Equivalence with the predicate logic version reflexive from Sect. 18.13:

```
Reflexive-reflexive : {i j k : Level} {A : Set i} → (R : 𝓡ℯℓ k A A) → Reflexive R → reflexive R
Reflexive-reflexive R ReflR {x} = ReflR x x ≡≡-refl
reflexive-Reflexive : {i j k : Level} {A : Set i} → (R : 𝓡ℯℓ k A A) → reflexive R → Reflexive R
reflexive-Reflexive R reflR x .x ≡≡-refl = reflR {x}
```

Coreflexivity

The relation-algebraic definition of coreflexivity:

```
Coreflexive : {i k : Level} {A : Set i} → 𝓡ℯℓ k A A → Set (i ⊍ k)
Coreflexive {k = k} R = R ⊆ idR {k}
```

The predicate-logic definition of coreflexivity:

```
coreflexive′ : {i k : Level} {A : Set i} → 𝓡ℯℓ k A A → Set (i ⊍ k)
coreflexive′ {k = k} {A} R = (x y : A) → R x y → _≡≡_ {k} x y
```

These two definitions really define the same function (we have to supply the implicit arguments for syntactic reasons only):

```
Coreflexive≡coreflexive′ : {i k : Level} {A : Set i} → Coreflexive {i} {k} {A} ≡ coreflexive′ {i} {k} {A}
Coreflexive≡coreflexive′ = ≡-refl
```

Univalence

The relation-algebraic definition of univalence:

```
Univalent : {i j k : Level} {A : Set i} {B : Set j} → 𝓡el k A B → Set (i ⊍ j ⊍ k)
Univalent {k = k} R = R ˘ ⨾ R ⊆ idR {k}
```

The predicate-logic definition of univalence:

```
univalent′ : {i j k : Level} {A : Set i} {B : Set j} → 𝓡el k A B → Set (i ⊍ j ⊍ k)
univalent′ {k = k} {A} {B} R = (x : A) (y₁ y₂ : B)
  → R x y₁ → R x y₂ → _≡≡_ {k} y₁ y₂
```

Their equivalence:

```
Univalent-univalent′ : {i j k : Level} {A : Set i} {B : Set j}
  → (R : 𝓡el k A B) → Univalent R → univalent′ R
Univalent-univalent′ R UnivalR x y₁ y₂ xRy₁ xRy₂ = UnivalR y₁ y₂ (x, (xRy₁, xRy₂))
univalent′-Univalent : {i j k : Level} {A : Set i} {B : Set j}
  → (R : 𝓡el k A B) → univalent′ R → Univalent R
univalent′-Univalent R univalR y₁ y₂ (x, (xRy₁, xRy₂)) = univalR x y₁ y₂ xRy₁ xRy₂
```

Totality

The relation-algebraic definition of totality:

```
RelTotal : {i j k : Level} {A : Set i} {B : Set j} → 𝓡el k A B → Set (i ⊍ j ⊍ k)
RelTotal {k = k} R = idR {k} ⊆ R ⨾ R ˘
```

The predicate-logic definition of totality:

```
relTotal′ : {i j k : Level} {A : Set i} {B : Set j} → 𝓡el k A B → Set (i ⊍ j ⊍ k)
relTotal′ {k = k} {A} R = (a : A) → ∃ (λ b → R a b)
```

Their equivalence:

```
RelTotal-total′ : {i j k : Level} → {A : Set i} → {B : Set j} → (R : 𝓡el k A B) → RelTotal R → relTotal′ R
RelTotal-total′ {k = k} R TotR a with TotR a a ≡≡-refl
... | (b, (aRb, _)) = b, aRb
relTotal′-Total : {i j k : Level} → {A : Set i} → {B : Set j} → (R : 𝓡el k A B) → relTotal′ R → RelTotal R
relTotal′-Total {k = k} R totR a .a ≡≡-refl with totR a
... | (b, aRb) = (b, (aRb, aRb))
```

18.21 Relation.Binary.Heterogeneous.Props.Identity

Identity relations are left- and right-identities with respect to relation composition, and are preserved by converse.

```
leftSubId : {i j k₁ k₂ : Level} {A : Set i} {B : Set j} {R : 𝓡el k₁ A B} → (idR {k₂} ⨾ R) ⊆ R
leftSubId x y (.x, (≡≡-refl, xRy)) = xRy
leftSupId : {i j k₁ k₂ : Level} {A : Set i} {B : Set j} {R : 𝓡el k₁ A B} → R ⊆ (idR {k₂} ⨾ R)
leftSupId = refl-leftSupId ≡≡-refl
  -- Passing R explicityly is necessary in the definitions of leftId and rightId.
leftId : {i j k₁ k₂ : Level} {A : Set i} {B : Set j} {R : 𝓡el k₁ A B} → (idR {k₂} ⨾ R) ≑ R
leftId {R = R} = leftSubId {R = R}, leftSupId {R = R}
```

```
rightSubId : {i j k₁ k₂ : Level} → {A : Set i} → {B : Set j} → {R : 𝓡ℯℓ k₁ A B} → (R ⨾ idR {k₂}) ⊆ R
rightSubId x y (.y, (xRy, ≡≡-refl)) = xRy
rightSupId : {i j k₁ k₂ : Level} → {A : Set i} → {B : Set j} → {R : 𝓡ℯℓ k₁ A B} → R ⊆ (R ⨾ idR {k₂})
rightSupId = refl-rightSupId ≡≡-refl
rightId : {i j k₁ k₂ : Level} → {A : Set i} → {B : Set j} → {R : 𝓡ℯℓ k₁ A B} → (R ⨾ idR {k₂}) ≑ R
rightId {R = R} = rightSubId {R = R}, rightSupId {R = R}
idR˘-sub : {i k : Level} → {A : Set i} → idR {k} ˘ ⊆ idR {k} {i} {A}
idR˘-sub x .x ≡≡-refl = ≡≡-refl
idR˘-sup : {i k : Level} → {A : Set i} → idR {k} ⊆ idR {k} {i} {A} ˘
idR˘-sup x .x ≡≡-refl = ≡≡-refl
idR˘ : {i k : Level} → {A : Set i} → idR {k} ˘ ≑ idR {k} {i} {A}
idR˘ = idR˘-sub, idR˘-sup
```

18.22 Relation.Binary.Heterogeneous.Props.PropEqProps

Coreflexive relations are symmetric:

```
Coreflexive-Symmetric : {i k : Level} {A : Set i} {R : 𝓡ℯℓ k A A}
                      → Coreflexive R → Symmetric R
Coreflexive-Symmetric CoreflR x y Ryx with CoreflR y x Ryx
Coreflexive-Symmetric CoreflR x .x Ryx | ≡≡-refl = Ryx
```

Coreflexive relations are transitive and cotransitive, and therefore idempotent:

```
Coreflexive-Transitive   : {i k : Level} {A : Set i} {R : 𝓡ℯℓ k A A}
  →                        Coreflexive R → Transitive R
Coreflexive-Transitive CoreflR x z (y, Rxy, Ryz) with CoreflR x y Rxy
Coreflexive-Transitive CoreflR x z (.x, Rxy, Ryz) | ≡≡-refl = Ryz
Coreflexive-Cotransitive : {i k : Level} {A : Set i} {R : 𝓡ℯℓ k A A}
  →                        Coreflexive R → Cotransitive R
Coreflexive-Cotransitive CoreflR x y Rxy with CoreflR x y Rxy
Coreflexive-Cotransitive CoreflR x .x Rxy | ≡≡-refl = (_, Rxy, Rxy)
Coreflexive-Idempotent : {i k : Level} {A : Set i} {R : 𝓡ℯℓ k A A}
  →                        Coreflexive R → Idempotent R
Coreflexive-Idempotent CoreflR
   = Coreflexive-Transitive CoreflR
  , Coreflexive-Cotransitive CoreflR
```

Relations created from functions via fun (Sect. 18.2) are univalent and total:

```
fun′-Univalent : {k i j : Level} {A : Set i} {B : Set j} → (f : A → B) → Univalent {i} {j} {k} (fun′ f)
fun′-Univalent f y y′ (x, (xy, xy′)) = ≡≡-trans (≡≡-sym xy) xy′
fun′-Total : {k i j : Level} {A : Set i} {B : Set j} → (f : A → B) → RelTotal {i} {j} {k} (fun′ f)
fun′-Total f x .x ≡≡-refl = (f x, (≡≡-refl, ≡≡-refl))
```

A totality proof for a relation induces a total function contained in that relation:

```
totalChoiceFunction-⊆′ : {k₁ k₂ i j : Level} {A : Set i} {B : Set j}
                        → (R : A ⟷⟨ k₁ ⟩ B)
                        → (total : relTotal R)
                        → fun′ {k₂} (totalChoiceFunction R total) ⊆ R
totalChoiceFunction-⊆′ R total a ∘ (totalChoiceFunction R total a) ≡≡-refl = proj₂ (total a)
```

If a relation R is both univalent and total, then R ≑ fun (totalChoiceFunction R total).

```
univalent-totalChoiceFunction′ : {k i j : Level} {A : Set i} {B : Set j}
  → (R : A ⟷⟨ k ⟩ B)
  → univalent′ R
  → relTotal′ R
  → ∃ (λ (f : A → B) → fun′ {k} f ≑ R)
univalent-totalChoiceFunction′ {A = A} {B} R unival total =
   totalChoiceFunction R total
  , totalChoiceFunction-⊆′ R total
  , (λ a b → unival a (proj₁ (total a)) b (proj₂ (total a)))
```

Relation composition of two function-relations is the function-relation arising from the corresponding function composition:

```
fun′-⨾-fun′-⊆ : {k₁ k₂ k₃ i₁ i₂ i₃ : Level} {A : Set i₁} {B : Set i₂} {C : Set i₃}
  → (f : A → B) → (g : B → C)
  → fun′ {k₁} f ⨾ fun′ {k₂} g ⊆ fun′ {k₃} (g ∘ f)
fun′-⨾-fun′-⊆ f g a ∘ (g (f a)) (∘ (f a), (≡≡-refl, ≡≡-refl)) = ≡≡-refl


fun′-∘-⊆ : {k₁ k₂ k₃ i₁ i₂ i₃ : Level} {A : Set i₁} {B : Set i₂} {C : Set i₃}
           → (f : A → B) → (g : B → C)
           → fun′ {k₃} (g ∘ f) ⊆ fun′ {k₁} f ⨾ fun′ {k₂} g
fun′-∘-⊆ f g a ∘ (g (f a)) ≡≡-refl = (f a, (≡≡-refl, ≡≡-refl))


fun′-⨾-fun′ : {k₁ k₂ k₃ i₁ i₂ i₃ : Level} {A : Set i₁} {B : Set i₂} {C : Set i₃}
           → (f : A → B) → (g : B → C)
           → fun′ {k₁} f ⨾ fun′ {k₂} g ≑ fun′ {k₃} (g ∘ f)
fun′-⨾-fun′ f g = fun′-⨾-fun′-⊆ f g, fun′-∘-⊆ f g
```

18.23 Relation.Binary.Heterogeneous.Props.Star

The following are the properties required for Kleene star in Sect. 14.3.

```
Star-isLeftSupIdentity : {i k₁ : Level} {A : Set i} {R : 𝓡ℯ𝓁 k₁ A A}
                        → {j k₂ : Level} {B : Set j} {S : 𝓡ℯ𝓁 k₂ A B}
                        → S ⊆ Star R ⨾ S
Star-isLeftSupIdentity x y Sxy = (x, ε, Sxy)
Star-recDef⊆ : {i k : Level} {A : Set i} {R : 𝓡ℯ𝓁 k A A} → Star R ⊆ idR {k} ∪ R ∪ Star R ⨾ Star R
Star-recDef⊆ = ⊆-trans Star-isLeftSupIdentity (⊆-trans ∪-upper₂ ∪-upper₂)
Star-isCoreflexive : {i k : Level} {A : Set i} {R : 𝓡ℯ𝓁 k A A} → idR {k} ⊆ Star R
Star-isCoreflexive x .x ≡≡-refl = ε
Star-isIncreasing : {i k : Level} {A : Set i} {R : 𝓡ℯ𝓁 k A A} → R ⊆ Star R
Star-isIncreasing x y Rxy = Rxy ◁ ε
Star-recDef⊇ : {i k : Level} {A : Set i} {R : 𝓡ℯ𝓁 k A A} → idR {k} ∪ R ∪ Star R ⨾ Star R ⊆ Star R
Star-recDef⊇ = ∪-universal Star-isCoreflexive
    (∪-universal Star-isIncreasing
      (λ x z SRSRxz → proj₁ (proj₂ SRSRxz) ◁◁ proj₂ (proj₂ SRSRxz)))
Star-recDef : {i k : Level} {A : Set i} {R : 𝓡ℯ𝓁 k A A} → Star R ≑ (idR {k} ∪ R ∪ (Star R ⨾ Star R))
Star-recDef = ⊆-≑-antisym Star-recDef⊆ Star-recDef⊇
  -- Data.Star.fold composes only with homogeneous relations.
Star-leftInd : {i k₁ : Level} {A : Set i} {R : 𝓡ℯ𝓁 k₁ A A}
           → {j k₂ : Level} {B : Set j} {S : 𝓡ℯ𝓁 k₂ A B}
           → R ⨾ S ⊆ S → Star R ⨾ S ⊆ S
Star-leftInd R⨾S⊆S x z (.x, ε, Sxz) = Sxz
Star-leftInd R⨾S⊆S x z (y, _◁_ {.x} {x′} {.y} Rxx′ x′StarRy, Syz)
```

```
  = R⨾S⊆S x z (x', Rxx', Star-leftInd R⨾S⊆S x' z (y, x'StarRy, Syz))
Star-rightInd  : {i k₁ : Level} {A : Set i} {R : 𝓡el k₁ A A}
               → {j k₂ : Level} {B : Set j} {Q : 𝓡el k₂ B A}
               → Q ⨾ R ⊆ Q → Q ⨾ Star R ⊆ Q
Star-rightInd {A = A} {R} {Q = Q} Q⨾R⊆Q x z (y, Qxy, StarRxz) = h y Qxy StarRxz
  where
    h : (y : A) → Q x y → Star R y z → Q x z
    h .z Qxz ε = Qxz
    h y Qxy (_◁_ {.y} {y'} {.z} Ryy' y'StarRz)
       = h y' (Q⨾R⊆Q x y' (y, Qxy, Ryy')) y'StarRz


symStar-isEquivalence : {i k : Level} {A : Set i} (R : 𝓡el k A A) → symmetric R → IsEquivalence (Star R)
symStar-isEquivalence R R-sym = record
  {refl   = ε
  ; sym   = reverse (λ {i} {j} → R-sym j i)
  ; trans = _◁◁_
  }


equivClosure : {i k : Level} {A : Set i} (R : 𝓡el k A A) → 𝓡el (i ⊍ k) A A
equivClosure R = Star (R ∪ R ˘)
equivClosure-isEquivalence : {i k : Level} {A : Set i} (R : 𝓡el k A A) → IsEquivalence {i} {i ⊍ k} {A} (equivClosure R)
equivClosure-isEquivalence R = record
  {refl   = ε
  ; sym   = reverse [inj₂, inj₁]′
  ; trans = _◁◁_
  }
```

Chapter 19

Implementations of Categoric Interfaces by Concrete Relations

Concrete relations, as defined in Chapter 18, provide models for the theories of Part I and Part II; the proofs for this are now mostly straight-forward adaptations, especially since we limit ourselves to instances where morphisms come from $\mathcal{Rel}$ k for a single level k.

Up to OCCs (Sect. 19.14), we define all individual instances directly, in order to enable uses of, for example, the OSGC interface to concrete relations without having to load, for example, Categoric.Allegory.

The higher theories are instantiated in groups in sections 19.15 (allegories) to 19.18 (division allegories) — if a concrete need arises to split any of these modules, that will be done.

19.1 Relation.Binary.Heterogeneous.Categoric

Re-export only:

```
open import Relation.Binary.Heterogeneous.Categoric.Semigroupoid            public
open import Relation.Binary.Heterogeneous.Categoric.ConvSemigroupoid        public
open import Relation.Binary.Heterogeneous.Categoric.OrderedSemigroupoid     public
open import Relation.Binary.Heterogeneous.Categoric.OSGC                    public
open import Relation.Binary.Heterogeneous.Categoric.SemiAllegory            public
open import Relation.Binary.Heterogeneous.Categoric.SemiCollagory           public
open import Relation.Binary.Heterogeneous.Categoric.DistrSemiAllegory       public
open import Relation.Binary.Heterogeneous.Categoric.DivSemiAllegory         public
open import Relation.Binary.Heterogeneous.Categoric.KSGC                    public

open import Relation.Binary.Heterogeneous.Categoric.Category                public
open import Relation.Binary.Heterogeneous.Categoric.ConvCategory            public
open import Relation.Binary.Heterogeneous.Categoric.OrderedCategory         public
open import Relation.Binary.Heterogeneous.Categoric.OCC                     public
open import Relation.Binary.Heterogeneous.Categoric.Allegory                public
open import Relation.Binary.Heterogeneous.Categoric.Collagory               public
open import Relation.Binary.Heterogeneous.Categoric.DistrAllegory           public
open import Relation.Binary.Heterogeneous.Categoric.DivAllegory             public
open import Relation.Binary.Heterogeneous.Categoric.KCC                     public
```

19.2 Relation.Binary.Heterogeneous.Categoric.Semigroupoid

```
RelHom : (i j : Level) → LocalSetoid (Set i) (ℓsuc (i ⊍ j)) (i ⊍ j)
RelHom i j = ≑-Setoid (i ⊍ j)
```

```
RelCompOp : (i j : Level) → CompOp {ℓsuc i} {ℓsuc (i ⊍ j)} {i ⊍ j} (RelHom i j)
RelCompOp _ _ = record
  {_⨾_ = _⨾_
  ;⨾-cong = ⨾-cong
  ;⨾-assoc = ⨾-assoc
  }
RelSemigroupoid : (i j : Level) → Semigroupoid {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (Set i)
RelSemigroupoid i j = record
  {Hom = RelHom i j
  ;compOp = RelCompOp i j
  }
```

19.3 Relation.Binary.Heterogeneous.Categoric.ConvSemigroupoid

```
RelConvOp : (i j : Level) → ConvOp (RelSemigroupoid i j)
RelConvOp i j = record
  {_˘ = Rel._˘
  ;˘-cong = Rel.˘-cong
  ;˘˘ = λ {A} {B} {R} → Rel.˘˘-Id R
  ;˘-involution = λ {A} {B} {X} {R} {S} → Rel.˘-Involute R S
  }
RelConvSemigroupoid : (i j : Level) → ConvSemigroupoid {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (Set i)
RelConvSemigroupoid i j = record
  {semigroupoid = RelSemigroupoid i j
  ;convOp = RelConvOp i j
  }
```

```
IsEquivalence-SymIdempot : {i j : Level} {A : Set i} {_≈_ : 𝓡𝓮𝓵 (i ⊍ j) A A}
  → IsEquivalence _≈_ → ConvSemigroupoid.SymIdempot (RelConvSemigroupoid i j)
IsEquivalence-SymIdempot {i} {j} {A} {_≈_} IE = let open IsEquivalence IE in record
  {obj = A
  ;⟪_⟫ = _≈_
  ;prop = record
    {symmetric  = (λ x y → sym), (λ x y → sym)
    ;idempotent = (λ x y x≈≈y → trans (proj₁ (proj₂ x≈≈y)) (proj₂ (proj₂ x≈≈y)))
                ,  (λ x y x≈y → y, x≈y, refl)
    }
  }
```

```
Setoid-SymIdempot : {i j : Level} → Setoid i (i ⊍ j)
  → ConvSemigroupoid.SymIdempot (RelConvSemigroupoid i j)
Setoid-SymIdempot {i} {j} S = IsEquivalence-SymIdempot {i} {j} (Setoid.isEquivalence S)
```

19.4 Relation.Binary.Heterogeneous.Categoric.OrderedSemigroupoid

```
RelOrderedSemigroupoid : (i j : Level) → OrderedSemigroupoid {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelOrderedSemigroupoid i j = record
  {Hom = ⊆-Poset (i ⊍ j)
  ;compOp = RelCompOp i j
  ;locOrd = record {⨾-monotone = ⨾-monotone}
  }
```

19.5 Relation.Binary.Heterogeneous.Categoric.OSGC

```
RelOSGC : (i j : Level) → OSGC {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelOSGC i j = record
  {OSGC_Base = record
    {orderedSemigroupoid = RelOrderedSemigroupoid i j
    ; convOp = RelConvOp i j
    ; ˘-monotone = ˘-monotone
    }
  }
```

19.6 Relation.Binary.Heterogeneous.Categoric.SemiAllegory

```
RelMeetOp : (i j : Level) → MeetOp (RelOrderedSemigroupoid i j)
RelMeetOp i j = record
  {meet = λ {A} {B} R S → record
    {value = R ∩ S
    ; proof = record
      {bound₁ = ∩-lower₁ {R₁ = R} {R₂ = S}
      ; bound₂ = ∩-lower₂ {R₁ = R} {R₂ = S}
      ; universal = ∩-universal
      }
    }
  }

RelDomainOp : (i j : Level) → OSGDomainOp (RelOrderedSemigroupoid i j)
RelDomainOp i j = let open OrderedSemigroupoid (RelOrderedSemigroupoid i j) in record
  {dom = dom
  ; domSubIdentity = λ {A} {B} {R}
    → (λ {Z} {S} → coreflexive-IsLeftSubidentity (dom-coreflexive {A = A} {B} {R}))
    ,  (λ {Z} {Q} → coreflexive-IsRightSubidentity (dom-coreflexive {A = A} {B} {R}))
  ; dom-⨾-idempotent  = λ {A} {B} {R : Mor A B} → dom-⨾-Idempotent
  ; domPreserves⊑      = λ {A} {B} {Q R : Mor A B} → domPreserves⊆
  ; domLeastPreserver = λ {A} {B} {R : Mor A B} {d : Mor A A} isSubid-d d⨾d≈d R⊑d⨾R
                          → domLeastPreserver (leftSubidentity-coreflexive j (proj₁ isSubid-d)) R⊑d⨾R
  ; domLocality = λ {A} {B} {C} {R} {S} → domLocality {A = A} {B} {C} {R} {S}
  }

RelRangeOp : (i j : Level) → OSGRangeOp (RelOrderedSemigroupoid i j)
RelRangeOp i j = let open OrderedSemigroupoid (RelOrderedSemigroupoid i j) in record
  {ran = ran
  ; ranSubIdentity = λ {A} {B} {R}
    → (λ {Z} {S} → coreflexive-IsLeftSubidentity (ran-coreflexive {A = A} {B} {R}))
    ,  (λ {Z} {Q} → coreflexive-IsRightSubidentity (ran-coreflexive {A = A} {B} {R}))
  ; ran-⨾-idempotent  = λ {A} {B} {R : Mor A B} → ran-⨾-Idempotent
  ; ranPreserves⊑      = λ {A} {B} {Q R : Mor A B} → ranPreserves⊆
  ; ranLeastPreserver = λ {A} {B} {R : Mor A B} {d : Mor B B} isSubid-d d⨾d≈d R⊑R⨾d
                          → ranLeastPreserver (leftSubidentity-coreflexive j (proj₁ isSubid-d)) R⊑R⨾d
  ; ranLocality = λ {A} {B} {C} {R} {S} → ranLocality {A = A} {B} {C} {R} {S}
  }

RelOSGDR : (i j : Level) → OSGDR {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelOSGDR i j = record
  {orderedSemigroupoid = RelOrderedSemigroupoid i j
```

```
  ; domainOp = RelDomainOp i j
  ; rangeOp = RelRangeOp i j
  }


RelDedekind : {i j l k₁ k₂ k₃ : Level} {A : Set i} {B : Set j} {C : Set l}
  → {Q : 𝓡eℓ k₁ A B}
  → {R : 𝓡eℓ k₂ B C}
  → {S : 𝓡eℓ k₃ A C}
  → (Q ⨾ R ∩ S) ⊆ (Q ∩ S ⨾ R ˘) ⨾ (R ∩ Q ˘ ⨾ S)
RelDedekind x z ((y, Qxy, Ryz), Sxz) =
  (y, (Qxy, (z, Sxz, Ryz))
    , (Ryz, (x, Qxy, Sxz))
  )


RelSemiAllegory : (i j : Level) → SemiAllegory {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelSemiAllegory i j = record
  {osgc = RelOSGC i j
  ; meetOp = RelMeetOp i j
  ; domainOp = RelDomainOp i j
  ; Dedekind = RelDedekind
  }
```

19.7 Relation.Binary.Heterogeneous.Categoric.SemiCollagory

```
RelJoinOp : (i j : Level) → JoinOp (RelOrderedSemigroupoid i j)
RelJoinOp i j = record
  {join = λ {A} {B} R S → record
    {value = R ∪ S
    ; proof = record
      {bound₁ = ∪-upper₁ {R₁ = R} {R₂ = S}
      ; bound₂ = ∪-upper₂ {R₁ = R} {R₂ = S}
      ; universal = ∪-universal
      }
    }
  }


RelLatticeSemigroupoid : (i j : Level) → LatticeSemigroupoid {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelLatticeSemigroupoid i j = record
  {orderedSemigroupoid = RelOrderedSemigroupoid i j
  ; meetOp = RelMeetOp i j
  ; joinOp = RelJoinOp i j
  }


RelHomLatticeDistr : (i j : Level) → HomLatticeDistr (RelLatticeSemigroupoid i j)
RelHomLatticeDistr i j = record
  {⊓-⊔-subdistribR = λ x y Q-∩-R∪S → let Qxy = proj₁ Q-∩-R∪S
    in Data.Sum.map (λ Rxy → (Qxy, Rxy)) (λ Sxy → (Qxy, Sxy)) (proj₂ Q-∩-R∪S)
  }


RelJoinCompDistrL : (i j : Level) → JoinCompDistrL (RelJoinOp i j)
RelJoinCompDistrL i j = record
  {⨾-⊔-subdistribL = λ x z Q∪R-⨾-S → let
    y = proj₁ Q∪R-⨾-S
```

```
    QRxy  =  proj₁ (proj₂ Q∪R-⨾-S)
    Syz  =  proj₂ (proj₂ Q∪R-⨾-S)
    in Data.Sum.map (λ Qxy → (y, Qxy, Syz)) (λ Rxy → (y, Rxy, Syz)) QRxy
  }


RelJoinCompDistrR : (i j : Level) → JoinCompDistrR (RelJoinOp i j)
RelJoinCompDistrR i j  =  record
  {⨾-⊔-subdistribR  =  λ x z Q-⨾-R∪S → let
    y  =  proj₁ Q-⨾-R∪S
    Qxy  =  proj₁ (proj₂ Q-⨾-R∪S)
    RSyz  =  proj₂ (proj₂ Q-⨾-R∪S)
    in Data.Sum.map (λ Ryz → (y, Qxy, Ryz)) (λ Syz → (y, Qxy, Syz)) RSyz
  }


RelUSLSemigroupoid : (i j : Level) → USLSemigroupoid {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelUSLSemigroupoid i j  =  record
  {orderedSemigroupoid  =  RelOrderedSemigroupoid i j
  ; joinOp               =  RelJoinOp i j
  ; joinCompDistrL       =  RelJoinCompDistrL i j
  ; joinCompDistrR       =  RelJoinCompDistrR i j
  }


RelUSLSGC : (i j : Level) → USLSGC {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelUSLSGC i j  =  record
  {osgc            =  RelOSGC i j
  ; joinOp         =  RelJoinOp i j
  ; joinCompDistrL  =  RelJoinCompDistrL i j
  ; joinCompDistrR  =  RelJoinCompDistrR i j
  }


RelSemiCollagory : (i j : Level) → SemiCollagory {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelSemiCollagory i j  =  record
  {semiAllegory    =  RelSemiAllegory      i j
  ; joinOp         =  RelJoinOp            i j
  ; homLatDistr    =  RelHomLatticeDistr i j
  ; joinCompDistrL  =  RelJoinCompDistrL  i j
  ; joinCompDistrR  =  RelJoinCompDistrR i j
  }
```

19.8 Relation.Binary.Heterogeneous.Categoric.DistrSemiAllegory

We use Data.Empty.Generalised for level-polymorphic empty sets.

```
RelBotMor : (i j : Level) → BotMor (RelOrderedSemigroupoid i j)
RelBotMor i j  =  record
  {leastMor  =  λ {A} {B} → record
    {mor  =  λ _ _ → ⊥
    ; proof  =  λ R x y ()
    }
  }


RelLeftZeroLaw : (i j : Level) → LeftZeroLaw (RelBotMor i j)
RelLeftZeroLaw i j  =  record {leftZero⊑  =  λ x z x⊥yRz → ⊥-elim (proj₁ (proj₂ x⊥yRz))}
```

```
RelRightZeroLaw : (i j : Level) → RightZeroLaw (RelBotMor i j)
RelRightZeroLaw i j = record {rightZero⊑ = λ x z xRy⊥z → ⊥-elim (proj₂ (proj₂ xRy⊥z))}

RelZeroMor : (i j : Level) → ZeroMor (RelOrderedSemigroupoid i j)
RelZeroMor i j = record
  {botMor        = RelBotMor       i j
  ; leftZeroLaw  = RelLeftZeroLaw  i j
  ; rightZeroLaw = RelRightZeroLaw i j
  }

RelDistrSemiAllegory : (i j : Level) → DistrSemiAllegory {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelDistrSemiAllegory i j = record
  {semiCollagory = RelSemiCollagory i j
  ; zeroMor      = RelZeroMor       i j
  }
```

19.9 Relation.Binary.Heterogeneous.Categoric.DivSemiAllegory

```
RelLeftResOp : (i j : Level) → LeftResOp (RelOrderedSemigroupoid i j)
RelLeftResOp i j = record
  {_/_ = _/_
  ; /-cancel-outer = λ {A} {B} {C} {S} {R} → /-cancel-outer {A = A} {B} {C} {S} {R}
  ; /-universal = /-universal
  }

RelRightResOp : (i j : Level) → RightResOp (RelOrderedSemigroupoid i j)
RelRightResOp i j = record
  {_\_ = _\_
  ; \-cancel-outer = λ {A} {B} {C} {S} {Q} → \-cancel-outer {A = A} {B} {C} {S} {Q}
  ; \-universal = \-universal
  }

RelSyqOp : (i j : Level) → SyqOp (RelOSGC i j)
RelSyqOp i j = SyqOp-from-ResOps.syqOp (RelSemiAllegory i j) (RelLeftResOp i j) (RelRightResOp i j)

RelLeftRestrResOp : (i j : Level) → LeftRestrResOp (RelOSGDR i j)
RelLeftRestrResOp i j = record
  {_/•_ = _/•_
  ; /•-cancel-outer = λ {A} {B} {C} {S} {R} → /•-cancel-outer {A = A} {B} {C} {S} {R}
  ; /•-restr = λ {A} {B} {C} {S} {R} → /•-restr {A = A} {B} {C} {S} {R}
  ; /•-universal = /•-universal
  }
RelRightRestrResOp : (i j : Level) → RightRestrResOp (RelOSGDR i j)
RelRightRestrResOp i j = record
  {_\•_ = _\•_
  ; \•-cancel-outer = λ {A} {B} {C} {S} {Q} → \•-cancel-outer {A = A} {B} {C} {S} {Q}
  ; \•-restr = λ {A} {B} {C} {S} {Q} → \•-restr {A = A} {B} {C} {S} {Q}
  ; \•-universal = \•-universal
  }

RelDivSemiAllegory : (i j : Level) → DivSemiAllegory {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelDivSemiAllegory i j = record
  {distrSemiAllegory = RelDistrSemiAllegory i j
  ; leftResOp        = RelLeftResOp         i j
  ; rightResOp       = RelRightResOp        i j
  ; syqOp            = RelSyqOp             i j
  }
```

19.10 Relation.Binary.Heterogeneous.Categoric.KSGC

```
RelTransClosOp : (i j : Level) → TransClosOp (RelUSLSemigroupoid i j)
RelTransClosOp i j = record
  {_⁺ = λ {A} R → Plus R
  ;⁺-recDef₁ = λ {A} {R} → Plus-recDef₁ {R = R}
  ;⁺-recDef₂ = λ {A} {R} → Plus-recDef₂ {R = R}
  ;⁺-leftInd = λ {A} {R} {B} {S} → Plus-leftInd {R = R} {S = S}
  ;⁺-rightInd = λ {A} {R} {B} {Q} → Plus-rightInd {R = R} {Q = Q}
  }

RelKleeneSemigroupoid : (i j : Level) → KleeneSemigroupoid {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelKleeneSemigroupoid i j = record
  {uslSemigroupoid = RelUSLSemigroupoid i j
  ;transClosOp     = RelTransClosOp     i j
  }

RelKSGC : (i j : Level) → KSGC {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelKSGC i j = record
  {uslsgc      = RelUSLSGC      i j
  ;transClosOp = RelTransClosOp i j
  }
```

19.11 Relation.Binary.Heterogeneous.Categoric.Category

```
RelIdOp : (i j : Level) → IdOp (RelHom i j) _⨾_
RelIdOp i j = record
  {Id = λ {A} → idR {A = A}
  ;leftId = leftId
  ;rightId = rightId
  }
RelCategory : (i j : Level) → Category {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (Set i)
RelCategory i j = record
  {semigroupoid = RelSemigroupoid i j
  ;idOp = RelIdOp i j
  }
```

19.12 Relation.Binary.Heterogeneous.Categoric.ConvCategory

```
RelConvCategory : (i j : Level) → ConvCategory {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (Set i)
RelConvCategory i j = record
  {convSemigroupoid = RelConvSemigroupoid i j
  ;idOp = RelIdOp i j
  }
```

19.13 Relation.Binary.Heterogeneous.Categoric.OrderedCategory

```
RelOrderedCategory : (i j : Level) → OrderedCategory {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelOrderedCategory i j = record
  {orderedSemigroupoid = RelOrderedSemigroupoid i j
  ;idOp = RelIdOp i j
  }
```

19.14 Relation.Binary.Heterogeneous.Categoric.OCC

```
RelOCC : (i j : Level) → OCC {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelOCC i j = record {OCC_Base = record
  {osgc = RelOSGC i j
  ; idOp = RelIdOp i j
  }}
mapping : {k i : Level} {A : Set i} {B : Set i} → (A → B) → OCC.Mapping (RelOCC i k) A B
mapping {k} {i} {A} {B} f = let open OCC (RelOCC i k) using (isMapping-from-I) in record
  {mor = fun′ f
  ; prf  = isMapping-from-I (fun′-Univalent f, fun′-Total f)
  }
```

19.15 Relation.Binary.Heterogeneous.Categoric.Allegory

```
RelAllegory : (i j : Level) → Allegory {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelAllegory i j = record
  {occ = RelOCC i j
  ; meetOp = RelMeetOp i j
  ; Dedekind = RelDedekind
  }
```

19.16 Relation.Binary.Heterogeneous.Categoric.Collagory

```
RelUSLCategory : (i j : Level) → USLCategory {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelUSLCategory i j = record
  {orderedCategory = RelOrderedCategory i j
  ; joinOp           = RelJoinOp i j
  ; joinCompDistrL   = RelJoinCompDistrL i j
  ; joinCompDistrR   = RelJoinCompDistrR i j
  }
```

```
RelUSLCC : (i j : Level) → USLCC {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelUSLCC i j = record
  {occ              = RelOCC i j
  ; joinOp          = RelJoinOp i j
  ; joinCompDistrL  = RelJoinCompDistrL i j
  ; joinCompDistrR  = RelJoinCompDistrR i j
  }
```

```
RelCollagory : (i j : Level) → Collagory {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelCollagory i j = record
  {allegory         = RelAllegory          i j
  ; joinOp          = RelJoinOp            i j
  ; homLatDistr     = RelHomLatticeDistr i j
  ; joinCompDistrL  = RelJoinCompDistrL  i j
  ; joinCompDistrR  = RelJoinCompDistrR  i j
  }
```

19.17 Relation.Binary.Heterogeneous.Categoric.DistrAllegory

```
RelDistrAllegory : (i j : Level) → DistrAllegory {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelDistrAllegory i j = record
  {collagory = RelCollagory i j
  ; zeroMor  = RelZeroMor  i j
  }
```

19.18 Relation.Binary.Heterogeneous.Categoric.DivAllegory

```
RelDivAllegory : (i j : Level) → DivAllegory {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelDivAllegory i j = record
  {distrAllegory = RelDistrAllegory i j
  ; leftResOp    = RelLeftResOp    i j
  ; rightResOp   = RelRightResOp i j
  ; syqOp        = RelSyqOp        i j
  }
```

19.19 Relation.Binary.Heterogeneous.Categoric.KCC

```
RelStarOp : (i j : Level) → StarOp (RelUSLCategory i j)
RelStarOp i j = record
  {_* = λ {A} R → Star R
  ; isStar = λ R → record
    {*-recDef   = Star-recDef {R = R}
    ; *-leftInd  = λ {B} {S} → Star-leftInd {R = R} {S = S}
    ; *-rightInd = λ {B} {Q} → Star-rightInd {R = R} {Q = Q}
    }
  }
```

```
RelKleeneCategory : (i j : Level) → KleeneCategory {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelKleeneCategory i j = record
  {uslCategory = RelUSLCategory i j
  ; zeroMor    = RelZeroMor     i j
  ; starOp     = RelStarOp      i j
  }
```

```
RelKCC : (i j : Level) → KCC {ℓsuc i} (ℓsuc (i ⊍ j)) (i ⊍ j) (i ⊍ j) (Set i)
RelKCC i j = record
  {uslcc    = RelUSLCC   i j
  ; zeroMor = RelZeroMor i j
  ; starOp  = RelStarOp  i j
  }
```

Chapter 20

Implementations of Categoric Interfaces by Setoid Homomorphisms

Setoids, with homomorphisms as defined in the standard library module Function.Equality, form a category.

20.1 Data.Sum.Setoid

```
_⊎⊎_ : {i₁ i₂ k₁ k₂ : Level} → Setoid i₁ k₁ → Setoid i₂ k₂ → Setoid (i₁ ⊍ i₂) (i₁ ⊍ i₂ ⊍ k₁ ⊍ k₂)
A ⊎⊎ B = A ⊎-setoid B
```

```
Inj₁ : {a b ℓ₁ ℓ₂ : Level} (A : Setoid a ℓ₁) (B : Setoid b ℓ₂) → A ⟶ A ⊎⊎ B
Inj₁ A B = record
  {_⟨$⟩_ = inj₁
  ; cong   = λ {i} {j} p → ₁∼₁ p
  }
Inj₂ : {a b ℓ₁ ℓ₂ : Level} (A : Setoid a ℓ₁) (B : Setoid b ℓ₂) → B ⟶ A ⊎⊎ B
Inj₂ A B = record
  {_⟨$⟩_ = inj₂
  ; cong   = λ {i} {j} p → ₂∼₂ p
  }
```

A "missing" helper function from Relation.Binary.Sum:

```
≈⊎-combine : {a b c ℓ₁ ℓ₂ : Level} {A : Setoid a ℓ₁} {B : Setoid b ℓ₂}
           → {R : {x y : ⌊ A ⊎⊎ B ⌋} → x ≈⌊ A ⊎⊎ B ⌋ y → Set c}
           → ({a₁ a₂ : ⌊ A ⌋} (a₁≈a₂ : a₁ ≈⌊ A ⌋ a₂) → R (₁∼₁ a₁≈a₂))
           → ({b₁ b₂ : ⌊ B ⌋} (b₁≈b₂ : b₁ ≈⌊ B ⌋ b₂) → R (₂∼₂ b₁≈b₂))
           → ({x y : ⌊ A ⊎⊎ B ⌋} → (x≈y : x ≈⌊ A ⊎⊎ B ⌋ y) → R x≈y)
≈⊎-combine P Q (₁∼₁ a₁≈a₂) = P a₁≈a₂
≈⊎-combine P Q (₂∼₂ b₁≈b₂) = Q b₁≈b₂
≈⊎-combine P Q (₁∼₂ ())
```

```
⟨[_,_]⟩ : {a b c ℓ₁ ℓ₂ ℓ₃ : Level} {A : Setoid a ℓ₁} {B : Setoid b ℓ₂} {C : Setoid c ℓ₃} →
  (A ⟶ C) → (B ⟶ C) → (A ⊎⊎ B ⟶ C)
⟨[_,_]⟩ {A = A} {B = B} {C = C} F G = record
  {_⟨$⟩_ = FG
  ; cong   = λ {i} {j} p → cng {i} {j} p
  }
  where
    FG : Setoid.Carrier (A ⊎⊎ B) → Setoid.Carrier C
```

```
    FG = [ _⟨$⟩_ F, _⟨$⟩_ G ]′
    cng : {x y : ⌊ A ⌋ ⊎ ⌊ B ⌋} → x ≈⌊ A ⊎-setoid B ⌋ y → FG x ≈⌊ C ⌋ FG y
    cng (₁∼₁ a₁≈a₂) = cong F a₁≈a₂
    cng (₂∼₂ b₁≈b₂) = cong G b₁≈b₂
    cng (₁∼₂ ())

⟨[,]⟩-cong :    {a b c ℓ₁ ℓ₂ ℓ₃ : Level} {A : Setoid a ℓ₁} {B : Setoid b ℓ₂} {C : Setoid c ℓ₃}
             → {F₁ F₂ : A ⟶ C} →    F₁ ≈⌊ A ⇨ C ⌋ F₂
               → {G₁ G₂ : B ⟶ C} → G₁ ≈⌊ B ⇨ C ⌋ G₂
             → ⟨[ F₁, G₁ ]⟩ ≈⌊ A ⊎⊎ B ⇨ C ⌋ ⟨[ F₂, G₂ ]⟩
⟨[,]⟩-cong F₁≈F₂ G₁≈G₂ (₁∼₂ ())
⟨[,]⟩-cong F₁≈F₂ G₁≈G₂ (₁∼₁ x∼₁y) = F₁≈F₂ x∼₁y
⟨[,]⟩-cong F₁≈F₂ G₁≈G₂ (₂∼₂ x∼₂y) = G₁≈G₂ x∼₂y

∘-⟨[,]⟩ : {a b c d ℓ₁ ℓ₂ ℓ₃ ℓ₄ : Level}
          → {A : Setoid a ℓ₁} {B : Setoid b ℓ₂} {C : Setoid c ℓ₃} {D : Setoid d ℓ₄}
          → {F : A ⟶ C} {G : B ⟶ C} {H : C ⟶ D}
          → H ∘ ⟨[ F, G ]⟩ ≈⌊ A ⊎⊎ B ⇨ D ⌋ ⟨[ H ∘ F, H ∘ G ]⟩
∘-⟨[,]⟩ (₁∼₂ ())
∘-⟨[,]⟩ {F = F} {G} {H} (₁∼₁ x∼₁y) = cong H (cong F x∼₁y)
∘-⟨[,]⟩ {F = F} {G} {H} (₂∼₂ x∼₂y) = cong H (cong G x∼₂y)

⊎-factors : {a b c ℓ₁ ℓ₂ ℓ₃ : Level} {A : Setoid a ℓ₁} {B : Setoid b ℓ₂} {C : Setoid c ℓ₃}
          → (R : A ⟶ C) → (S : B ⟶ C)
          → (U : A ⊎⊎ B ⟶ C)
          → Set (a ⊍ b ⊍ ℓ₁ ⊍ ℓ₂ ⊍ ℓ₃)
⊎-factors {A = A} {B} {C} R S U = (U ∘ Inj₁ A B ≈⌊ A ⇨ C ⌋ R) × (U ∘ Inj₂ A B ≈⌊ B ⇨ C ⌋ S)
⊎-factors₀ : {a b c ℓ₁ ℓ₂ ℓ₃ : Level} {A : Setoid a ℓ₁} {B : Setoid b ℓ₂} {C : Setoid c ℓ₃}
          → (R : A ⟶ C) → (S : B ⟶ C)
          → ⊎-factors R S ⟨[ R, S ]⟩
⊎-factors₀ R S = (λ x≈y → cong R x≈y), (λ x≈y → cong S x≈y)

⊎-factors-unique : {a b c ℓ₁ ℓ₂ ℓ₃ : Level} {A : Setoid a ℓ₁} {B : Setoid b ℓ₂} {C : Setoid c ℓ₃}
  → (R : A ⟶ C) → (S : B ⟶ C)
  → (V : A ⊎⊎ B ⟶ C)
  → ⊎-factors R S V
  → ⟨[ R, S ]⟩ ≈⌊ A ⊎⊎ B ⇨ C ⌋ V
⊎-factors-unique {A = A} {B} {C} R S V RS≈ικV {x} {y} x≈y = ≈⊎-combine {A = A} {B = B}
  {R = λ {x} {y} x≈y → [ _⟨$⟩_ R, _⟨$⟩_ S ] x ≈⌊ C ⌋ V ⟨$⟩ y}
  (λ {a₁} {a₂} a₁≈a₂ → Setoid.sym C ((proj₁ RS≈ικV (Setoid.sym A a₁≈a₂))))
  (λ {b₁} {b₂} b₁≈b₂ → Setoid.sym C ((proj₂ RS≈ικV (Setoid.sym B b₁≈b₂))))
  x≈y

_⊕_ : {ℓA ℓa : Level} {A : Setoid ℓA ℓa}
      → {ℓC ℓc : Level} {C : Setoid ℓC ℓc}
      → (A ⟶ C)
      → {ℓB ℓb : Level} {B : Setoid ℓB ℓb}
      → {ℓD ℓd : Level} {D : Setoid ℓD ℓd}
      → (B ⟶ D)
      → (A ⊎⊎ B ⟶ C ⊎⊎ D)
_⊕_ {C = C} F {D = D} G = ⟨[ Inj₁ C D ∘ F, Inj₂ C D ∘ G ]⟩

id⊕id : {a b ℓ : Level} {St₁ : Setoid a ℓ} {St₂ : Setoid b ℓ} {s : ⌊ St₁ ⌋ ⊎ ⌊ St₂ ⌋}
  → (id {A = St₁} ⊕ id {A = St₂}) ⟨$⟩ s ≡ s
id⊕id {a} {b} {ℓ} {St₁} {St₂} {inj₁ x} = ≡-refl
```

```
id⊕id {a} {b} {ℓ} {St₁} {St₂} {inj₂ y} = ≡-refl
id⊕id-≈ : {a b ℓ : Level} {A : Setoid a ℓ} {B : Setoid b ℓ}
  → (id {A = A} ⊕ id {A = B}) ≈⌊ A ⊎⊎ B ⇨ A ⊎⊎ B ⌋ id
id⊕id-≈ {a} {b} {ℓ} {A} {B} (₁∼₂ ())
id⊕id-≈ {a} {b} {ℓ} {A} {B} (₁∼₁ x∼₁y) = ₁∼₁ x∼₁y
id⊕id-≈ {a} {b} {ℓ} {A} {B} (₂∼₂ x∼₂y) = ₂∼₂ x∼₂y


⊕-∘-Inj₁ : {ℓA ℓa : Level} {A : Setoid ℓA ℓa}
  → {ℓC ℓc : Level} {C : Setoid ℓC ℓc}
  → (F : A ⟶ C)
  → {ℓB ℓb : Level} {B : Setoid ℓB ℓb}
  → {ℓD ℓd : Level} {D : Setoid ℓD ℓd}
  → (G : B ⟶ D)
  → (F ⊕ G) ∘ Inj₁ A B ≈⌊ A ⇨ C ⊎⊎ D ⌋ Inj₁ C D ∘ F
⊕-∘-Inj₁ F G x≈y = ₁∼₁ (cong F x≈y)


⊕-∘-Inj₂ : {ℓA ℓa : Level} {A : Setoid ℓA ℓa}
  → {ℓC ℓc : Level} {C : Setoid ℓC ℓc}
  → (F : A ⟶ C)
  → {ℓB ℓb : Level} {B : Setoid ℓB ℓb}
  → {ℓD ℓd : Level} {D : Setoid ℓD ℓd}
  → (G : B ⟶ D)
  → (F ⊕ G) ∘ Inj₂ A B ≈⌊ B ⇨ C ⊎⊎ D ⌋ Inj₂ C D ∘ G
⊕-∘-Inj₂ F G x≈y = ₂∼₂ (cong G x≈y)


⊕-∘ : {a b ℓ : Level} {A I J : Setoid a ℓ} {B C D : Setoid b ℓ} → {s : ⌊ A ⌋ ⊎ ⌊ B ⌋}
  (F : I ⟶ J) → (H : A ⟶ I) → (G : C ⟶ D) → (K : B ⟶ C) →
  ((F ⊕ G) ∘ (H ⊕ K)) ⟨$⟩ s ≡ ((F ∘ H) ⊕ (G ∘ K)) ⟨$⟩ s
⊕-∘ {s = inj₁ x} F H G K = ≡-refl
⊕-∘ {s = inj₂ y} F H G K = ≡-refl
⊕-∘-≈ : {a b ℓ : Level} {A I J : Setoid a ℓ} {B C D : Setoid b ℓ}
  (F : I ⟶ J) → (H : A ⟶ I) → (G : C ⟶ D) → (K : B ⟶ C) →
  (F ⊕ G) ∘ (H ⊕ K) ≈⌊ A ⊎⊎ B ⇨ J ⊎⊎ D ⌋ (F ∘ H) ⊕ (G ∘ K)
⊕-∘-≈ F H G K (₁∼₂ ())
⊕-∘-≈ F H G K (₁∼₁ x∼₁y) = ₁∼₁ (cong F (cong H x∼₁y))
⊕-∘-≈ F H G K (₂∼₂ x∼₂y) = ₂∼₂ (cong G (cong K x∼₂y))
⊕-cong₁ : {ℓA ℓa : Level} {A : Setoid ℓA ℓa}
  →      {ℓC ℓc : Level} {C : Setoid ℓC ℓc}
  →      {f g : A ⟶ C}
  →      {ℓB ℓb : Level} {B : Setoid ℓB ℓb}
  →      {ℓD ℓd : Level} {D : Setoid ℓD ℓd}
  →      {h : B ⟶ D}
  →      f ≈⌊ A ⇨ C ⌋ g
  →      (f ⊕ h) ≈⌊ A ⊎⊎ B ⇨ C ⊎⊎ D ⌋ (g ⊕ h)
⊕-cong₁ {A = A} {C = C} {f} {g} {B = B} {D = D} {h} f≈g (₁∼₂ ())
⊕-cong₁ {A = A} {C = C} {f} {g} {B = B} {D = D} {h} f≈g (₁∼₁ x∼₁y) = ₁∼₁ (f≈g x∼₁y)
⊕-cong₁ {A = A} {C = C} {f} {g} {B = B} {D = D} {h} f≈g (₂∼₂ x∼₂y) = ₂∼₂ (cong h x∼₂y)


⟨[,]⟩∘⊕ : {ℓA₁ ℓa₁ : Level} {A₁ : Setoid ℓA₁ ℓa₁}
         → {ℓA₂ ℓa₂ : Level} {A₂ : Setoid ℓA₂ ℓa₂}
         → {ℓB₁ ℓb₁ : Level} {B₁ : Setoid ℓB₁ ℓb₁}
         → {ℓB₂ ℓb₂ : Level} {B₂ : Setoid ℓB₂ ℓb₂}
         → {ℓC ℓc : Level}   {C : Setoid ℓC ℓc}
         → {H : A₁ ⟶ A₂} {K : B₁ ⟶ B₂} {F : A₂ ⟶ C} {G : B₂ ⟶ C}
         → ⟨[ F , G ]⟩ ∘ (H ⊕ K) ≈⌊ A₁ ⊎⊎ B₁ ⇨ C ⌋ ⟨[ F ∘ H , G ∘ K ]⟩
⟨[,]⟩∘⊕ (₁∼₂ ())
```

```
⟨[,]⟩∘⊕ {H = H} {K} {F} {G} (₁∼₁ x∼₁y) = cong F (cong H x∼₁y)
⟨[,]⟩∘⊕ {H = H} {K} {F} {G} (₂∼₂ x∼₂y) = cong G (cong K x∼₂y)

_⊕⊕_ : {ℓA ℓa : Level} {A : Setoid ℓA ℓa}
      → {ℓC ℓc : Level} {C : Setoid ℓC ℓc}
      → Inverse A C
      → {ℓB ℓb : Level} {B : Setoid ℓB ℓb}
      → {ℓD ℓd : Level} {D : Setoid ℓD ℓd}
      → Inverse B D
      → Inverse (A ⊎⊎ B) (C ⊎⊎ D)
_⊕⊕_ {A = A} {C = C} F {B = B} {D = D} G = record
  {to    = to
  ; from = from
  ; inverse-of = record
    {left-inverse-of = left
    ; right-inverse-of = right
    }
  }
  where
    to : (A ⊎⊎ B) ⟶ (C ⊎⊎ D)
    to = Inverse.to F ⊕ Inverse.to G
    from : (C ⊎⊎ D) ⟶ (A ⊎⊎ B)
    from = Inverse.from F ⊕ Inverse.from G
    Fi = Inverse.inverse-of F
    Gi = Inverse.inverse-of G
    left : (x : ⌊ A ⊎⊎ B ⌋) → from ⟨$⟩ (to ⟨$⟩ x) ≈⌊ A ⊎⊎ B ⌋ x
    left (inj₁ x) = ₁∼₁ (Inverse.left-inverse-of F x)
    left (inj₂ y) = ₂∼₂ (Inverse.left-inverse-of G y)
    right : (x : ⌊ C ⊎⊎ D ⌋) → to ⟨$⟩ (from ⟨$⟩ x) ≈⌊ C ⊎⊎ D ⌋ x
    right (inj₁ x) = ₁∼₁ (Inverse.right-inverse-of F x)
    right (inj₂ y) = ₂∼₂ (Inverse.right-inverse-of G y)
```

20.2 Relation.Binary.Setoid.Product

```
infixr 2 _××_
_××_ : ∀ {c₁ ℓ₁ c₂ ℓ₂} → (S₁ : Setoid c₁ ℓ₁) (S₂ : Setoid c₂ ℓ₂) → Setoid _ _
S₁ ×× S₂ = S₁ ×-setoid S₂
Proj₁ : {a b ℓ₁ ℓ₂ : Level} (A : Setoid a ℓ₁) (B : Setoid b ℓ₂) → A ×× B ⟶ A
Proj₁ A B = record
  {_⟨$⟩_ = proj₁
  ; cong = proj₁}
Proj₂ : {a b ℓ₁ ℓ₂ : Level} (A : Setoid a ℓ₁) (B : Setoid b ℓ₂) → A ×× B ⟶ B
Proj₂ A B = record
  {_⟨$⟩_ = proj₂
  ; cong = proj₂}
××-idem : ∀ {a ℓ₁ : Level} (A : Setoid a ℓ₁) → A ⟷ (A ×× A)
××-idem A = record
  {to = record
    {_⟨$⟩_ = λ x → x, x
    ; cong = λ x → x, x}
  ; from = record
    {_⟨$⟩_ = λ x → proj₁ x
    ; cong = λ x → proj₁ x}
  }
```

20.3 Relation.Binary.Setoid.Coequaliser

Coequalisers of Setoid homomorphisms are quotients of an equivelence closure; we provide such quotients first:

```
Quotient : {i k₁ k₂ : Level} (S : Setoid i k₁)
         → {_≈′_ : Rel ⌊ S ⌋ k₂} → IsEquivalence _≈′_
         → (Setoid._≈_ S ⇒ _≈′_) → Σ [S′ : Setoid i k₂] (S ⟶ S′)
Quotient S {_≈′_} isEq incl = record {Carrier = ⌊ S ⌋; _≈_ = _≈′_; isEquivalence = isEq}
                            , record {_⟨$⟩_ = λ x → x; cong = incl}
```

Given two setoid homomorphisms, their coequalisers is a quotient by the equivalence closure of the following relation:

```
data Ccomp {i₁ i₂ k₂ : Level} {A : Set i₁}
           (B : Setoid i₂ k₂) (F G : A → ⌊ B ⌋) : Rel ⌊ B ⌋ (i₁ ⊍ i₂ ⊍ k₂) where
  Cc : (x : A) → Ccomp B F G (F x) (G x)
  cC : (x : A) → Ccomp B F G (G x) (F x)
  EQ : {x y : ⌊ B ⌋} → x ≈⌊ B ⌋ y → Ccomp B F G x y
```

The relation Ccomp B F G is symmetric by construction:

```
isSym-Ccomp : {i₁ i₂ k₂ : Level} {A : Set i₁}
              (B : Setoid i₂ k₂) (F G : A → ⌊ B ⌋) → symmetric (Ccomp B F G)
isSym-Ccomp B F G ∘ (F x) ∘ (G x) (cC x) = Cc x
isSym-Ccomp B F G ∘ (G x) ∘ (F x) (Cc x) = cC x
isSym-Ccomp B F G u       v       (EQ u≈v) = EQ (Setoid.sym B u≈v)
```

This allows us to define the quotient using the equivalence closure obtained via symStar-isEquivalence:

```
coequaliser : {i₁ i₂ k₁ k₂ : Level} {S : Setoid i₁ k₁} {T : Setoid i₂ k₂}
            → (F G : S ⟶ T) → Σ [T′ : Setoid i₂ (i₁ ⊍ i₂ ⊍ k₂)] (T ⟶ T′)
coequaliser {T = T} F G = Quotient T
  (symStar-isEquivalence (Ccomp T (_⟨$⟩_ F) (_⟨$⟩_ G))
                         (isSym-Ccomp T (_⟨$⟩_ F) (_⟨$⟩_ G)))
  (λ {x} {y} → ⊆-trans (λ x y → EQ) Star-isIncreasing x y)
```

It remains to show that this actually produces a coequaliser — it coequalises:

```
coequaliser-prop : {i₁ i₂ k₁ k₂ : Level} {S : Setoid i₁ k₁} {T : Setoid i₂ k₂}
                 → (F G : S ⟶ T)
                 → let Qq = coequaliser F G
                       Q = proj₁ Qq
                       C = proj₂ Qq
                   in Setoid._≈_ (S ⇨ Q) (C ∘ F) (C ∘ G)
coequaliser-prop {T = T} F G {x} {y} x≈y = Cc x ◁′ EQ (cong G x≈y) ◁′ ε
  where
    -- Using _◁_ instead of _◁′_ leaves metavariables uninstantiated.
    Ξ = Ccomp T (_⟨$⟩_ F) (_⟨$⟩_ G)
    infixr 5 _◁′_
    _◁′_ : ∀ {i j k} (x : Ξ i j) (xs : Star Ξ j k) → Star Ξ i k
    _◁′_ = _◁_ {T = Ξ}
```

For universality, we moved into a local module where we assume a coequaliser candidate to be given:

```
private
  module CoequaliserCandidate
    {i₁ i₂ i₃ k₁ k₂ k₃ : Level} {S : Setoid i₁ k₁} {T : Setoid i₂ k₂}
    (F G : S ⟶ T)
```

```
    {R : Setoid i₃ k₃}
    (H : T ⟶ R)
    (H∘F≈H∘G : Setoid._≈_ (S ⇨ R) (H ∘ F) (H ∘ G))
    where
    coequaliser-cong : Ccomp T (_⟨$⟩_ F) (_⟨$⟩_ G) ⇒ (Setoid._≈_ R on (_⟨$⟩_ H))
    coequaliser-cong {∘ (F ⟨$⟩ s)} {∘ (G ⟨$⟩ s)}    (Cc s)     = H∘F≈H∘G (Setoid.refl S {s})
    coequaliser-cong {∘ (G ⟨$⟩ s)} {∘ (F ⟨$⟩ s)}    (cC s)     = Setoid.sym R (H∘F≈H∘G (Setoid.refl S {s}))
    coequaliser-cong {x}              {y}              (EQ x≈y)  = cong H x≈y
    Qq = coequaliser F G
    Q = proj₁ Qq
    C = proj₂ Qq
    Ξ = Ccomp T (_⟨$⟩_ F) (_⟨$⟩_ G)
    U-cong : {x y : ⌊ T ⌋} (x≋y : Star Ξ x y) → H ⟨$⟩ x ≈⌊ R ⌋ H ⟨$⟩ y
    U-cong {x} {y} x≋y = Star-leftInd {R = Ξ} {S = funRel H}
      (λ x z xCHz → let x′ = proj₁ xCHz
                         xCx′ = proj₁ (proj₂ xCHz)
                         x′Hz = proj₂ (proj₂ xCHz)
                     in Setoid.trans R (coequaliser-cong xCx′) x′Hz)
      x (H ⟨$⟩ y) (y, x≋y, Setoid.refl R)
    U : Q ⟶ R
    U = record {_⟨$⟩_ = _⟨$⟩_ H; cong = U-cong}
    coequaliser-factoring : Σ [U : Q ⟶ R] (H    ≈⌊ T ⇨ R ⌋ U ∘ C)
    coequaliser-factoring = U, (λ {x} {y} → cong H)
    coequaliser-factor-unique : (V : Q ⟶ R) → H ≈⌊ T ⇨ R ⌋ V ∘ C → U ≈⌊ Q ⇨ R ⌋ V
    coequaliser-factor-unique V H≈V∘C {x} {y} = λ (x≋y : Star Ξ x y) →    -- : H ⟨$⟩ x ≈⌊ R ⌋ V ⟨$⟩ y
      Setoid.trans R (U-cong x≋y) (H≈V∘C (Setoid.refl T))
    coequaliser-universal : ∃! (Setoid._≈_ (Q ⇨ R)) λ (U : Q ⟶ R) → H ≈⌊ T ⇨ R ⌋ U ∘ C
    coequaliser-universal = U, (λ {x} {y} → cong H)
                                ,λ {V} → coequaliser-factor-unique V
open CoequaliserCandidate public using
  (coequaliser-factoring
  ;coequaliser-factor-unique
  ;coequaliser-universal
  )
```

20.4 Relation.Binary.Setoid.Equaliser

Equalisers of Setoid homomorphisms are subobjects determined by where their images are equal; we provide general subobjects for substitutive predicates first:

```
SubSetoid : {i k₁ k₂ : Level} (S : Setoid i k₁)
          → {p : ⌊ S ⌋ → Set k₂} → (p-subst : (x y : ⌊ S ⌋) → x ≈⌊ S ⌋ y → p x → p y)
          → Σ [S′ : Setoid (i ⊍ k₂) k₁ ] (S′ ⟶ S)
SubSetoid S {p} p-subst
  = record {Carrier = Σ ⌊ S ⌋ p
           ;_≈_ = Setoid._≈_ S on proj₁
           ;isEquivalence = record
             {refl = Setoid.refl S
             ;sym = Setoid.sym S
             ;trans = Setoid.trans S
             }
           }
  ,  record {_⟨$⟩_ = proj₁;cong = id}
```

Given two setoid homomorphisms, their equaliser is the subobject determined by their image overlap:

```
CommonImg : {i₁ i₂ k₂ : Level} {A : Set i₁} (B : Setoid i₂ k₂) (F G : A → ⌊ B ⌋) → A → Set k₂
CommonImg B F G x = F x ≈⌊ B ⌋ G x
```

The predicate CommonImg B F G is substitutive by construction:

```
CommonImg-subst : {i₁ i₂ k₁ k₂ : Level} {A : Setoid i₁ k₁} (B : Setoid i₂ k₂) (F G : A ⟶ B) →
  (x y : ⌊ A ⌋) → x ≈⌊ A ⌋ y → CommonImg B (_⟨$⟩_ F) (_⟨$⟩_ G) x
    → CommonImg B (_⟨$⟩_ F) (_⟨$⟩_ G) y
CommonImg-subst B F G x y x≈y Fx≈Gx = let open SetoidCalc B in ≈-begin
    F ⟨$⟩ y
  ≈˘⟨ cong F x≈y ⟩
    F ⟨$⟩ x
  ≈⟨ Fx≈Gx ⟩
    G ⟨$⟩ x
  ≈⟨ cong G x≈y ⟩
    G ⟨$⟩ y
  □
```

This allows us to define the subobject for the equaliser:

```
equaliser : {i₁ i₂ k₁ k₂ : Level} {S : Setoid i₁ k₁} {T : Setoid i₂ k₂}
          → (F G : S ⟶ T) → Σ [S′ : Setoid (i₁ ⊍ k₂) k₁ ] (S′ ⟶ S)
equaliser {S = S} {T} F G = SubSetoid S {CommonImg T (_⟨$⟩_ F) (_⟨$⟩_ G)}
                                        (CommonImg-subst T F G)
```

It remains to show that this actually produces an equaliser — it equalises:

```
equaliser-prop : {i₁ i₂ k₁ k₂ : Level} {S : Setoid i₁ k₁} {T : Setoid i₂ k₂}
               → (F G : S ⟶ T)
               → let Qq = equaliser F G
                     Q = proj₁ Qq
                     E = proj₂ Qq
                 in Setoid._≈_ (Q ⇨ T) (F ∘ E) (G ∘ E)
equaliser-prop {T = T} F G {x, Fx≈Gx} {y, Fy≈Gy} x≈y = let open SetoidCalc T in ≈-begin
    F ⟨$⟩ x
  ≈⟨ cong F x≈y ⟩
    F ⟨$⟩ y
  ≈⟨ Fy≈Gy ⟩
    G ⟨$⟩ y
  □
```

For universality, we move into a local module where we assume an equaliser candidate to be given:

```
private
  module CoequaliserCandidate
    {i₁ i₂ i₃ k₁ k₂ k₃ : Level} {S : Setoid i₁ k₁} {T : Setoid i₂ k₂}
    (F G : S ⟶ T)
    {R : Setoid i₃ k₃}
    (H : R ⟶ S)
    (F∘H≈G∘H : Setoid._≈_ (R ⇨ T) (F ∘ H) (G ∘ H))
    where
    Qq = equaliser F G
    Q = proj₁ Qq
    E = proj₂ Qq
    U : R ⟶ Q
    U = record {_⟨$⟩_ = λ r → H ⟨$⟩ r, F∘H≈G∘H (Setoid.refl R); cong = cong H}
    equaliser-factors : Σ [U : R ⟶ Q ] H ≈⌊ R ⇨ S ⌋ E ∘ U
    equaliser-factors = U, cong H
```

```
    equaliser-factors-unique : (V : R ⟶ Q) → H ≈⌊ R ⇨ S ⌋ E ∘ V → U ≈⌊ R ⇨ Q ⌋ V
    equaliser-factors-unique V H≈E∘V x≈y  =  H≈E∘V x≈y

    equaliser-universal : ∃! (Setoid._≈_ (R ⇨ Q)) λ (U : R ⟶ Q) → H ≈⌊ R ⇨ S ⌋ E ∘ U
    equaliser-universal  =  U, cong H
                             , λ {V} → equaliser-factors-unique V
open CoequaliserCandidate public using
    (equaliser-factors
    ; equaliser-factors-unique
    ; equaliser-universal
    )
  renaming
    (U to equaliser-univMor)
```

20.5 Relation.Binary.Setoid.Category

```
open CatFinColimits using
  (HasCoEqualisers; HasCoproducts; Pushout; HasPushouts; constructPushout
  ; IsInitial; HasInitialObject)

module _ (i k : Level) where

  setoidCompOp : CompOp {ℓsuc (i ⊍ k)} {i ⊍ k} {i ⊍ k} {Setoid i k} _⇨_
  setoidCompOp  =  record
    {_⨾_  =  λ F G → G ∘ F
    ; ⨾-cong  =  λ {A} {B} {C} {F₁} {F₂} {G₁} {G₂} F₁≈F₂ G₁≈G₂ {x} {y} x≈y → G₁≈G₂ (F₁≈F₂ x≈y)
    ; ⨾-assoc  =  λ {A} {B} {C} {D} {F} {G} {H} {x} {y} x≈y → cong H (cong G (cong F x≈y))
    }

  setoidSemigroupoid : Semigroupoid (i ⊍ k) (i ⊍ k) (Setoid i k)
  setoidSemigroupoid  =  record {Hom  =  _⇨_; compOp  =  setoidCompOp}

  setoidCategory : Category (i ⊍ k) (i ⊍ k) (Setoid i k)
  setoidCategory  =  record
    {semigroupoid  =  setoidSemigroupoid
    ; idOp  =  record
      {Id       =  id
      ; leftId   =  λ {A} {B} {f} → cong f
      ; rightId  =  λ {A} {B} {f} → cong f
      }
    }

setoidHasCoEqualisers : (i k : Level) → HasCoEqualisers (setoidCategory i (i ⊍ k))
setoidHasCoEqualisers i k  =  record
  {coequ  =  coequaliser
  ; _⨾⇈_  =  coequaliser-prop
  ; ⇈-factoring  =  coequaliser-factoring
  ; ⇈-factor-unique  =  coequaliser-factor-unique
  }

setoidHasCoproducts : (i k : Level) → HasCoproducts (setoidCategory i (i ⊍ k))
setoidHasCoproducts i k  =  record
  {_⊞_  =  _⊎⊎_
```

```
  ; ι  =  λ {A} {B} → Inj₁ A B
  ; κ  =  λ {A} {B} → Inj₂ A B
  ; isCoproduct  =  λ {A} {B} {C} F G → record
    {univMor  =  ⟨[ F, G ]⟩
    ; univMor-factors-left  =  proj₁ (⊎-factors₀ F G)
    ; univMor-factors-right  =  proj₂ (⊎-factors₀ F G)
    ; univMor-unique  =  λ {V} ι⨾V≈F κ⨾V≈G {x} {y}
                   → Setoid.sym (A ⊎⊎ B ⇨ C) {⟨[ F, G ]⟩} {V} (⊎-factors-unique F G V (ι⨾V≈F, κ⨾V≈G))
    }
  }


setoidPushout : {i k : Level} {A B C : Setoid i (i ⊍ k)}
             → (F : A ⟶ B) (G : A ⟶ C)
             → Pushout (setoidCategory i (i ⊍ k)) F G
setoidPushout {i} {k}  =  constructPushout (setoidCategory i (i ⊍ k))
                                           (setoidHasCoproducts i k)
                                           (setoidHasCoEqualisers i k)


setoidHasPushouts : (i k : Level) → HasPushouts (setoidCategory i (i ⊍ k))
setoidHasPushouts i k  =  setoidPushout {i} {k}


setoidIsInitial :  {i k : Level} (I : Setoid i k)
               → (⌊ I ⌋ → ⊥) → IsInitial (setoidCategory i k) I
setoidIsInitial I no {A}  =  U, unique
  where
    U₀ : ⌊ I ⌋ → ⌊ A ⌋
    U₀ i with no i
    ... | ()
    cng : {i j : ⌊ I ⌋} → i ≈⌊ I ⌋ j → U₀ i ≈⌊ A ⌋ U₀ j
    cng {i} with no i
    ... | ()
    U : I ⟶ A
    U  =  record {_⟨$⟩_  =  U₀; cong  =  cng}
    unique : (V : I ⟶ A) → V ≈⌊ I ⇨ A ⌋ U
    unique V {i} with no i
    ... | ()


setoidMkInitialObject : {i k : Level} (I : Setoid i k) → (⌊ I ⌋ → ⊥) → HasInitialObject (setoidCategory i k)
setoidMkInitialObject I no  =  record {①  =  I; isInitial  =  setoidIsInitial I no}


setoidHasInitialObject : {i k : Level} → HasInitialObject (setoidCategory i k)
setoidHasInitialObject  =  setoidMkInitialObject GenEmpty.⊥ (λ ())


open CatFinLimits using
  (IsTerminal; HasTerminalObject; HasProducts; HasEqualisers
  ; Pullback; HasPullbacks; constructPullback)


setoidIsTerminal : {i k : Level} (T : Setoid i k)
                 → ⌊ T ⌋ → ({x y : ⌊ T ⌋} → x ≈⌊ T ⌋ y) → IsTerminal (setoidCategory i k) T
setoidIsTerminal T t eq {A}  =  U, λ _ _ → eq
  where
    U : A ⟶ T
    U  =  record {_⟨$⟩_  =  λ _ → t; cong  =  λ _ → Setoid.refl T}
```

```
setoidMkTerminalObject : {i k : Level} (T : Setoid i k)
  → ⌊ T ⌋ → ({x y : ⌊ T ⌋} → x ≈⌊ T ⌋ y) → HasTerminalObject (setoidCategory i k)
setoidMkTerminalObject T t eq = record {𝟙 = T; isInitial = setoidIsTerminal T t eq}


setoidHasTerminalObject : {i k : Level} → HasTerminalObject (setoidCategory i k)
setoidHasTerminalObject = setoidMkTerminalObject GenUnit.⊤ _ _


setoidHasProducts : (i k : Level) → HasProducts (setoidCategory i k)
setoidHasProducts i k = record
  {_⊠_ = _××_
  ;π = λ {A} {B} → Proj₁ A B
  ;ρ = λ {A} {B} → Proj₂ A B
  ;isProduct = λ {A} {B} {Z} F G → record
    {univMor = record {_⟨$⟩_ = λ z → F ⟨$⟩ z, G ⟨$⟩ z
                      ;cong = λ z₁≈z₂ → (cong F z₁≈z₂), (cong G z₁≈z₂)}
    ;univMor-factors-left = λ z₁≈z₂ → cong F z₁≈z₂
    ;univMor-factors-right = λ z₁≈z₂ → cong G z₁≈z₂
    ;univMor-unique = λ {V} V⨾π≈F V⨾ρ≈G z₁≈z₂ → V⨾π≈F z₁≈z₂, V⨾ρ≈G z₁≈z₂
    }
  }


setoidHasEqualisers : (i k : Level) → HasEqualisers (setoidCategory (i ⊍ k) i)
setoidHasEqualisers i k = record
  {coequ = equaliser
  ; _⨾⇈_ = equaliser-prop
  ; ⇈-factoring = equaliser-factors
  ; ⇈-factor-unique = equaliser-factors-unique
  }


setoidPullback : (i k : Level) → {A B C : Setoid (i ⊍ k) i}
               → (F : B ⟶ A) (G : C ⟶ A)
               → Pullback (setoidCategory (i ⊍ k) i) F G
setoidPullback i k = constructPullback (setoidCategory (i ⊍ k) i)
                                       (setoidHasProducts (i ⊍ k) i)
                                       (setoidHasEqualisers i k)


setoidHasPullbacks : (i k : Level) → HasPullbacks (setoidCategory (i ⊍ k) i)
setoidHasPullbacks i k = setoidPullback i k
```

Chapter 21

Abstract Representations of Concrete Relations

Since many implementations of concrete relations, including the relation types of the standard library and of Chapter 18, can relate elements taken from Sets of different Levels, the abstract theories of Part I and Part II can capture only small slices of the potential applications of such datatypes.

In the formalisations in this chapter, we carefully allow maximal universe polymorphism, so that the algebraic laws shown in the individual theories can be applied also to concrete relations "across Levels".

21.1 Relation.Binary.ElemRel.All

Re-export only:

```
open import Relation.Binary.Poset.ElemSet              public  -- Sect. 21.2
open import Relation.Binary.ElemRel                    public  -- Sect. 21.3
open import Relation.Binary.ElemRel.Core               public  -- Sect. 21.4

open import Relation.Binary.ElemRel.Conv               public  -- Sect. 21.5
open import Relation.Binary.ElemRel.Comp               public  -- Sect. 21.6
open import Relation.Binary.ElemRel.Id                 public  -- Sect. 21.7
open import Relation.Binary.ElemRel.SubId              public  -- Sect. 21.8

open import Relation.Binary.ElemRel.Conv2              public  -- Sect. 21.9
open import Relation.Binary.ElemRel.Comp3              public  -- Sect. 21.10
open import Relation.Binary.ElemRel.Involution         public  -- Sect. 21.11
open import Relation.Binary.ElemRel.CompAssoc          public  -- Sect. 21.12
open import Relation.Binary.ElemRel.Comp3UnionL        public  -- Sect. 21.13
open import Relation.Binary.ElemRel.Comp3UnionR        public  -- Sect. 21.14
open import Relation.Binary.ElemRel.Conv2-IdL          public  -- Sect. 21.15
open import Relation.Binary.ElemRel.Conv2-IdR          public  -- Sect. 21.16
open import Relation.Binary.ElemRel.LeftSubId          public  -- Sect. 21.17
open import Relation.Binary.ElemRel.RightSubId         public  -- Sect. 21.18

open import Relation.Binary.ElemRel.Dedekind           public  -- Sect. 21.19

open import Relation.Binary.ElemRel.Equivalence        public  -- Sect. 21.20
open import Relation.Binary.ElemSet.SetReprConversions public  -- Sect. 21.21

open import Relation.Binary.ElemRel.Dom                public  -- Sect. 21.22
open import Relation.Binary.ElemRel.Ran                public  -- Sect. 21.23
open import Relation.Binary.ElemRel.Conv2-Ran          public  -- Sect. 21.24

open import Relation.Binary.ElemRel.Homogeneous        public  -- Sect. 21.25
open import Relation.Binary.ElemRel.SubIdCong          public  -- Sect. 21.26
open import Relation.Binary.ElemSet.ReprIso            public  -- Sect. 21.27
```

21.2 Relation.Binary.Poset.ElemSet

```
module ElemSubset {ℓa₀ ℓa₁ : Level} (Elem : Setoid ℓa₀ ℓa₁)
                  {j ℓ : Level} {SetRepr : Set j} (_∈_ : ⌊ Elem ⌋ → SetRepr → Set ℓ) where
  private
    ℓa = ℓa₀ ⊍ ℓa₁
  open Setoid Elem using () renaming
    (Carrier to Elem₀; _≈_ to _~_; refl to ~-refl; reflexive to ~-reflexive; sym to ~-sym; trans to ~-trans)

  module ElemInclusion where
    infix 4 _⇒_
    _⇒_ : Rel SetRepr (ℓ ⊍ ℓa₀)
    _⇒_ R S = (a : Elem₀) → a ∈ R → a ∈ S
    ⇒-refl : {R : SetRepr} → R ⇒ R
    ⇒-refl {R} a a∈R = a∈R
    ⇒-trans : {Q R S : SetRepr} → Q ⇒ R → R ⇒ S → Q ⇒ S
    ⇒-trans Q⇒R R⇒S a a∈Q = R⇒S a (Q⇒R a a∈Q)
  open ElemInclusion public

  record IsElemSet {k₁ k₂ : Level} (_≈_ : Rel SetRepr k₁) (_⊆_ : Rel SetRepr k₂)
                   : Set (j ⊍ k₁ ⊍ k₂ ⊍ ℓ ⊍ ℓa) where
    field
      ∈-subst₁ : {R : SetRepr} {a₁ a₂ : Elem₀} → a₁ ~ a₂ → a₁ ∈ R → a₂ ∈ R
      ≈-to-⇒    : {R S : SetRepr} → R ≈ S → R ⇒ S
      ≈-to-⇔    : {R S : SetRepr} → R ≈ S → (R ⇒ S) × (S ⇒ R)
      ⊆-to-⇒    : {R S : SetRepr} → R ⊆ S → R ⇒ S
      ⊆-from-⇒  : {R S : SetRepr} → R ⇒ S → R ⊆ S
      ≈-from-⇔  : {R S : SetRepr} → (R ⇒ S) × (S ⇒ R) → R ≈ S
    isUniversal : SetRepr → Set (ℓ ⊍ ℓa₀)
    isUniversal S = (a : Elem₀) → a ∈ S

record ElemSetMeet (intersection : SetRepr → SetRepr → SetRepr) : Set (j ⊍ ℓ ⊍ ℓa₀) where
  field
    from-∈-intersection : (R S : SetRepr) → (p : Elem₀) → p ∈ intersection R S → p ∈ R × p ∈ S
    to-∈-intersection : (R S : SetRepr) → (p : Elem₀) → p ∈ R → p ∈ S → p ∈ intersection R S

record ElemSetJoin (union : SetRepr → SetRepr → SetRepr) : Set (j ⊍ ℓ ⊍ ℓa₀) where
  field
    from-∈-union : (R S : SetRepr) → (p : Elem₀) → p ∈ union R S → p ∈ R ⊎ p ∈ S
    to-∈-union : (R S : SetRepr) → (p : Elem₀) → p ∈ R ⊎ p ∈ S → p ∈ union R S

record ElemSetDifference (difference : SetRepr → SetRepr → SetRepr) : Set (j ⊍ ℓ ⊍ ℓa₀) where
  field
    from-∈-difference : (R S : SetRepr) → (p : Elem₀) → p ∈ difference R S → (p ∈ R) × (p ∈ S → ⊥)
    to-∈-difference : (R S : SetRepr) → (p : Elem₀) → p ∈ R → (p ∈ S → ⊥) → p ∈ difference R S

record ElemSetComplement (complement : SetRepr → SetRepr) : Set (j ⊍ ℓ ⊍ ℓa₀) where
  field
    from-∈-complement : (R : SetRepr) → (p : Elem₀) → p ∈ complement R → p ∈ R → ⊥
    to-∈-complement : (R : SetRepr) → (p : Elem₀) → (p ∈ R → ⊥) → p ∈ complement R
```

We keep the derived Setoid and Poset structures and ingredients in a separate module, so that **open** IsElemSet brings only the fields into scope, while the Setoid and Poset may in general be provided separately, and are not necessarily implemented on top of the IsElemSet ingredients. At the sametime, we move to a separate top-level module ElemSubset′ the parameterisation of which differes from that of ElemSubset only in that Elem and _∊_ are now implicit parameters, since they can be inferred from IsElemSet types.

```
module ElemSubset′ {ℓa₀ ℓa₁ : Level} {Elem : Setoid ℓa₀ ℓa₁}
  {j ℓ : Level} {SetRepr : Set j} {_∊_ : ⌊ Elem ⌋ → SetRepr → Set ℓ} where
  open ElemSubset Elem _∊_


  module ElemSet-Poset {k₁ k₂ : Level} {_≈_ : Rel SetRepr k₁} {_⊆_ : Rel SetRepr k₂}
                       (isElemSet : IsElemSet _≈_ _⊆_) where
    open IsElemSet isElemSet
    ≈-refl : {R : SetRepr} → R ≈ R
    ≈-refl {R} = ≈-from-⇔ (⇒-refl, ⇒-refl)
    ≈-sym : {R S : SetRepr} → R ≈ S → S ≈ R
    ≈-sym {R} {S} R≈S with ≈-to-⇔ R≈S
    ... | R⇒S, S⇒R = ≈-from-⇔ (S⇒R, R⇒S)
    ≈-trans : {Q R S : SetRepr} → Q ≈ R → R ≈ S → Q ≈ S
    ≈-trans Q≈R R≈S with ≈-to-⇔ Q≈R | ≈-to-⇔ R≈S
    ... | Q⇒R, R⇒Q | R⇒S, S⇒R = ≈-from-⇔ (⇒-trans Q⇒R R⇒S, ⇒-trans S⇒R R⇒Q)
    isEquivalence : IsEquivalence _≈_
    isEquivalence = record {refl = ≈-refl; sym = ≈-sym; trans = ≈-trans}
    setoid : Setoid j k₁
    setoid = record {Carrier = SetRepr; _≈_ = _≈_; isEquivalence = isEquivalence}
    ⊆-refl : {R : SetRepr} → R ⊆ R
    ⊆-refl = ⊆-from-⇒ ⇒-refl
    ⊆-reflexive : {R S : SetRepr} → R ≈ S → R ⊆ S
    ⊆-reflexive R≈S = ⊆-from-⇒ (≈-to-⇒ R≈S)
    ⊆-trans : {Q R S : SetRepr} → Q ⊆ R → R ⊆ S → Q ⊆ S
    ⊆-trans Q⊆R R⊆S = ⊆-from-⇒ (⇒-trans (⊆-to-⇒ Q⊆R) (⊆-to-⇒ R⊆S))
    ⊆-trans₁ : {Q R S : SetRepr} → Q ⊆ R → R ≈ S → Q ⊆ S
    ⊆-trans₁ Q⊆R R≈S = ⊆-from-⇒ (⇒-trans (⊆-to-⇒ Q⊆R) (≈-to-⇒ R≈S))
    ⊆-trans₂ : {Q R S : SetRepr} → Q ≈ R → R ⊆ S → Q ⊆ S
    ⊆-trans₂ Q≈R R⊆S = ⊆-from-⇒ (⇒-trans (≈-to-⇒ Q≈R) (⊆-to-⇒ R⊆S))
    ⊆-antisym : {Q R : SetRepr} → Q ⊆ R → R ⊆ Q → Q ≈ R
    ⊆-antisym Q⊆R R⊆Q = ≈-from-⇔ (⊆-to-⇒ Q⊆R, ⊆-to-⇒ R⊆Q)
    isPreorder : IsPreorder _≈_ _⊆_
    isPreorder = record {isEquivalence = isEquivalence
                        ; reflexive = ⊆-reflexive; trans = ⊆-trans}
    isPartialOrder : IsPartialOrder _≈_ _⊆_
    isPartialOrder = record {isPreorder = isPreorder; antisym = ⊆-antisym}
    poset : Poset j k₁ k₂
    poset = record
      {Carrier = SetRepr; _≈_ = _≈_; _≤_ = _⊆_
      ; isPartialOrder = isPartialOrder
      }


module ElemSet-Meet {k₁ k₂ : Level} {_≈_ : Rel SetRepr k₁} {_⊆_ : Rel SetRepr k₂}
                    (isElemSet : IsElemSet _≈_ _⊆_) (IsPO : IsPartialOrder _≈_ _⊆_)
                    {intersection : SetRepr → SetRepr → SetRepr}
                    (EM : ElemSetMeet intersection) where
  PO : Poset j k₁ k₂
  PO = record {isPartialOrder = IsPO}
  open IsElemSet isElemSet
  open ElemSetMeet EM
```

```
  intersection-lower₁ : {R S : SetRepr} → intersection R S ⊆ R
  intersection-lower₁ {R} {S} = ⊆-from-⇒ (λ a a∈M → proj₁ (from-∈-intersection _ _ a a∈M))
  intersection-lower₂ : {R S : SetRepr} → intersection R S ⊆ S
  intersection-lower₂ {R} {S} = ⊆-from-⇒ (λ a a∈M → proj₂ (from-∈-intersection _ _ a a∈M))
  intersection-universal : {R S X : SetRepr} → X ⊆ R → X ⊆ S → X ⊆ intersection R S
  intersection-universal {R} {S} {X} X⊆R X⊆S = ⊆-from-⇒ (λ a a∈X → to-∈-intersection _ _ a (⊆-to-⇒ X⊆R a a∈X)
                                                                                              (⊆-to-⇒ X⊆S a a∈X))
  meet : (R S : SetRepr) → PosetMeet.Meet PO R S
  meet R S = record
    {value = intersection R S
    ; proof = record
      {bound₁    = intersection-lower₁
      ; bound₂    = intersection-lower₂
      ; universal = intersection-universal
      }
    }

module ElemSet-Join {k₁ k₂ : Level} {_≈_ : Rel SetRepr k₁} {_⊆_ : Rel SetRepr k₂}
                    (isElemSet : IsElemSet _≈_ _⊆_) (IsPO : IsPartialOrder _≈_ _⊆_)
                    {union : SetRepr → SetRepr → SetRepr}
                    (EJ : ElemSetJoin union) where
  PO : Poset j k₁ k₂
  PO = record {isPartialOrder = IsPO}
  open IsElemSet isElemSet
  open ElemSetJoin EJ
  union-upper₁-⇒ : {R S : SetRepr} → R ⇒ union R S
  union-upper₁-⇒ {R} {S} a a∈R = to-∈-union R S a (inj₁ a∈R)
  union-upper₁ : {R S : SetRepr} → R ⊆ union R S
  union-upper₁ = ⊆-from-⇒ union-upper₁-⇒
  union-upper₂-⇒ : {R S : SetRepr} → S ⇒ union R S
  union-upper₂-⇒ {R} {S} a a∈S = to-∈-union R S a (inj₂ a∈S)
  union-upper₂ : {R S : SetRepr} → S ⊆ union R S
  union-upper₂ = ⊆-from-⇒ union-upper₂-⇒
  -- union-upper₂ R S = ⊆-from-⇒ (λ a a∈S → to-∈-union R S a (inj₂ a∈S))
  union-universal-⇒ : {R S X : SetRepr} → R ⇒ X → S ⇒ X → union R S ⇒ X
  union-universal-⇒ {R} {S} {X} R⇒X S⇒X a a∈RS with from-∈-union R S a a∈RS
  ... | inj₁ a∈R = R⇒X a a∈R
  ... | inj₂ a∈S = S⇒X a a∈S
  union-universal : {R S X : SetRepr} → R ⊆ X → S ⊆ X → union R S ⊆ X
  union-universal {R} {S} {X} R⊆X S⊆X = ⊆-from-⇒ (union-universal-⇒ (⊆-to-⇒ R⊆X) (⊆-to-⇒ S⊆X))
  join : (R S : SetRepr) → PosetJoin.Join PO R S
  join R S = record
    {value = union R S
    ; proof = record
      {bound₁    = union-upper₁
      ; bound₂    = union-upper₂
      ; universal = union-universal
      }
    }

module ElemSet-Lattice {k₁ k₂ : Level} {_≈_ : Rel SetRepr k₁} {_⊆_ : Rel SetRepr k₂}
                       (isElemSet : IsElemSet _≈_ _⊆_) (IsPO : IsPartialOrder _≈_ _⊆_)
                       {intersection : SetRepr → SetRepr → SetRepr}
                       (EM : ElemSetMeet intersection)
                       {union : SetRepr → SetRepr → SetRepr}
                       (EJ : ElemSetJoin union) where
```

```
  PO : Poset j k₁ k₂
  PO = record {isPartialOrder = IsPO}
  open IsElemSet isElemSet
  open ElemSetMeet EM
  open ElemSetJoin EJ
  ∩-∪-subdistribR-⇒ : {Q R S : SetRepr} → intersection Q (union R S) ⇒ union (intersection Q R) (intersection Q S)
  ∩-∪-subdistribR-⇒ {Q} {R} {S} a a∈Q∩RS with from-∈-intersection _ _ a a∈Q∩RS
  ... | a∈Q, a∈RS with from-∈-union _ _ a a∈RS
  ... | inj₁ a∈R = to-∈-union _ _ a (inj₁ (to-∈-intersection _ _ a a∈Q a∈R))
  ... | inj₂ a∈S = to-∈-union _ _ a (inj₂ (to-∈-intersection _ _ a a∈Q a∈S))
  ∩-∪-subdistribR : {Q R S : SetRepr} → intersection Q (union R S) ⊆ union (intersection Q R) (intersection Q S)
  ∩-∪-subdistribR = ⊆-from-⇒ ∩-∪-subdistribR-⇒


open ElemSubset′ public


IsElemSet′ : {ℓa₀ ℓa₁ : Level} (Elem : Setoid ℓa₀ ℓa₁)
               {j k₁ k₂ ℓ : Level} (SetRepr : Poset j k₁ k₂) (_∈_ : ⌊ Elem ⌋ → ⌊ SetRepr ≤⌋ → Set ℓ)
             → Set (j ⊍ k₁ ⊍ k₂ ⊍ ℓ ⊍ ℓa₀ ⊍ ℓa₁)
IsElemSet′ Elem SetRepr _∈_ = let open Poset SetRepr in ElemSubset.IsElemSet Elem _∈_ _≈_ _≤_


module IsElemSet′ {ℓa₀ ℓa₁ : Level} {Elem : Setoid ℓa₀ ℓa₁}
                  {j k₁ k₂ ℓ : Level} (SetRepr : Poset j k₁ k₂)
                  {_∈_ : Setoid.Carrier Elem → ⌊ SetRepr ≤⌋ → Set ℓ}
                  (isElemSet : IsElemSet′ Elem SetRepr _∈_) where
  open ElemSubset Elem _∈_
  open ElemInclusion public
  open IsElemSet isElemSet public
  SetRepr≈ : Setoid j k₁
  SetRepr≈ = posetSetoid SetRepr
  open Setoid′ SetRepr≈ public renaming (Carrier to SetRepr₀)
  open Poset-round SetRepr public


module ElemSetPoset {ℓa₀ ℓa₁ : Level} {Elem : Setoid ℓa₀ ℓa₁}
                    {j k₁ k₂ ℓ : Level} (SetRepr : Poset j k₁ k₂)
                    {_∈_ : ⌊ Elem ⌋ → ⌊ SetRepr ≤⌋ → Set ℓ} where
  open ElemSubset Elem _∈_


module ElemSet-Meet′ (isElemSet : IsElemSet′ Elem SetRepr _∈_)
                     {intersection : ⌊ SetRepr ≤⌋ → ⌊ SetRepr ≤⌋ → ⌊ SetRepr ≤⌋}
                     (EM : ElemSetMeet intersection)
   = ElemSet-Meet isElemSet (Poset.isPartialOrder SetRepr) EM


module ElemSet-Join′ (isElemSet : IsElemSet′ Elem SetRepr _∈_)
                     {union : ⌊ SetRepr ≤⌋ → ⌊ SetRepr ≤⌋ → ⌊ SetRepr ≤⌋}
                     (EJ : ElemSetJoin union)
   = ElemSet-Join isElemSet (Poset.isPartialOrder SetRepr) EJ


module ElemSet-Lattice′ (isElemSet : IsElemSet′ Elem SetRepr _∈_)
                        {intersection : ⌊ SetRepr ≤⌋ → ⌊ SetRepr ≤⌋ → ⌊ SetRepr ≤⌋}
                        (EM : ElemSetMeet intersection)
                        {union : ⌊ SetRepr ≤⌋ → ⌊ SetRepr ≤⌋ → ⌊ SetRepr ≤⌋}
                        (EJ : ElemSetJoin union)
   = ElemSet-Lattice isElemSet (Poset.isPartialOrder SetRepr) EM EJ
```

```
open ElemSubset public
open ElemSetPoset public

record ElemSet {ℓa₀ ℓa₁ : Level} (Elem : Setoid ℓa₀ ℓa₁) (j k₁ k₂ ℓ : Level)
                  : Set (ℓsuc (j ⊍ k₁ ⊍ k₂ ⊍ ℓ) ⊍ ℓa₀ ⊍ ℓa₁) where
  field
    SetRepr : Poset j k₁ k₂
    _∈_ : ⌊ Elem ⌋ → ⌊ SetRepr ≤⌋ → Set ℓ
    isElemSet : IsElemSet′ Elem SetRepr _∈_
  open IsElemSet′ SetRepr isElemSet public

module ElemSet-Meet″ {ℓa₀ ℓa₁ : Level} {Elem : Setoid ℓa₀ ℓa₁}
                          {j k₁ k₂ ℓ : Level} (elemSet : ElemSet Elem j k₁ k₂ ℓ)
                          {intersection : let S = ElemSet.SetRepr₀ elemSet in S → S → S}
                          (EM : ElemSetMeet Elem (ElemSet._∈_ elemSet) intersection)
  = ElemSet-Meet′ (ElemSet.SetRepr elemSet) (ElemSet.isElemSet elemSet) EM

module ElemSet-Join″ {ℓa₀ ℓa₁ : Level} {Elem : Setoid ℓa₀ ℓa₁}
                         {j k₁ k₂ ℓ : Level} (elemSet : ElemSet Elem j k₁ k₂ ℓ)
                         {union : let S = ElemSet.SetRepr₀ elemSet in S → S → S}
                         (EJ : ElemSetJoin Elem (ElemSet._∈_ elemSet) union)
  = ElemSet-Join′ (ElemSet.SetRepr elemSet) (ElemSet.isElemSet elemSet) EJ

module ElemSet-Lattice″ {ℓa₀ ℓa₁ : Level} {Elem : Setoid ℓa₀ ℓa₁}
                            {j k₁ k₂ ℓ : Level} (elemSet : ElemSet Elem j k₁ k₂ ℓ)
                            {intersection : let S = ElemSet.SetRepr₀ elemSet in S → S → S}
                            (EM : ElemSetMeet Elem (ElemSet._∈_ elemSet) intersection)
                            {union : let S = ElemSet.SetRepr₀ elemSet in S → S → S}
                            (EJ : ElemSetJoin Elem (ElemSet._∈_ elemSet) union)
  = ElemSet-Lattice′ (ElemSet.SetRepr elemSet) (ElemSet.isElemSet elemSet) EM EJ

module ElemSetR {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
                    {ℓr₀ ℓr₁ ℓr₂ ℓr₃ : Level} (R : ElemSet A ℓr₀ ℓr₁ ℓr₂ ℓr₃) where
  open ElemSet R public using () renaming
    (SetRepr≈ to R≈; SetRepr₀ to R₀; isElemSet to R-isElemSet
    ; _∈_ to _∈R_; _≈_ to _≈R_; _⊆_ to _⊆R_; _⇒_ to _⇒R_
    ; ⊆-from-⇒ to ⊆R-from-⇒
    ; ⊆-to-⇒   to ⊆R-to-⇒
    ; ≈-from-⇔ to ≈R-from-⇔
    ; ≈-to-⇔   to ≈R-to-⇔
    ; ≈-to-⇒   to ≈R-to-⇒
    )
module ElemSetS {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
                    {ℓs₀ ℓs₁ ℓs₂ ℓs₃ : Level} (S : ElemSet A ℓs₀ ℓs₁ ℓs₂ ℓs₃) where
  open ElemSet S public using () renaming
    (SetRepr≈ to S≈; SetRepr₀ to S₀; isElemSet to S-isElemSet
    ; _∈_ to _∈S_; _≈_ to _≈S_; _⊆_ to _⊆S_; _⇒_ to _⇒S_
    ; ⊆-from-⇒ to ⊆S-from-⇒
    ; ⊆-to-⇒   to ⊆S-to-⇒
    ; ≈-from-⇔ to ≈S-from-⇔
    ; ≈-to-⇔   to ≈S-to-⇔
    ; ≈-to-⇒   to ≈S-to-⇒
    )

module TwoElemSets {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
                       {ℓr₀ ℓr₁ ℓr₂ ℓr₃ : Level} (R : ElemSet A ℓr₀ ℓr₁ ℓr₂ ℓr₃)
```

```
                  {ℓs₀ ℓs₁ ℓs₂ ℓs₃ : Level} (S : ElemSet A ℓs₀ ℓs₁ ℓs₂ ℓs₃) where
  open SetoidA A public
  open ElemSetR R public
  open ElemSetS S public
```

21.3 Relation.Binary.ElemRel

For speed of type-checking, this module has been split into submodules that are all re-exported directly from here.

```
open import Relation.Binary.ElemRel.Core public
open import Relation.Binary.ElemRel.Conv public
open import Relation.Binary.ElemRel.Comp public
open import Relation.Binary.ElemRel.Id public
open import Relation.Binary.ElemRel.SubId public
open import Relation.Binary.ElemRel.Conv2 public
open import Relation.Binary.ElemRel.Comp3 public
open import Relation.Binary.ElemRel.Involution public
open import Relation.Binary.ElemRel.CompAssoc public
open import Relation.Binary.ElemRel.Comp3UnionL public
open import Relation.Binary.ElemRel.Comp3UnionR public
open import Relation.Binary.ElemRel.Dedekind public
open import Relation.Binary.ElemRel.Conv2-IdL public
open import Relation.Binary.ElemRel.Conv2-IdR public
open import Relation.Binary.ElemRel.SubIdCong public
open import Relation.Binary.ElemRel.LeftSubId public
open import Relation.Binary.ElemRel.RightSubId public
```

21.4 Relation.Binary.ElemRel.Core

IsElemRel A B {RelRepr = RelRepr} _∈_ _≈_ _⊆_ documents that RelRepr can be considered as representing a set of pairs, with _∈_ as element relation, and that equality _≈_ and inclusion _⊆_ on RelRepr are consistent with that view.

IsElemRel is defined as a separate type from IsElemSet (A ×× B) _∈_ _≈_ _⊆_, for which conversions are defined below, since that type would not allow the constituent setoids A and B to be derived from it as implicit arguments (since they are not derivable from A ×× B).

```
record IsElemRel {ℓa₀ ℓa₁ ℓb₀ ℓb₁ : Level} (A : Setoid ℓa₀ ℓa₁) (B : Setoid ℓb₀ ℓb₁)
                 {j k₁ k₂ ℓ : Level} {RelRepr : Set j}
                 (_∈_ : ⌊ A ⌋ × ⌊ B ⌋ → RelRepr → Set ℓ)
                 (_≈_ : Rel RelRepr k₁) (_⊆_ : Rel RelRepr k₂)
                   : Set (j ⊍ k₁ ⊍ k₂ ⊍ ℓ ⊍ ℓa₀ ⊍ ℓa₁ ⊍ ℓb₀ ⊍ ℓb₁) where
  open SetoidA A
  open SetoidB B
  open ElemInclusion (A ×× B) _∈_
  field
    ∈-subst₁₁  : (R : RelRepr) {a₁ a₂ : A₀} {b : B₀} → a₁ ≈A a₂ → (a₁, b) ∈ R → (a₂, b) ∈ R
    ∈-subst₁₂  : (R : RelRepr) {a : A₀} {b₁ b₂ : B₀} → b₁ ≈B b₂ → (a, b₁) ∈ R → (a, b₂) ∈ R
    ≈-to-⇒     : {R S : RelRepr} → R ≈ S → R ⇒ S
    ≈-to-⇔     : {R S : RelRepr} → R ≈ S → (R ⇒ S) × (S ⇒ R)
    ⊆-to-⇒     : {R S : RelRepr} → R ⊆ S → R ⇒ S
    ⊆-from-⇒   : {R S : RelRepr} → R ⇒ S → R ⊆ S
    ≈-from-⇔   : {R S : RelRepr} → (R ⇒ S) × (S ⇒ R) → R ≈ S
```

```
retract-IsElemRel : {ℓa₀ ℓa₁ ℓb₀ ℓb₁ : Level} {A : Setoid ℓa₀ ℓa₁} {B : Setoid ℓb₀ ℓb₁}
                    {j₁ j₂ k₁ k₂ ℓ : Level} {RelRepr₁ : Set j₁}
                    {_∈_ : ⌊ A ⌋ × ⌊ B ⌋ → RelRepr₁ → Set ℓ}
                    {_≈_ : Rel RelRepr₁ k₁} {_⊆_ : Rel RelRepr₁ k₂}
                    {RelRepr₂ : Set j₂} → (f : RelRepr₂ → RelRepr₁)
                  → IsElemRel A B _∈_ _≈_ _⊆_
                  → IsElemRel A B (λ a r → a ∈ f r) (_≈_ on f) (_⊆_ on f)
retract-IsElemRel f isElemRel = let open IsElemRel isElemRel in record
  {∈-subst₁₁  = λ _ → ∈-subst₁₁ _
  ;∈-subst₁₂  = λ _ → ∈-subst₁₂ _
  ;≈-to-⇒     = ≈-to-⇒
  ;≈-to-⇔     = ≈-to-⇔
  ;⊆-to-⇒     = ⊆-to-⇒
  ;⊆-from-⇒ = ⊆-from-⇒
  ;≈-from-⇔ = ≈-from-⇔
  }

setElemRel : {ℓa₀ ℓa₁ ℓb₀ ℓb₁ : Level} (A : Setoid ℓa₀ ℓa₁) (B : Setoid ℓb₀ ℓb₁)
             {j k₁ k₂ ℓ : Level} {RelRepr : Set j}
             {_∈_ : ⌊ A ⌋ × ⌊ B ⌋ → RelRepr → Set ℓ}
             {_≈_ : Rel RelRepr k₁} {_⊆_ : Rel RelRepr k₂}
           → IsElemSet (A ×× B) _∈_ _≈_ _⊆_
           → IsElemRel A B _∈_ _≈_ _⊆_
setElemRel A B {RelRepr = RelRepr} {_∈_} isElemSet = let
  open SetoidA A
  open SetoidB B
  open IsElemSet (A ×× B) _∈_ isElemSet
  ∈-subst₁₁ : (R : RelRepr) {a₁ a₂ : A₀} {b : B₀} → a₁ ≈A a₂ → (a₁, b) ∈ R → (a₂, b) ∈ R
  ∈-subst₁₁ R a₁≈a₂ a₁Rb = ∈-subst₁ (a₁≈a₂, ≈B-refl) a₁Rb
  ∈-subst₁₂ : (R : RelRepr) {a : A₀} {b₁ b₂ : B₀} → b₁ ≈B b₂ → (a, b₁) ∈ R → (a, b₂) ∈ R
  ∈-subst₁₂ R b₁≈b₂ aRb₁ = ∈-subst₁ (≈A-refl, b₁≈b₂) aRb₁
  in record
    {∈-subst₁₁ = ∈-subst₁₁; ∈-subst₁₂ = ∈-subst₁₂
    ;⊆-to-⇒ = ⊆-to-⇒; ⊆-from-⇒ = ⊆-from-⇒
    ;≈-to-⇒ = ≈-to-⇒; ≈-to-⇔ = ≈-to-⇔; ≈-from-⇔ = ≈-from-⇔
    }

IsElemRel′ : {ℓa₀ ℓa₁ ℓb₀ ℓb₁ : Level} (A : Setoid ℓa₀ ℓa₁) (B : Setoid ℓb₀ ℓb₁)
           → {j k₁ k₂ ℓ : Level} (RelRepr : Poset j k₁ k₂)
           → (_∈_ : ⌊ A ⌋ × ⌊ B ⌋ → ⌊ RelRepr ≤⌋ → Set ℓ)
           → Set (j ⊍ k₁ ⊍ k₂ ⊍ ℓ ⊍ ℓa₀ ⊍ ℓa₁ ⊍ ℓb₀ ⊍ ℓb₁)
IsElemRel′ A B RelRepr _∈_ = let open Poset RelRepr in IsElemRel A B _∈_ _≈_ _≤_

module IsElemRel′ {ℓa₀ ℓa₁ ℓb₀ ℓb₁ : Level} {A : Setoid ℓa₀ ℓa₁} {B : Setoid ℓb₀ ℓb₁}
                  {j k₁ k₂ ℓ : Level} (RelRepr : Poset j k₁ k₂)
                  {_∈_ : ⌊ A ⌋ × ⌊ B ⌋ → ⌊ RelRepr ≤⌋ → Set ℓ}
                  (isElemRel : IsElemRel′ A B RelRepr _∈_) where
  open SetoidA A
  open SetoidB B
  open IsElemRel isElemRel
  open Poset RelRepr using (_≈_) renaming (Carrier to RelRepr₀; _≤_ to _⊆_)
  ∈-subst₁ : {R : RelRepr₀} {p₁ p₂ : A₀ × B₀} → p₁ ≈⌊ A ×× B ⌋ p₂ → p₁ ∈ R → p₂ ∈ R
  ∈-subst₁ {R} (a₁≈a₂, b₁≈b₂) a₁Rb₁ = ∈-subst₁₂ R b₁≈b₂ (∈-subst₁₁ R a₁≈a₂ a₁Rb₁)
  isElemSet : IsElemSet (A ×× B) _∈_ _≈_ _⊆_
  isElemSet = record
    {∈-subst₁ = ∈-subst₁
```

```agda
    ; ⊆-to-⇒  =  ⊆-to-⇒ ; ⊆-from-⇒  =  ⊆-from-⇒
    ; ≈-to-⇒   =  ≈-to-⇒ ; ≈-to-⇔  =  ≈-to-⇔ ; ≈-from-⇔  =  ≈-from-⇔
    }
  isUniversal : RelRepr₀ → Set (ℓ ⊍ ℓa₀ ⊍ ℓb₀)   -- (Re-declaration is cheaper than re-export.)
  isUniversal = IsElemSet.isUniversal isElemSet   -- = (p : ⌊ A ⌋ × ⌊ B ⌋) → p ∈ S

  rel : (R : RelRepr₀) → A₀ → B₀ → Set ℓ
  rel R a₀ b₀ = (a₀ , b₀) ∈ R

  total : (R : RelRepr₀) → Set (ℓ ⊍ ℓa₀ ⊍ ℓb₀)
  total R = (a₀ : A₀) → Σ [ b₀ : B₀ ] (a₀ , b₀) ∈ R

  univalent : (R : RelRepr₀) → Set (ℓ ⊍ ℓa₀ ⊍ ℓb₀ ⊍ ℓb₁)
  univalent R = (a₀ : A₀) (b₀ b₁ : B₀) → (a₀ , b₀) ∈ R → (a₀ , b₁) ∈ R → b₀ ≈B b₁

  surjective : (R : RelRepr₀) → Set (ℓ ⊍ ℓa₀ ⊍ ℓb₀)
  surjective R = (b₀ : B₀) → Σ [ a₀ : A₀ ] (a₀ , b₀) ∈ R

  injective : (R : RelRepr₀) → Set (ℓ ⊍ ℓa₀ ⊍ ℓb₀ ⊍ ℓa₁)
  injective R = (a₀ a₁ : A₀) (b₀ : B₀) → (a₀ , b₀) ∈ R → (a₁ , b₀) ∈ R → a₀ ≈A a₁

  _dom⊆_ : (R S : RelRepr₀) → Set (ℓ ⊍ ℓa₀ ⊍ ℓb₀)
  R dom⊆ S = (a₀ : A₀) (b₀ : B₀) → (a₀ , b₀) ∈ R → Σ [ b₁ : B₀ ] (a₀ , b₁) ∈ S

  open ElemInclusion (A ×× B) _∈_ public
  open IsElemRel isElemRel public
  open Setoid′ (posetSetoid RelRepr) public renaming (Carrier to RelRepr₀)
  open Poset-round RelRepr public

record ElemRel {ℓa₀ ℓa₁ ℓb₀ ℓb₁ : Level} (A : Setoid ℓa₀ ℓa₁) (B : Setoid ℓb₀ ℓb₁)
               (j k₁ k₂ ℓ : Level) : Set (ℓsuc (j ⊍ k₁ ⊍ k₂ ⊍ ℓ) ⊍ ℓa₀ ⊍ ℓa₁ ⊍ ℓb₀ ⊍ ℓb₁) where
  field
    RelRepr : Poset j k₁ k₂
    _∈_ : ⌊ A ⌋ × ⌊ B ⌋ → ⌊ RelRepr ≤⌋ → Set ℓ
    isElemRel : IsElemRel′ A B RelRepr _∈_
  elemSet : ElemSet (A ×× B) j k₁ k₂ ℓ
  elemSet = record
    {SetRepr = RelRepr
    ; _∈_ = _∈_
    ; isElemSet = IsElemRel′.isElemSet RelRepr isElemRel
    }
  open IsElemRel′ RelRepr isElemRel public
```

21.5 Relation.Binary.ElemRel.Conv

```agda
record ElemRelConv
    {ℓa₀ ℓa₁ : Level} (A : Setoid ℓa₀ ℓa₁)
    {ℓb₀ ℓb₁ : Level} (B : Setoid ℓb₀ ℓb₁)
    {ℓr₀ : Level} {ReprAB : Set ℓr₀} {ℓ∈AB : Level} (_∈AB_ : ⌊ A ⌋ × ⌊ B ⌋ → ReprAB → Set ℓ∈AB)
    {ℓs₀ : Level} {ReprBA : Set ℓs₀} {ℓ∈BA : Level} (_∈BA_ : ⌊ B ⌋ × ⌊ A ⌋ → ReprBA → Set ℓ∈BA)
    (conv : ReprAB → ReprBA) : Set (ℓa₀ ⊍ ℓb₀ ⊍ ℓr₀ ⊍ ℓ∈AB ⊍ ℓ∈BA) where
  open SetoidA A
  open SetoidB B
  field
    from-∈-conv : (R : ReprAB) (a : A₀) (b : B₀) → (b , a) ∈BA conv R → (a , b) ∈AB R
    to-∈-conv : (R : ReprAB) (a : A₀) (b : B₀) → (a , b) ∈AB R → (b , a) ∈BA conv R
```

21.6 Relation.Binary.ElemRel.Comp

```agda
record ElemRelComp
    {ℓa₀ ℓa₁ : Level} (A : Setoid ℓa₀ ℓa₁)
```

```
    {ℓb₀ ℓb₁ : Level} (B : Setoid ℓb₀ ℓb₁)
    {ℓc₀ ℓc₁ : Level} (C : Setoid ℓc₀ ℓc₁)
    {ℓq₀ ℓq∈ : Level} {ReprAB : Set ℓq₀} (_∈AB_ : ⌊ A ⌋ × ⌊ B ⌋ → ReprAB → Set ℓq∈)
    {ℓr₀ ℓr∈ : Level} {ReprBC : Set ℓr₀} (_∈BC_ : ⌊ B ⌋ × ⌊ C ⌋ → ReprBC → Set ℓr∈)
    {ℓs₀ ℓs∈ : Level} {ReprAC : Set ℓs₀} (_∈AC_ : ⌊ A ⌋ × ⌊ C ⌋ → ReprAC → Set ℓs∈)
    (comp : ReprAB → ReprBC → ReprAC) : Set (ℓa₀ ⊍ ℓb₀ ⊍ ℓc₀ ⊍ ℓq₀ ⊍ ℓr₀ ⊍ ℓs₀ ⊍ ℓq∈ ⊍ ℓr∈ ⊍ ℓs∈) where
  open SetoidA A
  open SetoidB B
  open SetoidC C
  field
    from-∈-comp : (R : ReprAB) (S : ReprBC) (a : A₀) (c : C₀)
                → (a, c) ∈AC comp R S → Σ [ b : B₀ ] (a, b) ∈AB R × (b, c) ∈BC S
    to-∈-comp   : (R : ReprAB) (S : ReprBC) (a : A₀) (b : B₀) (c : C₀)
                → (a, b) ∈AB R → (b, c) ∈BC S → (a, c) ∈AC comp R S
```

21.7 Relation.Binary.ElemRel.Id

```
record ElemRelId
    {ℓa₀ ℓa₁ : Level} (A : Setoid ℓa₀ ℓa₁)
    {ℓs₀ : Level} {ReprAA : Set ℓs₀} {ℓ∈AA : Level} (_∈AA_ : ⌊ A ⌋ × ⌊ A ⌋ → ReprAA → Set ℓ∈AA)
    (Id : ReprAA) : Set (ℓa₀ ⊍ ℓa₁ ⊍ ℓ∈AA) where
  open SetoidA A
  field
    from-∈-Id : (a₁ a₂ : A₀) → (a₁, a₂) ∈AA Id → a₁ ≈A a₂
    to-∈-Id : (a₁ a₂ : A₀) → a₁ ≈A a₂ → (a₁, a₂) ∈AA Id
```

21.8 Relation.Binary.ElemRel.SubId

```
record ElemRelSubId
    {ℓa₀ ℓa₁ : Level} (A : Setoid ℓa₀ ℓa₁)
    {ℓr₀ : Level} {ReprA : Set ℓr₀} {ℓ∈A : Level} (_∈A_ : ⌊ A ⌋ → ReprA → Set ℓ∈A)
    {ℓs₀ : Level} {ReprAA : Set ℓs₀} {ℓ∈AA : Level} (_∈AA_ : ⌊ A ⌋ × ⌊ A ⌋ → ReprAA → Set ℓ∈AA)
    (SubId : ReprA → ReprAA) : Set (ℓa₀ ⊍ ℓa₁ ⊍ ℓr₀ ⊍ ℓ∈A ⊍ ℓ∈AA) where
  open SetoidA A
  field
    from-∈-SubId : (s : ReprA) (a₁ a₂ : A₀) → (a₁, a₂) ∈AA SubId s → a₁ ∈A s × a₁ ≈A a₂
    to-∈-SubId : (s : ReprA) (a₁ a₂ : A₀) → a₁ ∈A s → a₁ ≈A a₂ → (a₁, a₂) ∈AA SubId s
```

21.9 Relation.Binary.ElemRel.Conv2

```
open ElemRel
module ElemRel-Conv2
    {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
    {ℓb₀ ℓb₁ : Level} {B : Setoid ℓb₀ ℓb₁}
    {ℓq₀ ℓq₁ ℓq₂ ℓq∈ : Level} (AB : ElemRel A B ℓq₀ ℓq₁ ℓq₂ ℓq∈)
    {ℓr₀ ℓr₁ ℓr₂ ℓr∈ : Level} (BA : ElemRel B A ℓr₀ ℓr₁ ℓr₂ ℓr∈)
    {convAB : RelRepr₀ AB → RelRepr₀ BA}
    (EC-AB : ElemRelConv A B (_∈_ AB) (_∈_ BA) convAB)
    {convBA : RelRepr₀ BA → RelRepr₀ AB}
    (EC-BA : ElemRelConv B A (_∈_ BA) (_∈_ AB) convBA)
  where
```

```
  private
    module AB = ElemRel AB
    module BA = ElemRel BA
  open ElemRelConv EC-AB renaming (from-∈-conv to from-∈-convAB; to-∈-conv to to-∈-convAB)
  open ElemRelConv EC-BA renaming (from-∈-conv to from-∈-convBA; to-∈-conv to to-∈-convBA)

  conv-monotone-⇒ : {R₁ R₂ : AB.RelRepr₀} → R₁ AB.⇒ R₂ → convAB R₁ BA.⇒ convAB R₂
  conv-monotone-⇒ {R₁} {R₂} R₁⇒R₂ (b, a) bR₁˘a =
    to-∈-convAB R₂ a b (R₁⇒R₂ (a, b) (from-∈-convAB R₁ a b bR₁˘a))
  conv-monotone : {R₁ R₂ : AB.RelRepr₀} → R₁ AB.⊆ R₂ → convAB R₁ BA.⊆ convAB R₂
  conv-monotone R₁⊆R₂ = BA.⊆-from-⇒ (conv-monotone-⇒ (AB.⊆-to-⇒ R₁⊆R₂))
  conv-cong :    {R₁ R₂ : AB.RelRepr₀} → R₁ AB.≈ R₂ → convAB R₁ BA.≈ convAB R₂
  conv-cong R₁≈R₂ with AB.≈-to-⇔ R₁≈R₂
  ... | R₁⇒R₂, R₂⇒R₁ = BA.≈-from-⇔ (conv-monotone-⇒ R₁⇒R₂, conv-monotone-⇒ R₂⇒R₁)

  conv-conv-⇒ : {R : AB.RelRepr₀} → convBA (convAB R) AB.⇒ R
  conv-conv-⇒ {R} (a, b) aR˘˘b = from-∈-convAB R a b (from-∈-convBA (convAB R) b a aR˘˘b)
  conv-conv-⊆  : {R : AB.RelRepr₀} → convBA (convAB R) AB.⊆ R
  conv-conv-⊆ {R} = AB.⊆-from-⇒ conv-conv-⇒
  conv-conv-⟸ :   {R : AB.RelRepr₀} → R AB.⇒ convBA (convAB R)
  conv-conv-⟸ {R} (a, b) aRb = to-∈-convBA (convAB R) b a (to-∈-convAB R a b aRb)
  conv-conv-⊇     :   {R : AB.RelRepr₀} → R AB.⊆ convBA (convAB R)
  conv-conv-⊇ {R} = AB.⊆-from-⇒ conv-conv-⟸
  conv-conv :    {R : AB.RelRepr₀} → convBA (convAB R) AB.≈ R
  conv-conv {R} = AB.≈-from-⇔ (conv-conv-⇒, conv-conv-⟸)
```

21.10 Relation.Binary.ElemRel.Comp3

```
open ElemRel
module ElemRel-Comp3
    {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
    {ℓb₀ ℓb₁ : Level} {B : Setoid ℓb₀ ℓb₁}
    {ℓc₀ ℓc₁ : Level} {C : Setoid ℓc₀ ℓc₁}
    {ℓq₀ ℓq₁ ℓq₂ ℓq∈ : Level} (AB : ElemRel A B ℓq₀ ℓq₁ ℓq₂ ℓq∈)
    {ℓr₀ ℓr₁ ℓr₂ ℓr∈ : Level} (BC : ElemRel B C ℓr₀ ℓr₁ ℓr₂ ℓr∈)
    {ℓs₀ ℓs₁ ℓs₂ ℓs∈ : Level} (AC : ElemRel A C ℓs₀ ℓs₁ ℓs₂ ℓs∈)
    {comp : RelRepr₀ AB → RelRepr₀ BC → RelRepr₀ AC}
    (EC : ElemRelComp A B C (_∈_ AB) (_∈_ BC) (_∈_ AC) comp)
  where
  private
    module AB = ElemRel AB
    module BC = ElemRel BC
    module AC = ElemRel AC
  open ElemRelComp EC
  comp-monotone-⇒ : {R₁ R₂ : AB.RelRepr₀} {S₁ S₂ : BC.RelRepr₀}
                    → R₁ AB.⇒ R₂ → S₁ BC.⇒ S₂ → comp R₁ S₁ AC.⇒ comp R₂ S₂
  comp-monotone-⇒ {R₁} {R₂} {S₁} {S₂} R₁⇒R₂ S₁⇒S₂ (a, c) aR₁S₁c
    with from-∈-comp R₁ S₁ a c aR₁S₁c
  ... | b, aR₁b, bS₁c = to-∈-comp R₂ S₂ a b c (R₁⇒R₂ (a, b) aR₁b) (S₁⇒S₂ (b, c) bS₁c)
  comp-monotone : {R₁ R₂ : AB.RelRepr₀} {S₁ S₂ : BC.RelRepr₀}
                → R₁ AB.⊆ R₂ → S₁ BC.⊆ S₂ → comp R₁ S₁ AC.⊆ comp R₂ S₂
  comp-monotone R₁⊆R₂ S₁⊆S₂ = AC.⊆-from-⇒ (comp-monotone-⇒ (AB.⊆-to-⇒ R₁⊆R₂)
                                                          (BC.⊆-to-⇒ S₁⊆S₂))
```

```
comp-cong : {R₁ R₂ : AB.RelRepr₀} {S₁ S₂ : BC.RelRepr₀}
          → R₁ AB.≈ R₂ → S₁ BC.≈ S₂ → comp R₁ S₁ AC.≈ comp R₂ S₂
comp-cong R₁≈R₂ S₁≈S₂ with AB.≈-to-⇔ R₁≈R₂ | BC.≈-to-⇔ S₁≈S₂
... | R₁⇒R₂, R₂⇒R₁ | S₁⇒S₂, S₂⇒S₁ = AC.≈-from-⇔ (comp-monotone-⇒ R₁⇒R₂ S₁⇒S₂
                                                  , comp-monotone-⇒ R₂⇒R₁ S₂⇒S₁)
```

21.11 Relation.Binary.ElemRel.Involution

[WK: *ElemRel-Involution has 30 Level parameters, by involving 3 objects and 6 homsets.* **]**

```
open ElemRel
module ElemRel-Involution
    {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
    {ℓb₀ ℓb₁ : Level} {B : Setoid ℓb₀ ℓb₁}
    {ℓc₀ ℓc₁ : Level} {C : Setoid ℓc₀ ℓc₁}
    {ℓq₀ ℓq₁ ℓq₂ ℓq∈ : Level} (AB : ElemRel A B ℓq₀ ℓq₁ ℓq₂ ℓq∈)
    {ℓr₀ ℓr₁ ℓr₂ ℓr∈ : Level} (BC : ElemRel B C ℓr₀ ℓr₁ ℓr₂ ℓr∈)
    {ℓs₀ ℓs₁ ℓs₂ ℓs∈ : Level} (AC : ElemRel A C ℓs₀ ℓs₁ ℓs₂ ℓs∈)
    {ℓt₀ ℓt₁ ℓt₂ ℓt∈ : Level} (BA : ElemRel B A ℓt₀ ℓt₁ ℓt₂ ℓt∈)
    {ℓu₀ ℓu₁ ℓu₂ ℓu∈ : Level} (CB : ElemRel C B ℓu₀ ℓu₁ ℓu₂ ℓu∈)
    {ℓv₀ ℓv₁ ℓv₂ ℓv∈ : Level} (CA : ElemRel C A ℓv₀ ℓv₁ ℓv₂ ℓv∈)
    {convAB : RelRepr₀ AB → RelRepr₀ BA}
    (EC-AB : ElemRelConv A B (_∈_ AB) (_∈_ BA) convAB)
    {convBC : RelRepr₀ BC → RelRepr₀ CB}
    (EC-BC : ElemRelConv B C (_∈_ BC) (_∈_ CB) convBC)
    {convAC : RelRepr₀ AC → RelRepr₀ CA}
    (EC-AC : ElemRelConv A C (_∈_ AC) (_∈_ CA) convAC)
    {comp : RelRepr₀ AB → RelRepr₀ BC → RelRepr₀ AC}
    {comp′ : RelRepr₀ CB → RelRepr₀ BA → RelRepr₀ CA}
    (EC : ElemRelComp A B C (_∈_ AB) (_∈_ BC) (_∈_ AC) comp)
    (EC′ : ElemRelComp C B A (_∈_ CB) (_∈_ BA) (_∈_ CA) comp′)
  where
    private
      module AB = ElemRel AB
      module BC = ElemRel BC
      module AC = ElemRel AC
      module CA = ElemRel CA
    open ElemRelConv EC-AB renaming (from-∈-conv to from-∈-convAB; to-∈-conv to to-∈-convAB)
    open ElemRelConv EC-BC renaming (from-∈-conv to from-∈-convBC; to-∈-conv to to-∈-convBC)
    open ElemRelConv EC-AC renaming (from-∈-conv to from-∈-convAC; to-∈-conv to to-∈-convAC)
    open ElemRelComp EC
    open ElemRelComp EC′ renaming (from-∈-comp to from-∈-comp′; to-∈-comp to to-∈-comp′)

    conv-involution-⇒ : {R : AB.RelRepr₀} {S : BC.RelRepr₀}
                      → convAC (comp R S) CA.⇒ comp′ (convBC S) (convAB R)
    conv-involution-⇒ {R} {S} (c, a) cRS-˘a
      with from-∈-comp R S a c (from-∈-convAC (comp R S) a c cRS-˘a)
    ... | b, aRb, bSc = to-∈-comp′ (convBC S) (convAB R) c b a (to-∈-convBC S b c bSc)
                                                          (to-∈-convAB R a b aRb)

    conv-involution-⊆ : {R : AB.RelRepr₀} {S : BC.RelRepr₀}
                      → convAC (comp R S) CA.⊆ comp′ (convBC S) (convAB R)
    conv-involution-⊆ = CA.⊆-from-⇒ conv-involution-⇒

    conv-involution-⇐ : {R : AB.RelRepr₀} {S : BC.RelRepr₀}
                      → comp′ (convBC S) (convAB R) CA.⇒ convAC (comp R S)
    conv-involution-⇐ {R} {S} (c, a) cS˘R˘a with from-∈-comp′ (convBC S) (convAB R) c a cS˘R˘a
    ... | b, cS˘b, bR˘a = to-∈-convAC (comp R S) a c (to-∈-comp R S a b c (from-∈-convAB R a b bR˘a)
                                                                  (from-∈-convBC S b c cS˘b))
```

```
conv-involution-⊇  : {R : AB.RelRepr0} {S : BC.RelRepr0}
                   → comp′ (convBC S) (convAB R) CA.⊆ convAC (comp R S)
conv-involution-⊇  = CA.⊆-from-⇒ conv-involution-⇐
conv-involution : {R : AB.RelRepr0} {S : BC.RelRepr0}
                → convAC (comp R S) CA.≈ comp′ (convBC S) (convAB R)
conv-involution  = CA.≈-from-⇔ (conv-involution-⇒, conv-involution-⇐)
```

21.12 Relation.Binary.ElemRel.CompAssoc

[WK: *ElemRel-CompAssoc* has *32* *Level* parameters, by involving 4 objects and 6 homsets.]

```
open ElemRel
module ElemRel-CompAssoc
    {ℓa0 ℓa1 : Level} {A : Setoid ℓa0 ℓa1}
    {ℓb0 ℓb1 : Level} {B : Setoid ℓb0 ℓb1}
    {ℓc0 ℓc1 : Level} {C : Setoid ℓc0 ℓc1}
    {ℓd0 ℓd1 : Level} {D : Setoid ℓd0 ℓd1}
    {ℓq0 ℓq1 ℓq2 ℓq∈ : Level} (AB : ElemRel A B ℓq0 ℓq1 ℓq2 ℓq∈)
    {ℓr0 ℓr1 ℓr2 ℓr∈ : Level} (BC : ElemRel B C ℓr0 ℓr1 ℓr2 ℓr∈)
    {ℓs0 ℓs1 ℓs2 ℓs∈ : Level} (AC : ElemRel A C ℓs0 ℓs1 ℓs2 ℓs∈)
    {ℓt0 ℓt1 ℓt2 ℓt∈ : Level} (BD : ElemRel B D ℓt0 ℓt1 ℓt2 ℓt∈)
    {ℓu0 ℓu1 ℓu2 ℓu∈ : Level} (CD : ElemRel C D ℓu0 ℓu1 ℓu2 ℓu∈)
    {ℓv0 ℓv1 ℓv2 ℓv∈ : Level} (AD : ElemRel A D ℓv0 ℓv1 ℓv2 ℓv∈)
    {compABC : RelRepr0 AB → RelRepr0 BC → RelRepr0 AC}
    {compACD : RelRepr0 AC → RelRepr0 CD → RelRepr0 AD}
    {compBCD : RelRepr0 BC → RelRepr0 CD → RelRepr0 BD}
    {compABD : RelRepr0 AB → RelRepr0 BD → RelRepr0 AD}
    (EC-ABC : ElemRelComp A B C (_∈_ AB) (_∈_ BC) (_∈_ AC) compABC)
    (EC-ACD : ElemRelComp A C D (_∈_ AC) (_∈_ CD) (_∈_ AD) compACD)
    (EC-BCD : ElemRelComp B C D (_∈_ BC) (_∈_ CD) (_∈_ BD) compBCD)
    (EC-ABD : ElemRelComp A B D (_∈_ AB) (_∈_ BD) (_∈_ AD) compABD)
  where
    private
      module AB = ElemRel AB
      module BC = ElemRel BC
      module CD = ElemRel CD
      module AD = ElemRel AD
    open ElemRelComp EC-ABC
      renaming (from-∈-comp to from-∈-compABC; to-∈-comp to to-∈-compABC)
    open ElemRelComp EC-ACD
      renaming (from-∈-comp to from-∈-compACD; to-∈-comp to to-∈-compACD)
    open ElemRelComp EC-BCD
      renaming (from-∈-comp to from-∈-compBCD; to-∈-comp to to-∈-compBCD)
    open ElemRelComp EC-ABD
      renaming (from-∈-comp to from-∈-compABD; to-∈-comp to to-∈-compABD)
    comp-assoc-⇒   : {Q : AB.RelRepr0} {R : BC.RelRepr0} {S : CD.RelRepr0}
                   → compACD (compABC Q R) S AD.⇒ compABD Q (compBCD R S)
    comp-assoc-⇒ {Q} {R} {S} (a, d) aQR-Sd with from-∈-compACD _ _ a d aQR-Sd
    ... | c, aQRc, cSd with from-∈-compABC _ _ a c aQRc
    ... | b, aQb, bRc = to-∈-compABD _ _ a b d aQb (to-∈-compBCD _ _ b c d bRc cSd)
    comp-assoc-⇐ : {Q : AB.RelRepr0} {R : BC.RelRepr0} {S : CD.RelRepr0}
                 → compABD Q (compBCD R S) AD.⇒ compACD (compABC Q R) S
    comp-assoc-⇐ {Q} {R} {S} (a, d) aQ-RSd with from-∈-compABD _ _ a d aQ-RSd
    ... | b, aQb, bRSd with from-∈-compBCD _ _ b d bRSd
    ... | c, bRc, cSd = to-∈-compACD _ _ a c d (to-∈-compABC _ _ a b c aQb bRc) cSd
    comp-assoc-⊆   : {Q : AB.RelRepr0} {R : BC.RelRepr0} {S : CD.RelRepr0}
```

```
              → compACD (compABC Q R) S AD.⊆ compABD Q (compBCD R S)
  comp-assoc-⊆ = AD.⊆-from-⇒ comp-assoc-⇒
  comp-assoc-⊇   : {Q : AB.RelRepr₀} {R : BC.RelRepr₀} {S : CD.RelRepr₀}
              → compABD Q (compBCD R S) AD.⊆ compACD (compABC Q R) S
  comp-assoc-⊇ = AD.⊆-from-⇒ comp-assoc-⇐
  comp-assoc : {Q : AB.RelRepr₀} {R : BC.RelRepr₀} {S : CD.RelRepr₀}
           → compACD (compABC Q R) S AD.≈ compABD Q (compBCD R S)
  comp-assoc = AD.≈-from-⇔ (comp-assoc-⇒, comp-assoc-⇐)
```

21.13 Relation.Binary.ElemRel.Comp3UnionL

```
open ElemRel
module ElemRel-Comp3UnionL
    {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
    {ℓb₀ ℓb₁ : Level} {B : Setoid ℓb₀ ℓb₁}
    {ℓc₀ ℓc₁ : Level} {C : Setoid ℓc₀ ℓc₁}
    {ℓq₀ ℓq₁ ℓq₂ ℓq∈ : Level} (AB : ElemRel A B ℓq₀ ℓq₁ ℓq₂ ℓq∈)
    {ℓr₀ ℓr₁ ℓr₂ ℓr∈ : Level} (BC : ElemRel B C ℓr₀ ℓr₁ ℓr₂ ℓr∈)
    {ℓs₀ ℓs₁ ℓs₂ ℓs∈ : Level} (AC : ElemRel A C ℓs₀ ℓs₁ ℓs₂ ℓs∈)
    {comp : RelRepr₀ AB → RelRepr₀ BC → RelRepr₀ AC}
    (EC : ElemRelComp A B C (_∈_ AB) (_∈_ BC) (_∈_ AC) comp)
    {unionAB : RelRepr₀ AB → RelRepr₀ AB → RelRepr₀ AB}
    {unionAC : RelRepr₀ AC → RelRepr₀ AC → RelRepr₀ AC}
    (EJ-AB : ElemSetJoin (A ×× B) (_∈_ AB) unionAB)
    (EJ-AC : ElemSetJoin (A ×× C) (_∈_ AC) unionAC)
  where
    private
      module AB = ElemRel AB
      module BC = ElemRel BC
      module AC = ElemRel AC
    open ElemRelComp EC
    open ElemSetJoin (A ×× B) AB._∈_ EJ-AB renaming
      (from-∈-union to from-∈-unionAB; to-∈-union to to-∈-unionAB)
    open ElemSetJoin (A ×× C) AC._∈_ EJ-AC renaming
      (from-∈-union to from-∈-unionAC; to-∈-union to to-∈-unionAC)
    union-comp-subdistrL-⇒ : {R₁ R₂ : AB.RelRepr₀} {S : BC.RelRepr₀}
                         → comp (unionAB R₁ R₂) S AC.⇒ unionAC (comp R₁ S) (comp R₂ S)
    union-comp-subdistrL-⇒ {R₁} {R₂} {S} (a, c) aR₁₂Sc with from-∈-comp _ _ a c aR₁₂Sc
    ... | b, aR₁₂b, bSc with from-∈-unionAB R₁ R₂ (a, b) aR₁₂b
    ... | inj₁ aR₁b = to-∈-unionAC _ _ (a, c) (inj₁ (to-∈-comp _ _ a b c aR₁b bSc))
    ... | inj₂ aR₂b = to-∈-unionAC _ _ (a, c) (inj₂ (to-∈-comp _ _ a b c aR₂b bSc))
    union-comp-subdistrL : {R₁ R₂ : AB.RelRepr₀} {S : BC.RelRepr₀}
                       → comp (unionAB R₁ R₂) S AC.⊆ unionAC (comp R₁ S) (comp R₂ S)
    union-comp-subdistrL = AC.⊆-from-⇒ union-comp-subdistrL-⇒
```

21.14 Relation.Binary.ElemRel.Comp3UnionR

```
open ElemRel
module ElemRel-Comp3UnionR
    {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
    {ℓb₀ ℓb₁ : Level} {B : Setoid ℓb₀ ℓb₁}
    {ℓc₀ ℓc₁ : Level} {C : Setoid ℓc₀ ℓc₁}
```

```
    {ℓq₀ ℓq₁ ℓq₂ ℓq∈ : Level} (AB : ElemRel A B ℓq₀ ℓq₁ ℓq₂ ℓq∈)
    {ℓr₀ ℓr₁ ℓr₂ ℓr∈ : Level} (BC : ElemRel B C ℓr₀ ℓr₁ ℓr₂ ℓr∈)
    {ℓs₀ ℓs₁ ℓs₂ ℓs∈ : Level} (AC : ElemRel A C ℓs₀ ℓs₁ ℓs₂ ℓs∈)
    {comp : RelRepr₀ AB → RelRepr₀ BC → RelRepr₀ AC}
    (EC : ElemRelComp A B C (_∈_ AB) (_∈_ BC) (_∈_ AC) comp)
    {unionBC : RelRepr₀ BC → RelRepr₀ BC → RelRepr₀ BC}
    {unionAC : RelRepr₀ AC → RelRepr₀ AC → RelRepr₀ AC}
    (EJ-BC : ElemSetJoin (B ×× C) (_∈_ BC) unionBC)
    (EJ-AC : ElemSetJoin (A ×× C) (_∈_ AC) unionAC)
  where
    private
      module AB = ElemRel AB
      module BC = ElemRel BC
      module AC = ElemRel AC
    open ElemRelComp EC
    open ElemSetJoin (B ×× C) BC._∈_ EJ-BC renaming
      (from-∈-union to from-∈-unionBC; to-∈-union to to-∈-unionBC)
    open ElemSetJoin (A ×× C) AC._∈_ EJ-AC renaming
      (from-∈-union to from-∈-unionAC; to-∈-union to to-∈-unionAC)

    union-comp-subdistrR-⇒ : {R : AB.RelRepr₀} {S₁ S₂ : BC.RelRepr₀}
                           → comp R (unionBC S₁ S₂) AC.⇒ unionAC (comp R S₁) (comp R S₂)
    union-comp-subdistrR-⇒ {R} {S₁} {S₂} (a, c) aRS₁₂c with from-∈-comp _ _ a c aRS₁₂c
    ... | b, aRb, bS₁₂c with from-∈-unionBC S₁ S₂ (b, c) bS₁₂c
    ... | inj₁ bS₁c = to-∈-unionAC _ _ (a, c) (inj₁ (to-∈-comp _ _ a b c aRb bS₁c))
    ... | inj₂ bS₂c = to-∈-unionAC _ _ (a, c) (inj₂ (to-∈-comp _ _ a b c aRb bS₂c))

    union-comp-subdistrR : {R : AB.RelRepr₀} {S₁ S₂ : BC.RelRepr₀}
                         → comp R (unionBC S₁ S₂) AC.⊆ unionAC (comp R S₁) (comp R S₂)
    union-comp-subdistrR = AC.⊆-from-⇒ union-comp-subdistrR-⇒
```

21.15 Relation.Binary.ElemRel.Conv2-IdL

```
open ElemRel
module ElemRel-Conv2-IdL
    {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
    {ℓb₀ ℓb₁ : Level} {B : Setoid ℓb₀ ℓb₁}
    {ℓq₀ ℓq₁ ℓq₂ ℓq∈ : Level} (AB : ElemRel A B ℓq₀ ℓq₁ ℓq₂ ℓq∈)
    {ℓr₀ ℓr₁ ℓr₂ ℓr∈ : Level} (BA : ElemRel B A ℓr₀ ℓr₁ ℓr₂ ℓr∈)
    {ℓs₀ ℓs₁ ℓs₂ ℓs∈ : Level} (AA : ElemRel A A ℓs₀ ℓs₁ ℓs₂ ℓs∈)
    {convAB : RelRepr₀ AB → RelRepr₀ BA}
    (EC-AB : ElemRelConv A B (_∈_ AB) (_∈_ BA) convAB)
    {comp : RelRepr₀ AB → RelRepr₀ BA → RelRepr₀ AA}
    (EC : ElemRelComp A B A (_∈_ AB) (_∈_ BA) (_∈_ AA) comp)
    {Id : RelRepr₀ AA} (EId : ElemRelId A (_∈_ AA) Id)
  where
    open SetoidA A
    open SetoidB B
    private
      module AB = ElemRel AB
      module BA = ElemRel BA
      module AA = ElemRel AA
    open ElemRelConv EC-AB renaming (from-∈-conv to from-∈-convAB; to-∈-conv to to-∈-convAB)
    open ElemRelComp EC
    open ElemRelId EId

    injective-to-⇒Id : {R : AB.RelRepr₀} → AB.injective R → comp R (convAB R) AA.⇒ Id
    injective-to-⇒Id {R} injR (a₁, a₂) a₁RR˘a₂ with from-∈-comp _ _ a₁ a₂ a₁RR˘a₂
```

```
  ... | b , a₁Rb , bR˘a₂ = let a₁≈a₂ = injR a₁ a₂ b a₁Rb (from-∈-convAB R a₂ b bR˘a₂)
    in to-∈-Id a₁ a₂ a₁≈a₂

  injective-to-⊆Id : {R : AB.RelRepr₀} → AB.injective R → comp R (convAB R) AA.⊆ Id
  injective-to-⊆Id injR = AA.⊆-from-⇒ (injective-to-⇒Id injR)

  injective-from-⇒Id : {R : AB.RelRepr₀} → comp R (convAB R) AA.⇒ Id → AB.injective R
  injective-from-⇒Id RR˘⇒I a₁ a₂ b a₁Rb a₂Rb = from-∈-Id a₁ a₂
    (RR˘⇒I (a₁ , a₂) (to-∈-comp _ _ a₁ b a₂ a₁Rb (to-∈-convAB _ a₂ b a₂Rb)))

  injective-from-⊆Id : {R : AB.RelRepr₀} → comp R (convAB R) AA.⊆ Id → AB.injective R
  injective-from-⊆Id RR˘⊆I = injective-from-⇒Id (AA.⊆-to-⇒ RR˘⊆I)


  total-to-Id⇒ : {R : AB.RelRepr₀} → AB.total R → Id AA.⇒ comp R (convAB R)
  total-to-Id⇒ {R} totalR (a₁ , a₂) a₁Ia₂ with totalR a₁
  ... | b , a₁Rb = to-∈-comp R _ a₁ b a₂ a₁Rb
                     (to-∈-convAB R a₂ b (AB.∈-subst₁₁ R (from-∈-Id a₁ a₂ a₁Ia₂) a₁Rb))

  total-to-Id⊆ : {R : AB.RelRepr₀} → AB.total R → Id AA.⊆ comp R (convAB R)
  total-to-Id⊆ totalR = AA.⊆-from-⇒ (total-to-Id⇒ totalR)

  total-from-Id⇒ : {R : AB.RelRepr₀} → Id AA.⇒ comp R (convAB R) → AB.total R
  total-from-Id⇒ I⇒RR˘ a with from-∈-comp _ _ a a (I⇒RR˘ (a , a) (to-∈-Id a a ≈A-refl))
  ... | b , aRb , bR˘a = b , aRb

  total-from-Id⊆ : {R : AB.RelRepr₀} → Id AA.⊆ comp R (convAB R) → AB.total R
  total-from-Id⊆ I⊆RR˘ = total-from-Id⇒ (AA.⊆-to-⇒ I⊆RR˘)
```

21.16 Relation.Binary.ElemRel.Conv2-IdR

```
open ElemRel

module ElemRel-Conv2-IdR
    {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
    {ℓb₀ ℓb₁ : Level} {B : Setoid ℓb₀ ℓb₁}
    {ℓq₀ ℓq₁ ℓq₂ ℓq∈ : Level} (AB : ElemRel A B ℓq₀ ℓq₁ ℓq₂ ℓq∈)
    {ℓr₀ ℓr₁ ℓr₂ ℓr∈ : Level} (BA : ElemRel B A ℓr₀ ℓr₁ ℓr₂ ℓr∈)
    {ℓs₀ ℓs₁ ℓs₂ ℓs∈ : Level} (BB : ElemRel B B ℓs₀ ℓs₁ ℓs₂ ℓs∈)
    {convAB : RelRepr₀ AB → RelRepr₀ BA}
    (EC-AB : ElemRelConv A B (_∈_ AB) (_∈_ BA) convAB)
    {comp : RelRepr₀ BA → RelRepr₀ AB → RelRepr₀ BB}
    (EC : ElemRelComp B A B (_∈_ BA) (_∈_ AB) (_∈_ BB) comp)
    {Id : RelRepr₀ BB} (EId : ElemRelId B (_∈_ BB) Id)
  where
  open SetoidA A
  open SetoidB B
  private
    module AB = ElemRel AB
    module BA = ElemRel BA
    module BB = ElemRel BB
  open ElemRelConv EC-AB
  open ElemRelComp EC
  open ElemRelId EId


  unival-to-⇒Id : {R : AB.RelRepr₀} → AB.univalent R → comp (convAB R) R BB.⇒ Id
  unival-to-⇒Id {R} univalR (b₁ , b₂) b₁R˘Rb₂ with from-∈-comp _ _ b₁ b₂ b₁R˘Rb₂
  ... | a , b₁R˘a , aRb₂ = let b₁≈b₂ = univalR a b₁ b₂ (from-∈-conv R a b₁ b₁R˘a) aRb₂
    in to-∈-Id b₁ b₂ b₁≈b₂

  unival-to-⊆Id : {R : AB.RelRepr₀} → AB.univalent R → comp (convAB R) R BB.⊆ Id
  unival-to-⊆Id univalR = BB.⊆-from-⇒ (unival-to-⇒Id univalR)
```

```
unival-from-⇒Id : {R : AB.RelRepr₀} → comp (convAB R) R BB.⇒ Id → AB.univalent R
unival-from-⇒Id R˘R⇒I a b₁ b₂ aRb₁ aRb₂ = from-∈-Id b₁ b₂
  (R˘R⇒I (b₁, b₂) (to-∈-comp _ _ b₁ a b₂ (to-∈-conv _ a b₁ aRb₁) aRb₂))
unival-from-⊆Id : {R : AB.RelRepr₀} → comp (convAB R) R BB.⊆ Id → AB.univalent R
unival-from-⊆Id R˘R⊆I = unival-from-⇒Id (BB.⊆-to-⇒ R˘R⊆I)


surjective-to-Id⇒ : {R : AB.RelRepr₀} → AB.surjective R → Id BB.⇒ comp (convAB R) R
surjective-to-Id⇒ {R} surjR (b₁, b₂) b₁Ib₂ with surjR b₁
... | a, aRb₁ = to-∈-comp _ R b₁ a b₂ (to-∈-conv R a b₁ aRb₁)
                                          (AB.∈-subst₁₂ R (from-∈-Id b₁ b₂ b₁Ib₂) aRb₁)
surjective-to-Id⊆ : {R : AB.RelRepr₀} → AB.surjective R → Id BB.⊆ comp (convAB R) R
surjective-to-Id⊆ surjR = BB.⊆-from-⇒ (surjective-to-Id⇒ surjR)
surjective-from-Id⇒ : {R : AB.RelRepr₀} → Id BB.⇒ comp (convAB R) R → AB.surjective R
surjective-from-Id⇒ I⇒R˘R b with from-∈-comp _ _ b b (I⇒R˘R (b, b) (to-∈-Id b b ≈B-refl))
... | a, bR˘a, aRb = a, aRb
surjective-from-Id⊆ : {R : AB.RelRepr₀} → Id BB.⊆ comp (convAB R) R → AB.surjective R
surjective-from-Id⊆ I⊆R˘R = surjective-from-Id⇒ (BB.⊆-to-⇒ I⊆R˘R)
```

21.17 Relation.Binary.ElemRel.LeftSubId

```
open ElemRel
module ElemRel-LeftSubId
    {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
    {ℓb₀ ℓb₁ : Level} {B : Setoid ℓb₀ ℓb₁}
    {ℓq₀ ℓq₁ ℓq₂ ℓq∈ : Level} (ℙA : ElemSet A ℓq₀ ℓq₁ ℓq₂ ℓq∈)
    {ℓr₀ ℓr₁ ℓr₂ ℓr∈ : Level} (AA : ElemRel A A ℓr₀ ℓr₁ ℓr₂ ℓr∈)
    {ℓu₀ ℓu₁ ℓu₂ ℓu∈ : Level} (AB : ElemRel A B ℓu₀ ℓu₁ ℓu₂ ℓu∈)
    {comp : RelRepr₀ AA → RelRepr₀ AB → RelRepr₀ AB}
    (EC : ElemRelComp A A B (_∈_ AA) (_∈_ AB) (_∈_ AB) comp)
    {SubId : ElemSet.SetRepr₀ ℙA → RelRepr₀ AA}
    (ES : ElemRelSubId A (ElemSet._∈_ ℙA) (_∈_ AA) SubId)
  where
    private
      module ℙA = ElemSet ℙA
      module AA = ElemRel AA
      module AB = ElemRel AB
    open SetoidA A
    open ElemRelComp EC
    open ElemRelSubId ES
    leftId-⇒   : (as : ℙA.SetRepr₀) {R : AB.RelRepr₀} → comp (SubId as) R AB.⇒ R
    leftId-⇒ as {R} (a₁, b) a₁b∈IR with from-∈-comp (SubId as) R a₁ b a₁b∈IR
    ... | a₂, a₁Ia₂, a₂Rb with from-∈-SubId as a₁ a₂ a₁Ia₂
    ... | a₁∈s, a₁≈a₂ = AB.∈-subst₁₁ R (≈A-sym a₁≈a₂) a₂Rb
    leftId-⊆   : (as : ℙA.SetRepr₀) {R : AB.RelRepr₀} → comp (SubId as) R AB.⊆ R
    leftId-⊆ as = AB.⊆-from-⇒ (leftId-⇒ as)
    leftId-⟸ : (as : ℙA.SetRepr₀) → ℙA.isUniversal as
               → {R : AB.RelRepr₀} → R AB.⇒ comp (SubId as) R
    leftId-⟸ as isTop {R} (a, b) aRb = to-∈-comp (SubId as) R a a b
                                            (to-∈-SubId as a a (isTop a) ≈A-refl) aRb
    leftId-⊇   : (as : ℙA.SetRepr₀) → ℙA.isUniversal as
                 → {R : AB.RelRepr₀} → R AB.⊆ comp (SubId as) R
    leftId-⊇ as isTop = AB.⊆-from-⇒ (leftId-⟸ as isTop)
    leftId :       (as : ℙA.SetRepr₀) → ℙA.isUniversal as
```

```
    →       {R : AB.RelRepr₀} → comp (SubId as) R AB.≈ R
  leftId as isTop = AB.≈-from-⇔ (leftId-⇒ as, leftId-⇐ as isTop)
```

21.18 Relation.Binary.ElemRel.RightSubId

```
open ElemRel
module ElemRel-RightSubId
    {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
    {ℓb₀ ℓb₁ : Level} {B : Setoid ℓb₀ ℓb₁}
    {ℓq₀ ℓq₁ ℓq₂ ℓq∈ : Level} (ℙB : ElemSet B ℓq₀ ℓq₁ ℓq₂ ℓq∈)
    {ℓr₀ ℓr₁ ℓr₂ ℓr∈ : Level} (BB : ElemRel B B ℓr₀ ℓr₁ ℓr₂ ℓr∈)
    {ℓu₀ ℓu₁ ℓu₂ ℓu∈ : Level} (AB : ElemRel A B ℓu₀ ℓu₁ ℓu₂ ℓu∈)
    {comp : RelRepr₀ AB → RelRepr₀ BB → RelRepr₀ AB}
    (EC : ElemRelComp A B B (_∈_ AB) (_∈_ BB) (_∈_ AB) comp)
    {SubId : ElemSet.SetRepr₀ ℙB → RelRepr₀ BB}
    (ES : ElemRelSubId B (ElemSet._∈_ ℙB) (_∈_ BB) SubId)
  where
    private
      module ℙB = ElemSet ℙB
      module BB = ElemRel BB
      module AB = ElemRel AB
    open SetoidB B
    open ElemRelComp EC
    open ElemRelSubId ES

    rightId-⇒   : (bs : ℙB.SetRepr₀) {R : AB.RelRepr₀} → comp R (SubId bs) AB.⇒ R
    rightId-⇒ bs {R} (a, b₂) aRIb₂ with from-∈-comp R (SubId bs) a b₂ aRIb₂
    ... | b₁, aRb₁, b₁Ib₂ with from-∈-SubId bs b₁ b₂ b₁Ib₂
    ... | b₁∈s, b₁≈b₂ = AB.∈-subst₁₂ R b₁≈b₂ aRb₁

    rightId-⊆   : (bs : ℙB.SetRepr₀) {R : AB.RelRepr₀} → comp R (SubId bs) AB.⊆ R
    rightId-⊆ bs = AB.⊆-from-⇒ (rightId-⇒ bs)

    rightId-⇐ : (bs : ℙB.SetRepr₀) → ℙB.isUniversal bs
               → {R : AB.RelRepr₀} → R AB.⇒ comp R (SubId bs)
    rightId-⇐ bs isTop {R} (a, b) aRb = to-∈-comp R (SubId bs) a b b aRb
                                          (to-∈-SubId bs b b (isTop b) ≈B-refl)

    rightId-⊇   : (bs : ℙB.SetRepr₀) → ℙB.isUniversal bs
                 → {R : AB.RelRepr₀} → R AB.⊆ comp R (SubId bs)
    rightId-⊇ bs isTop = AB.⊆-from-⇒ (rightId-⇐ bs isTop)

    rightId :    (bs : ℙB.SetRepr₀) → ℙB.isUniversal bs
           →    {R : AB.RelRepr₀} → comp R (SubId bs) AB.≈ R
    rightId bs isTop = AB.≈-from-⇔ (rightId-⇒ bs, rightId-⇐ bs isTop)
```

21.19 Relation.Binary.ElemRel.Dedekind

```
open ElemRel
module ElemRel-Dedekind
    {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
    {ℓb₀ ℓb₁ : Level} {B : Setoid ℓb₀ ℓb₁}
    {ℓc₀ ℓc₁ : Level} {C : Setoid ℓc₀ ℓc₁}
    {ℓq₀ ℓq₁ ℓq₂ ℓq∈ : Level} (AB : ElemRel A B ℓq₀ ℓq₁ ℓq₂ ℓq∈)
    {ℓr₀ ℓr₁ ℓr₂ ℓr∈ : Level} (BC : ElemRel B C ℓr₀ ℓr₁ ℓr₂ ℓr∈)
    {ℓs₀ ℓs₁ ℓs₂ ℓs∈ : Level} (AC : ElemRel A C ℓs₀ ℓs₁ ℓs₂ ℓs∈)
    {ℓt₀ ℓt₁ ℓt₂ ℓt∈ : Level} (BA : ElemRel B A ℓt₀ ℓt₁ ℓt₂ ℓt∈)
```

```
    {ℓu₀ ℓu₁ ℓu₂ ℓu∈ : Level} (CB : ElemRel C B ℓu₀ ℓu₁ ℓu₂ ℓu∈)
    {convAB : RelRepr₀ AB → RelRepr₀ BA}
    {convBC : RelRepr₀ BC → RelRepr₀ CB}
    (EConv-AB : ElemRelConv A B (_∈_ AB) (_∈_ BA) convAB)
    (EConv-BC : ElemRelConv B C (_∈_ BC) (_∈_ CB) convBC)
    {compABC : RelRepr₀ AB → RelRepr₀ BC → RelRepr₀ AC}
    {compACB : RelRepr₀ AC → RelRepr₀ CB → RelRepr₀ AB}
    {compBAC : RelRepr₀ BA → RelRepr₀ AC → RelRepr₀ BC}
    (EComp-ABC : ElemRelComp A B C (_∈_ AB) (_∈_ BC) (_∈_ AC) compABC)
    (EComp-ACB : ElemRelComp A C B (_∈_ AC) (_∈_ CB) (_∈_ AB) compACB)
    (EComp-BAC : ElemRelComp B A C (_∈_ BA) (_∈_ AC) (_∈_ BC) compBAC)
    {meetAB : RelRepr₀ AB → RelRepr₀ AB → RelRepr₀ AB}
    {meetBC : RelRepr₀ BC → RelRepr₀ BC → RelRepr₀ BC}
    {meetAC : RelRepr₀ AC → RelRepr₀ AC → RelRepr₀ AC}
    (EM-AB : ElemSetMeet (A ×× B) (_∈_ AB) meetAB)
    (EM-BC : ElemSetMeet (B ×× C) (_∈_ BC) meetBC)
    (EM-AC : ElemSetMeet (A ×× C) (_∈_ AC) meetAC)
  where
    open SetoidA A
    open SetoidB B
    open SetoidB C
    private
      module AB where
        open ElemSetMeet (A ×× B) (_∈_ AB) EM-AB public
        open ElemRel AB public
        open ElemRelConv EConv-AB public
      module BC where
        open ElemSetMeet (B ×× C) (_∈_ BC) EM-BC public
        open ElemRel BC public
        open ElemRelConv EConv-BC public
      module AC where
        open ElemSetMeet (A ×× C) (_∈_ AC) EM-AC public
        open ElemRel AC public
      module ABC = ElemRelComp EComp-ABC
      module ACB = ElemRelComp EComp-ACB
      module BAC = ElemRelComp EComp-BAC


    Dedekind-⇒ : {Q : AB.RelRepr₀} {R : BC.RelRepr₀} {S : AC.RelRepr₀}
                 → meetAC (compABC Q R) S AC.⇒ compABC (meetAB Q (compACB S (convBC R)))
                                                        (meetBC R (compBAC (convAB Q) S))
    Dedekind-⇒ {S} {Q} {R} (a, c) aQR∩Sc with AC.from-∈-intersection _ _ (a, c) aQR∩Sc
    ... | aQRc, aSc with ABC.from-∈-comp _ _ a c aQRc
    ... | b, aQb, bRc = ABC.to-∈-comp _ _ a b c
                 (AB.to-∈-intersection _ _ (a, b) aQb
                   (ACB.to-∈-comp _ _ a c b aSc (BC.to-∈-conv _ b c bRc)))
                 (BC.to-∈-intersection _ _ (b, c) bRc
                   (BAC.to-∈-comp _ _ b a c (AB.to-∈-conv _ a b aQb) aSc))
    Dedekind : {Q : AB.RelRepr₀} {R : BC.RelRepr₀} {S : AC.RelRepr₀}
                 → meetAC       (compABC Q R) S
                 AC.⊆ compABC (meetAB Q (compACB S (convBC R)))
                              (meetBC R (compBAC (convAB Q) S))
    Dedekind = AC.⊆-from-⇒ Dedekind-⇒
```

21.20 Relation.Binary.ElemRel.Equivalence

The material here could be split if any of it is needed in different contexts.

```
open HomElemRel using () renaming
  (_∈_ to ∈1; RelRepr0 to RelRepr1)   -- only used in module parameters
module ElemRel-Refl
    {ℓa0 ℓa1 : Level} {A : Setoid ℓa0 ℓa1}
    {ℓq0 ℓq1 ℓq2 ℓq∈ : Level} (AA : HomElemRel A ℓq0 ℓq1 ℓq2 ℓq∈)
    {Id : RelRepr1 AA} (EId : ElemRelId A (∈1 AA) Id)
  where
    open SetoidA A
    open HomElemRel AA
    open ElemRelId EId

    coreflexive-to-⇒Id : {R : RelRepr0} → coreflexive R → R ⇒ Id
    coreflexive-to-⇒Id {R} corR (a1, a2) a1Ra2 = to-∈-Id a1 a2 (corR a1 a2 a1Ra2)
    coreflexive-to-⊆Id : {R : RelRepr0} → coreflexive R → R ⊆ Id
    coreflexive-to-⊆Id corR = ⊆-from-⇒ (coreflexive-to-⇒Id corR)
    coreflexive-from-⇒Id : {R : RelRepr0} → R ⇒ Id → coreflexive R
    coreflexive-from-⇒Id R⇒I a1 a2 a1Ra2 = from-∈-Id a1 a2 (R⇒I (a1, a2) a1Ra2)
    coreflexive-from-⊆Id : {R : RelRepr0} → R ⊆ Id → coreflexive R
    coreflexive-from-⊆Id R⊆I = coreflexive-from-⇒Id (⊆-to-⇒ R⊆I)

    reflexive-to-Id⇒ : {R : RelRepr0} → reflexive R → Id ⇒ R
    reflexive-to-Id⇒ {R} reflR (a1, a2) a1Ia2 = reflR a1 a2 (from-∈-Id a1 a2 a1Ia2)
    reflexive-to-Id⊆ : {R : RelRepr0} → reflexive R → Id ⊆ R
    reflexive-to-Id⊆ reflR = ⊆-from-⇒ (reflexive-to-Id⇒ reflR)
    reflexive-from-Id⇒ : {R : RelRepr0} → Id ⇒ R → reflexive R
    reflexive-from-Id⇒ I⇒R a1 a2 a1≈a2 = I⇒R (a1, a2) (to-∈-Id a1 a2 a1≈a2)
    reflexive-from-Id⊆ : {R : RelRepr0} → Id ⊆ R → reflexive R
    reflexive-from-Id⊆ I⊆R = reflexive-from-Id⇒ (⊆-to-⇒ I⊆R)


open HomElemRel using () renaming
  (_∈_ to ∈1; RelRepr0 to RelRepr1)   -- only used in module parameters
module ElemRel-PER
    {ℓa0 ℓa1 : Level} {A : Setoid ℓa0 ℓa1}
    {ℓq0 ℓq1 ℓq2 ℓq∈ : Level} (AA : HomElemRel A ℓq0 ℓq1 ℓq2 ℓq∈)
    {conv : RelRepr1 AA → RelRepr1 AA}
    (EConv : ElemRelConv A A (∈1 AA) (∈1 AA) conv)
    {comp : RelRepr1 AA → RelRepr1 AA → RelRepr1 AA}
    (EComp : ElemRelComp A A A (∈1 AA) (∈1 AA) (∈1 AA) comp)
  where
    open SetoidA A
    open HomElemRel AA
    open ElemRelConv EConv
    open ElemRelComp EComp

    symmetric-to-˘⇒ : {R : RelRepr0} → symmetric R → conv R ⇒ R
    symmetric-to-˘⇒ {R} symR (a1, a2) a1R˘a2 = symR a2 a1 (from-∈-conv R a2 a1 a1R˘a2)
    symmetric-to-˘⊆ : {R : RelRepr0} → symmetric R → conv R ⊆ R
    symmetric-to-˘⊆ symR = ⊆-from-⇒ (symmetric-to-˘⇒ symR)
    symmetric-to-⇒˘ : {R : RelRepr0} → symmetric R → R ⇒ conv R
    symmetric-to-⇒˘ {R} symR (a1, a2) a1Ra2 = to-∈-conv R a2 a1 (symR a1 a2 a1Ra2)
    symmetric-to-⊆˘ : {R : RelRepr0} → symmetric R → R ⊆ conv R
    symmetric-to-⊆˘ symR = ⊆-from-⇒ (symmetric-to-⇒˘ symR)
    symmetric-to-˘≈ : {R : RelRepr0} → symmetric R → conv R ≈ R
    symmetric-to-˘≈ symR = ≈-from-⇔ (symmetric-to-˘⇒ symR, symmetric-to-⇒˘ symR)
```

```
symmetric-from-˘⇒ : {R : RelRepr₀} → conv R ⇒ R → symmetric R
symmetric-from-˘⇒ {R} R˘⇒R a₁ a₂ a₁Ra₂ = R˘⇒R (a₂ , a₁) (to-∈-conv R a₁ a₂ a₁Ra₂)

symmetric-from-˘⊆ : {R : RelRepr₀} → conv R ⊆ R → symmetric R
symmetric-from-˘⊆ R˘⊆R = symmetric-from-˘⇒ (⊆-to-⇒ R˘⊆R)

symmetric-from-˘≈ : {R : RelRepr₀} → conv R ≈ R → symmetric R
symmetric-from-˘≈ R˘≈R = symmetric-from-˘⇒ (≈-to-⇒ R˘≈R)

symmetric-from-≈˘ : {R : RelRepr₀} → R ≈ conv R → symmetric R
symmetric-from-≈˘ R≈R˘ = symmetric-from-˘⇒ (proj₂ (≈-to-⇔ R≈R˘))


transitive-to-⇒ : {R : RelRepr₀} → transitive R → comp R R ⇒ R
transitive-to-⇒ {R} transR (a₁ , a₃) a₁RRa₃ with from-∈-comp _ _ a₁ a₃ a₁RRa₃
... | a₂ , a₁Ra₂ , a₂Ra₃ = transR a₁ a₂ a₃ a₁Ra₂ a₂Ra₃

transitive-to-⊆ : {R : RelRepr₀} → transitive R → comp R R ⊆ R
transitive-to-⊆ transR = ⊆-from-⇒ (transitive-to-⇒ transR)

transitive-from-⇒ : {R : RelRepr₀} → comp R R ⇒ R → transitive R
transitive-from-⇒ RR⇒R a₁ a₂ a₃ a₁Ra₂ a₂Ra₃ = RR⇒R (a₁ , a₃) (to-∈-comp _ _ a₁ a₂ a₃ a₁Ra₂ a₂Ra₃)

transitive-from-⊆ : {R : RelRepr₀} → comp R R ⊆ R → transitive R
transitive-from-⊆ RR⊆R = transitive-from-⇒ (⊆-to-⇒ RR⊆R)

transitive-from-≈ : {R : RelRepr₀} → comp R R ≈ R → transitive R
transitive-from-≈ RR≈R = transitive-from-⇒ (≈-to-⇒ RR≈R)


quotProj₁-⇒ : (E P : RelRepr₀) → symmetric E → transitive E
             → idempotU P → P ⇒ E → comp P (conv P) ⇒ E
quotProj₁-⇒ E P E-sym E-trans P-idempotU P⇒E (a₁ , a₃) a₁PP˘a₃ with from-∈-comp _ _ a₁ a₃ a₁PP˘a₃
... | a₂ , a₁Pa₂ , a₂P˘a₃ = let
  a₃Pa₂ = from-∈-conv _ a₃ a₂ a₂P˘a₃
  a₃Ea₂ = P⇒E (a₃ , a₂) a₃Pa₂
  a₂Ea₃ = E-sym a₃ a₂ a₃Ea₂
  a₁Ea₂ = P⇒E (a₁ , a₂) a₁Pa₂
  in E-trans a₁ a₂ a₃ a₁Ea₂ a₂Ea₃

quotProj₁-⇐ : (E P : RelRepr₀) → symmetric E → E dom⊆ P
             → ((a b c : A₀) → (a , b) ∈ E → (b , c) ∈ P → (a , c) ∈ P)
             → E ⇒ comp P (conv P)
quotProj₁-⇐ E P E-sym E-dom⊆-P leftClosed (a₁ , a₃) a₁Ea₃
  with E-dom⊆-P a₃ a₁ (E-sym a₁ a₃ a₁Ea₃)
... | a₂ , a₃Pa₂ = let a₁Pa₂ = leftClosed a₁ a₃ a₂ a₁Ea₃ a₃Pa₂
    in to-∈-comp _ _ a₁ a₂ a₃ a₁Pa₂ (to-∈-conv _ a₃ a₂ a₃Pa₂)

quotProj₁ : (E P : RelRepr₀) → symmetric E → transitive E
          → idempotU P → P ⇒ E → E dom⊆ P
          → ((a b c : A₀) → (a , b) ∈ E → (b , c) ∈ P → (a , c) ∈ P)
          → comp P (conv P) ≈ E
quotProj₁ E P E-sym E-trans P-idempotU P⇒E E-dom⊆-P leftClosed = ≈-from-⇔
  (quotProj₁-⇒ E P E-sym E-trans P-idempotU P⇒E , quotProj₁-⇐ E P E-sym E-dom⊆-P leftClosed)
```

21.21 Relation.Binary.ElemSet.SetReprConversions

```
record SetReprConversions {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
                          {ℓr₀ ℓr₁ ℓr₂ ℓr₃ : Level} (R : ElemSet A ℓr₀ ℓr₁ ℓr₂ ℓr₃)
                          {ℓs₀ ℓs₁ ℓs₂ ℓs₃ : Level} (S : ElemSet A ℓs₀ ℓs₁ ℓs₂ ℓs₃)
                          : Set (ℓa₀ ⊍ ℓr₀ ⊍ ℓr₃ ⊍ ℓs₀ ⊍ ℓs₃) where
  open TwoElemSets R S using (A₀; R₀; S₀; _∈R_; _∈S_)
  field R→S : R₀ → S₀
        S→R : S₀ → R₀
```

```
  from-∈-R→S : (r : R₀) (a : A₀) → a ∈S R→S r → a ∈R r
  to-∈-R→S   : (r : R₀) (a : A₀) → a ∈R r      → a ∈S R→S r
  from-∈-S→R : (s : S₀) (a : A₀) → a ∈R S→R s → a ∈S s
  to-∈-S→R   : (s : S₀) (a : A₀) → a ∈S s      → a ∈R S→R s

SetReprConversions⁻¹ : {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
                       {ℓr₀ ℓr₁ ℓr₂ ℓr₃ : Level} {R : ElemSet A ℓr₀ ℓr₁ ℓr₂ ℓr₃}
                       {ℓs₀ ℓs₁ ℓs₂ ℓs₃ : Level} {S : ElemSet A ℓs₀ ℓs₁ ℓs₂ ℓs₃}
                     → SetReprConversions R S → SetReprConversions S R
SetReprConversions⁻¹ SRC = let open SetReprConversions SRC in record
  {R→S         = S→R
  ;S→R         = R→S
  ;from-∈-R→S = from-∈-S→R
  ;to-∈-R→S   = to-∈-S→R
  ;from-∈-S→R = from-∈-R→S
  ;to-∈-S→R   = to-∈-R→S
  }
```

21.22 Relation.Binary.ElemRel.Dom

```
record ElemRelDom
    {ℓa₀ ℓa₁ : Level} (A : Setoid ℓa₀ ℓa₁)
    {ℓb₀ ℓb₁ : Level} (B : Setoid ℓb₀ ℓb₁)
    {ℓr₀ : Level} {ReprA : Set ℓr₀} {ℓ∈A : Level} (_∈A_ : ⌊ A ⌋ → ReprA → Set ℓ∈A)
    {ℓs₀ : Level} {ReprAB : Set ℓs₀} {ℓ∈AB : Level} (_∈AB_ : ⌊ A ⌋ × ⌊ B ⌋ → ReprAB → Set ℓ∈AB)
    (dom : ReprAB → ReprA) : Set (ℓa₀ ⊍ ℓa₁ ⊍ ℓb₀ ⊍ ℓb₁ ⊍ ℓr₀ ⊍ ℓs₀ ⊍ ℓ∈A ⊍ ℓ∈AB) where
  open SetoidA A
  open SetoidB B
  field
    from-∈-dom : (R : ReprAB) (a : A₀) → a ∈A dom R → Σ b : B₀ • (a, b) ∈AB R
    to-∈-dom : (R : ReprAB) (a : A₀) (b : B₀) → (a, b) ∈AB R → a ∈A dom R
```

21.23 Relation.Binary.ElemRel.Ran

```
record ElemRelRan
    {ℓa₀ ℓa₁ : Level} (A : Setoid ℓa₀ ℓa₁)
    {ℓb₀ ℓb₁ : Level} (B : Setoid ℓb₀ ℓb₁)
    {ℓr₀ : Level} {ReprB : Set ℓr₀} {ℓ∈B : Level} (_∈B_ : ⌊ B ⌋ → ReprB → Set ℓ∈B)
    {ℓs₀ : Level} {ReprAB : Set ℓs₀} {ℓ∈AB : Level} (_∈AB_ : ⌊ A ⌋ × ⌊ B ⌋ → ReprAB → Set ℓ∈AB)
    (ran : ReprAB → ReprB) : Set (ℓa₀ ⊍ ℓa₁ ⊍ ℓb₀ ⊍ ℓb₁ ⊍ ℓr₀ ⊍ ℓs₀ ⊍ ℓ∈B ⊍ ℓ∈AB) where
  open SetoidA A
  open SetoidB B
  field
    from-∈-ran : (R : ReprAB) (b : B₀) → b ∈B ran R → Σ [ a : A₀ ] (a, b) ∈AB R
    to-∈-ran : (R : ReprAB) (a : A₀) (b : B₀) → (a, b) ∈AB R → b ∈B ran R
```

21.24 Relation.Binary.ElemRel.Conv2-Ran

```
open ElemRel
module ElemRel-Conv2-Ran
```

```
    {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
    {ℓb₀ ℓb₁ : Level} {B : Setoid ℓb₀ ℓb₁}
    {ℓq₀ ℓq₁ ℓq₂ ℓq∈ : Level} (ℙB : ElemSet B ℓq₀ ℓq₁ ℓq₂ ℓq∈)
    {ℓr₀ ℓr₁ ℓr₂ ℓr∈ : Level} (AB : ElemRel A B ℓr₀ ℓr₁ ℓr₂ ℓr∈)
    {ℓs₀ ℓs₁ ℓs₂ ℓs∈ : Level} (BA : ElemRel B A ℓs₀ ℓs₁ ℓs₂ ℓs∈)
    {ℓu₀ ℓu₁ ℓu₂ ℓu∈ : Level} (BB : ElemRel B B ℓu₀ ℓu₁ ℓu₂ ℓu∈)
    {conv : RelRepr₀ AB → RelRepr₀ BA}
    (EC-AB : ElemRelConv A B (_∈_ AB) (_∈_ BA) conv)
    {comp : RelRepr₀ BA → RelRepr₀ AB → RelRepr₀ BB}
    (EC : ElemRelComp B A B (_∈_ BA) (_∈_ AB) (_∈_ BB) comp)
    {Id : RelRepr₀ BB} (EId : ElemRelId B (_∈_ BB) Id)
    {meet : RelRepr₀ BB → RelRepr₀ BB → RelRepr₀ BB}
    (EM-BB : ElemSetMeet (B ×× B) (_∈_ BB) meet)
    {ran : RelRepr₀ AB → ElemSet.SetRepr₀ ℙB}
    (ERan : ElemRelRan A B (ElemSet._∈_ ℙB) (_∈_ AB) ran)
    {SubId : ElemSet.SetRepr₀ ℙB → RelRepr₀ BB}
    (ES : ElemRelSubId B (ElemSet._∈_ ℙB) (_∈_ BB) SubId)
  where
    open SetoidA A
    open SetoidB B
    private
      module AB = ElemRel AB
      module BA = ElemRel BA
      module BB = ElemRel BB
    open ElemRelConv EC-AB
    open ElemRelComp EC
    open ElemRelId EId
    open ElemSetMeet (B ×× B) (_∈_ BB) EM-BB
    open ElemRelRan ERan
    open ElemRelSubId ES


    SubId-ran-⇒ : {R : AB.RelRepr₀} → SubId (ran R) BB.⇒ meet Id (comp (conv R) R)
    SubId-ran-⇒ {R} (b₁ , b₂) b₁RanRb₂ with from-∈-SubId _ b₁ b₂ b₁RanRb₂
    ... | b₁∈ranR , b₁≈b₂ with from-∈-ran R b₁ b₁∈ranR
    ... | a , aRb₁ = to-∈-intersection _ _ (b₁ , b₂) (to-∈-Id b₁ b₂ b₁≈b₂)
                         (to-∈-comp _ _ b₁ a b₂ (to-∈-conv _ a b₁ aRb₁) (AB.∈-subst₁₂ _ b₁≈b₂ aRb₁))
    SubId-ran-⊆ : {R : AB.RelRepr₀} → SubId (ran R) BB.⊆ meet Id (comp (conv R) R)
    SubId-ran-⊆ = BB.⊆-from-⇒ SubId-ran-⇒
    SubId-ran-⇐ : {R : AB.RelRepr₀} → meet Id (comp (conv R) R) BB.⇒ SubId (ran R)
    SubId-ran-⇐ {R} (b₁ , b₂) b₁I∩R˘Rb₂ with from-∈-intersection _ _ (b₁ , b₂) b₁I∩R˘Rb₂
    ... | b₁Ib₂ , b₁R˘Rb₂ with from-∈-comp _ _ b₁ b₂ b₁R˘Rb₂
    ... | a , b₁R˘a , aRb₂ = let b₁≈b₂ = from-∈-Id b₁ b₂ b₁Ib₂
      in to-∈-SubId _ b₁ b₂ (to-∈-ran _ a b₁ (from-∈-conv _ a b₁ b₁R˘a)) b₁≈b₂
    SubId-ran-⊇ : {R : AB.RelRepr₀} → meet Id (comp (conv R) R) BB.⊆ SubId (ran R)
    SubId-ran-⊇ = BB.⊆-from-⇒ SubId-ran-⇐
    SubId-ran : {R : AB.RelRepr₀} → SubId (ran R) BB.≈ meet Id (comp (conv R) R)
    SubId-ran = BB.≈-from-⇔ (SubId-ran-⇒ , SubId-ran-⇐)
```

21.25 Relation.Binary.ElemRel.Homogeneous

```
module IsHomElemRel {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
                    {j k₁ k₂ ℓ : Level} (RelRepr : Poset j k₁ k₂)
                    {_∈_ : ⌊ A ⌋ × ⌊ A ⌋ → ⌊ RelRepr ≤⌋ → Set ℓ}
                    (isElemRel : IsElemRel′ A A RelRepr _∈_) where
  open SetoidA A
```

```
  open IsElemRel′ RelRepr isElemRel

  reflexive : (R : RelRepr₀) → Set (ℓ ⊍ ℓa₀ ⊍ ℓa₁)
  reflexive R = (a₀ a₁ : A₀) → a₀ ≈A a₁ → (a₀ , a₁) ∈ R

  coreflexive : (R : RelRepr₀) → Set (ℓ ⊍ ℓa₀ ⊍ ℓa₁)
  coreflexive R = (a₀ a₁ : A₀) → (a₀ , a₁) ∈ R → a₀ ≈A a₁

  symmetric : (R : RelRepr₀) → Set (ℓ ⊍ ℓa₀)
  symmetric R = (a₀ a₁ : A₀) → (a₀ , a₁) ∈ R → (a₁ , a₀) ∈ R

  antisymmetric : (R : RelRepr₀) → Set (ℓ ⊍ ℓa₀ ⊍ ℓa₁)
  antisymmetric R = (a₀ a₁ : A₀) → (a₀ , a₁) ∈ R → (a₁ , a₀) ∈ R → a₀ ≈A a₁

  transitive : (R : RelRepr₀) → Set (ℓ ⊍ ℓa₀)
  transitive R = (a₀ a₁ a₂ : A₀) → (a₀ , a₁) ∈ R → (a₁ , a₂) ∈ R → (a₀ , a₂) ∈ R

    -- If R is univalent, then it satisfies idempotU iff it is idempotent.
  idempotU : (R : RelRepr₀) → Set (ℓ ⊍ ℓa₀)
  idempotU R = (a₀ a₁ : A₀) → (a₀ , a₁) ∈ R → (a₁ , a₁) ∈ R


HomElemRel : {ℓa₀ ℓa₁ : Level} (A : Setoid ℓa₀ ℓa₁)
  →              (j k₁ k₂ ℓ : Level) → Set (ℓsuc (j ⊍ k₁ ⊍ k₂ ⊍ ℓ) ⊍ ℓa₀ ⊍ ℓa₁)
HomElemRel A = ElemRel A A

module HomElemRel {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
                  {j k₁ k₂ ℓ : Level} (HER : HomElemRel A j k₁ k₂ ℓ) where
  open ElemRel HER public
  open IsHomElemRel RelRepr isElemRel public
```

21.26 Relation.Binary.ElemRel.SubIdCong

```
open ElemRel

module ElemRel-SubIdCong
    {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
    {ℓq₀ ℓq₁ ℓq₂ ℓq∈ : Level} (ℙA : ElemSet A ℓq₀ ℓq₁ ℓq₂ ℓq∈)
    {ℓr₀ ℓr₁ ℓr₂ ℓr∈ : Level} (AA : ElemRel A A ℓr₀ ℓr₁ ℓr₂ ℓr∈)
    {SubId : ElemSet.SetRepr₀ ℙA → RelRepr₀ AA}
    (ES : ElemRelSubId A (ElemSet._∈_ ℙA) (_∈_ AA) SubId)
  where
    private
      module ℙA = ElemSet ℙA
      module AA = ElemRel AA
    open SetoidA A
    open ElemRelSubId ES

    SubId-monotone-⇒ : {s₁ s₂ : ℙA.SetRepr₀} → s₁ ℙA.⇒ s₂ → SubId s₁ AA.⇒ SubId s₂
    SubId-monotone-⇒ {s₁} {s₂} s₁⇒s₂ (a₁ , a₂) a₁S₁a₂ with from-∈-SubId _ a₁ a₂ a₁S₁a₂
    ... | a₁∈s₁ , a₁≈a₂ = to-∈-SubId _ a₁ a₂ (s₁⇒s₂ a₁ a₁∈s₁) a₁≈a₂

    SubId-monotone : {s₁ s₂ : ℙA.SetRepr₀} → s₁ ℙA.⊆ s₂ → SubId s₁ AA.⊆ SubId s₂
    SubId-monotone s₁⊆s₂ = AA.⊆-from-⇒ (SubId-monotone-⇒ (ℙA.⊆-to-⇒ s₁⊆s₂))

    SubId-cong : {s₁ s₂ : ℙA.SetRepr₀} → s₁ ℙA.≈ s₂ → SubId s₁ AA.≈ SubId s₂
    SubId-cong s₁≈s₂ with ℙA.≈-to-⇔ s₁≈s₂
    ... | s₁⇒s₂ , s₂⇒s₁ = AA.≈-from-⇔ (SubId-monotone-⇒ s₁⇒s₂ , SubId-monotone-⇒ s₂⇒s₁)

    SubId⁻¹-monotone-⇒ : {s₁ s₂ : ℙA.SetRepr₀} → SubId s₁ AA.⇒ SubId s₂ → s₁ ℙA.⇒ s₂
    SubId⁻¹-monotone-⇒ {s₁} {s₂} S₁⇒S₂ a₁ a₁∈s₁ = proj₁ (from-∈-SubId _ a₁ a₁
      (S₁⇒S₂ (a₁ , a₁) (to-∈-SubId _ a₁ a₁ a₁∈s₁ ≈A-refl)))

    SubId⁻¹-monotone : {s₁ s₂ : ℙA.SetRepr₀} → SubId s₁ AA.⊆ SubId s₂ → s₁ ℙA.⊆ s₂
    SubId⁻¹-monotone S₁⊆S₂ = ℙA.⊆-from-⇒ (SubId⁻¹-monotone-⇒ (AA.⊆-to-⇒ S₁⊆S₂))

    SubId⁻¹-cong : {s₁ s₂ : ℙA.SetRepr₀} → SubId s₁ AA.≈ SubId s₂ → s₁ ℙA.≈ s₂
```

```
  SubId⁻¹-cong S₁≈S₂ with AA.≈-to-⇔ S₁≈S₂
  ... | S₁⇒S₂, S₂⇒S₁  =   ℙA.≈-from-⇔ (SubId⁻¹-monotone-⇒ S₁⇒S₂, SubId⁻¹-monotone-⇒ S₂⇒S₁)
```

21.27 Relation.Binary.ElemSet.ReprIso

```
module SetReprRightInverse {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
                           {ℓr₀ ℓr₁ ℓr₂ ℓr₃ : Level} {R : ElemSet A ℓr₀ ℓr₁ ℓr₂ ℓr₃}
                           {ℓs₀ ℓs₁ ℓs₂ ℓs₃ : Level} {S : ElemSet A ℓs₀ ℓs₁ ℓs₂ ℓs₃}
                           (RtoS : SetReprConversions R S) where
  open TwoElemSets R S
  open SetReprConversions RtoS
  R→S→R-⇒ : (r : R₀) → S→R (R→S r) ⇒R r
  R→S→R-⇒ r a a∈r′ = from-∈-R→S r a (from-∈-S→R (R→S r) a a∈r′)
  R→S→R-⇐ : (r : R₀) → r ⇒R S→R (R→S r)
  R→S→R-⇐ r a a∈r = to-∈-S→R (R→S r) a (to-∈-R→S r a a∈r)
  R→S→R-⊆ : (r : R₀) → S→R (R→S r) ⊆R r
  R→S→R-⊆ r = ⊆R-from-⇒ (R→S→R-⇒ r)
  R→S→R-⊇ : (r : R₀) → r ⊆R S→R (R→S r)
  R→S→R-⊇ r = ⊆R-from-⇒ (R→S→R-⇐ r)
  R→S→R : (r : R₀) → S→R (R→S r) ≈R r
  R→S→R r = ≈R-from-⇔ (R→S→R-⇒ r, R→S→R-⇐ r)
  R→S-monotone-⇒ : (r₁ r₂ : R₀) → r₁ ⇒R r₂ → R→S r₁ ⇒S R→S r₂
  R→S-monotone-⇒ r₁ r₂ r₁⇒r₂ a a∈s₁ = let
    a∈r₁ = from-∈-R→S r₁ a a∈s₁
    a∈r₂ = r₁⇒r₂ a a∈r₁
    in to-∈-R→S r₂ a a∈r₂
  R→S-monotone : (r₁ r₂ : R₀) → r₁ ⊆R r₂ → R→S r₁ ⊆S R→S r₂
  R→S-monotone r₁ r₂ r₁⊆r₂ = ⊆S-from-⇒ (R→S-monotone-⇒ r₁ r₂ (⊆R-to-⇒ r₁⊆r₂))
  R→S-cong : (r₁ r₂ : R₀) → r₁ ≈R r₂ → R→S r₁ ≈S R→S r₂
  R→S-cong r₁ r₂ r₁≈r₂ with ≈R-to-⇔ r₁≈r₂
  ... | r₁⇒r₂, r₂⇒r₁ = ≈S-from-⇔ (R→S-monotone-⇒ r₁ r₂ r₁⇒r₂, R→S-monotone-⇒ r₂ r₁ r₂⇒r₁)
  R⟶S : R≈ ⟶ S≈
  R⟶S = record {_⟨$⟩_ = R→S; cong = λ {r₁} {r₂} r₁≈r₂ → R→S-cong r₁ r₂ r₁≈r₂}


module ReprIso {ℓa₀ ℓa₁ : Level} {A : Setoid ℓa₀ ℓa₁}
               {ℓr₀ ℓr₁ ℓr₂ ℓr₃ : Level} {R : ElemSet A ℓr₀ ℓr₁ ℓr₂ ℓr₃}
               {ℓs₀ ℓs₁ ℓs₂ ℓs₃ : Level} {S : ElemSet A ℓs₀ ℓs₁ ℓs₂ ℓs₃}
               (RtoS : SetReprConversions R S) where
  open TwoElemSets R S
  open SetReprConversions RtoS public
  open SetReprRightInverse RtoS public
  module SetReprLeftInverse where
    open SetReprRightInverse (SetReprConversions⁻¹ RtoS) public renaming
      (R→S→R-⇒          to S→R→S-⇒
      ; R→S→R-⇐         to S→R→S-⇐
      ; R→S→R-⊆         to S→R→S-⊆
      ; R→S→R-⊇         to S→R→S-⊇
      ; R→S→R           to S→R→S
      ; R→S-monotone-⇒  to S→R-monotone-⇒
      ; R→S-monotone    to S→R-monotone
      ; R→S-cong        to S→R-cong
      ; R⟶S             to S⟶R
      )
```

```
open SetReprLeftInverse public
R⟶S⟶R : LeftInverse R≈ S≈
R⟶S⟶R = record {to = R⟶S; from = S⟶R; left-inverse-of = R→S→R}
S⟶R⟶S : RightInverse R≈ S≈
S⟶R⟶S = record {to = S⟶R; from = R⟶S; left-inverse-of = S→R→S}
R⟷S-inverse : S⟶R InverseOf R⟶S
R⟷S-inverse = record {left-inverse-of = R→S→R; right-inverse-of = S→R→S}
R⟷S : Inverse R≈ S≈
R⟷S = record {to = R⟶S; from = S⟶R; inverse-of = R⟷S-inverse}
```

Part IV

More Products

Chapter 22

Product Categories

22.1 Categoric.Product.Semigroupoid

```
module ProdComp
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} (Base₁ : Semigroupoid {i₁} j₁ k₁ Obj₁)
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} (Base₂ : Semigroupoid {i₂} j₂ k₂ Obj₂)
  where
  open Semigroupoid Base₁ using () renaming (_⨾_ to _⨾₁_; Hom to Hom₁; Mor to Mor₁; _≈_ to _≈₁_)
  open Semigroupoid Base₂ using () renaming (_⨾_ to _⨾₂_; Hom to Hom₂; Mor to Mor₂; _≈_ to _≈₂_)
  open Semigroupoid

  infixr 9 _⨾P_
  infix  4 _≈P_

  ProdObj = Obj₁ × Obj₂

  ProdHom : ProdObj → ProdObj → Setoid (j₁ ⊍ j₂) (k₁ ⊍ k₂)
  ProdHom (A₁ , A₂) (B₁ , B₂) = Hom₁ A₁ B₁ ×-setoid Hom₂ A₂ B₂

  ProdMor : ProdObj → ProdObj → Set (j₁ ⊍ j₂)
  ProdMor A B = ⌊ ProdHom A B ⌋

  _≈P_ : {A B : ProdObj} → ProdMor A B → ProdMor A B → Set (k₁ ⊍ k₂)
  _≈P_ {A} {B} = Setoid._≈_ (ProdHom A B)

  _⨾P_ : {A B C : ProdObj} → ProdMor A B → ProdMor B C → ProdMor A C
  (F₁ , F₂) ⨾P (G₁ , G₂) = F₁ ⨾₁ G₁ , F₂ ⨾₂ G₂

  ⨾P-cong : {A B C : ProdObj} {F₁ F₂ : ProdMor A B} {G₁ G₂ : ProdMor B C}
    → F₁ ≈P F₂ → G₁ ≈P G₂ → F₁ ⨾P G₁ ≈P F₂ ⨾P G₂
  ⨾P-cong (f₁≈f₂ , F₁≈F₂) (g₁≈g₂ , G₁≈G₂) = ⨾-cong Base₁ f₁≈f₂ g₁≈g₂ , ⨾-cong Base₂ F₁≈F₂ G₁≈G₂

  ⨾P-assoc : {A B C D : ProdObj} {F : ProdMor A B} {G : ProdMor B C} {H : ProdMor C D}
    → (F ⨾P G) ⨾P H ≈P F ⨾P (G ⨾P H)
  ⨾P-assoc {F = F₁ , F₂} {G₁ , G₂} {H₁ , H₂} = ⨾-assoc Base₁ {_} {_} {_} {_} {F₁} {G₁} {H₁}
                                           , ⨾-assoc Base₂ {_} {_} {_} {_} {F₂} {G₂} {H₂}

ProductSemigroupoid : Semigroupoid {i₁ ⊍ i₂} (j₁ ⊍ j₂) (k₁ ⊍ k₂) ProdObj
ProductSemigroupoid = record
  {Hom = ProdHom
  ; compOp = record
    {_⨾_     = _⨾P_
    ; ⨾-cong  = ⨾P-cong
    ; ⨾-assoc = ⨾P-assoc
    }
  }

open ProdComp public using (ProductSemigroupoid)
```

22.2 Categoric.Product.Category

```
module ProdCat
  {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
  ProductCategory : Category {i₁ ⊍ i₂} (j₁ ⊍ j₂) (k₁ ⊍ k₂) (Obj₁ × Obj₂)
  ProductCategory = record
    {semigroupoid = ProductSemigroupoid 𝒞₁.semigroupoid 𝒞₂.semigroupoid
    ; idOp = record
      {Id        = Id₁, Id₂
      ; leftId   = leftId₁, leftId₂
      ; rightId  = rightId₁, rightId₂
      }
    }

Proj₁ : Functor ProductCategory 𝒞₁
Proj₁ = record
  {obj = proj₁
  ; mor = proj₁
  ; mor-cong = proj₁
  ; mor-⨾ = ≈₁-refl
  ; mor-Id = ≈₁-refl
  }

Proj₂ : Functor ProductCategory 𝒞₂
Proj₂ = record
  {obj = proj₂
  ; mor = proj₂
  ; mor-cong = proj₂
  ; mor-⨾ = ≈₂-refl
  ; mor-Id = ≈₂-refl
  }

open ProdCat public

module _
  {i₁ j₁ k₁ : Level} {Obj₁ : Set i₁} {𝒞₁ : Category j₁ k₁ Obj₁}
  {i₂ j₂ k₂ : Level} {Obj₂ : Set i₂} {𝒞₂ : Category j₂ k₂ Obj₂}
  {i₃ j₃ k₃ : Level} {Obj₃ : Set i₃} {𝒞₃ : Category j₃ k₃ Obj₃}
    where
  open Category₁ 𝒞₁
  open Category₂ 𝒞₂
  open Category₃ 𝒞₃
  infixr 4 _▼_
  _▼_ : Functor 𝒞₃ 𝒞₁ → Functor 𝒞₃ 𝒞₂ → Functor 𝒞₃ (ProductCategory 𝒞₁ 𝒞₂)
  F ▼ G = record
    {obj = λ A → F.obj A, G.obj A
    ; mor = λ f → F.mor f, G.mor f
    ; mor-cong = λ f≈g → F.mor-cong f≈g, G.mor-cong f≈g
```

```
    ; mor-⨾ = F.mor-⨾, G.mor-⨾
    ; mor-Id = F.mor-Id, G.mor-Id
    }
    where
      module F = Functor F
      module G = Functor G


biFunctor : Bifunctor 𝒞₁ 𝒞₂ 𝒞₃ → Functor (ProductCategory 𝒞₁ 𝒞₂) 𝒞₃
biFunctor F = record
  {obj = λ {(A, B) → F.obj A B}
  ; mor = λ {(f, g) → F.mor f g}
  ; mor-cong = λ {(f₁≈f₂, g₁≈g₂) → F.mor-cong f₁≈f₂ g₁≈g₂}
  ; mor-⨾ = F.mor-⨾
  ; mor-Id = F.mor-Id
  }
  where
    module F = Bifunctor F

toBifunctor : Functor (ProductCategory 𝒞₁ 𝒞₂) 𝒞₃ → Bifunctor 𝒞₁ 𝒞₂ 𝒞₃
toBifunctor F = record
  {obj = λ A B → F.obj (A, B)
  ; mor = λ f g → F.mor (f, g)
  ; mor-cong = λ f₁≈f₂ g₁≈g₂ → F.mor-cong (f₁≈f₂, g₁≈g₂)
  ; mor-⨾   = F.mor-⨾
  ; mor-Id = F.mor-Id
  }
  where
    module F = Functor F


open import Categoric.Category.FinColimits
open CatFinColimits


ProductInitial : HasInitialObject 𝒞₁ → HasInitialObject 𝒞₂ → HasInitialObject (ProductCategory 𝒞₁ 𝒞₂)
ProductInitial hasInit₁ hasInit₂ = record
  {⓵ = ⓵₁, ⓵₂
  ; isInitial = (ⓘ₁, ⓘ₂), (λ _ → ≈ⓘ₁, ≈ⓘ₂)
  }
  where
    open HasInitialObject₁ 𝒞₁ hasInit₁
    open HasInitialObject₂ 𝒞₂ hasInit₂


ProductCoproducts : HasCoproducts 𝒞₁ → HasCoproducts 𝒞₂ → HasCoproducts (ProductCategory 𝒞₁ 𝒞₂)
ProductCoproducts hasCoprod₁ hasCoprod₂ = record
  {_⊞_ = λ {(A₁, A₂) (B₁, B₂) → (A₁ ⊞₁ B₁), (A₂ ⊞₂ B₂)}
  ; ι = ι₁, ι₂
  ; κ = κ₁, κ₂
  ; isCoproduct = λ {(F₁, F₂) (G₁, G₂) → record
    {univMor = F₁ ⚠₁ G₁, F₂ ⚠₂ G₂
    ; univMor-factors-left = ι⨾⚠ 𝒞₁ hasCoprod₁, ι⨾⚠ 𝒞₂ hasCoprod₂
    ; univMor-factors-right = κ⨾⚠ 𝒞₁ hasCoprod₁, κ⨾⚠ 𝒞₂ hasCoprod₂
    ; univMor-unique = λ {(ι≈₁, ι≈₂) (κ≈₁, κ≈₂) → ⚠-unique 𝒞₁ hasCoprod₁ ι≈₁ κ≈₁
                                                 , ⚠-unique 𝒞₂ hasCoprod₂ ι≈₂ κ≈₂}
    }}
  }
  where
    open HasCoproducts₁ 𝒞₁ hasCoprod₁
    open HasCoproducts₂ 𝒞₂ hasCoprod₂
    open HasCoproducts
```

```
module _ {i₄ j₄ k₄ : Level} {Obj₄ : Set i₄} {𝒞₄ : Category j₄ k₄ Obj₄}
         (F : Functor 𝒞₁ 𝒞₃) (G : Functor 𝒞₂ 𝒞₄)
    where
  open Category₄ 𝒞₄
  private
    module F = Functor F
    module G = Functor G

  ProdFunctor : Functor (ProductCategory 𝒞₁ 𝒞₂) (ProductCategory 𝒞₃ 𝒞₄)
  ProdFunctor = record
    {obj = λ {(A₁, A₂) → F.obj A₁, G.obj A₂}
    ; mor = λ {(f₁, f₂) → F.mor f₁, G.mor f₂}
    ; mor-cong = λ {(f₁≈g₁, f₂≈g₂) → F.mor-cong f₁≈g₁, G.mor-cong f₂≈g₂}
    ; mor-⨾ = F.mor-⨾, G.mor-⨾
    ; mor-Id = F.mor-Id, G.mor-Id
    }
```

Chapter 23

Sort-Indexed Product Allegories etc.

23.1 Categoric.SortIndexedProduct.OrderedSemigroupoid

```
SIPOrderedSemigroupoid :  (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
                          → OrderedSemigroupoid {i} j k₁ k₂ Obj
                          → OrderedSemigroupoid {i} j k₁ k₂ (Sort → Obj)
SIPOrderedSemigroupoid Sort Base = let
  open OrderedSemigroupoid Base
  SG = SIPSemigroupoid Sort semigroupoid
  in record
  {Hom     = λ A B → record
    {Carrier = ∀ s → Mor (A s) (B s)
    ; _≈_ = λ F G → ∀ s → F s ≈ G s
    ; _≤_ = λ F G → ∀ s → F s ⊑ G s
    ; isPartialOrder = record
      {isPreorder = record
        {isEquivalence = Setoid.isEquivalence (Semigroupoid.Hom SG A B)
        ; reflexive = λ eq s → ⊑-reflexive (eq s)
        ; trans = λ fg gh s → ⊑-trans (fg s) (gh s)
        }
      ; antisym = λ leq geq s → ⊑-antisym (leq s) (geq s)
      }
    }
  ; compOp = Semigroupoid.compOp SG
  ; locOrd = record
    {⨾-monotone = λ {A} {B} {C} {F} {F'} {G} {G'} leqF leqG s → ⨾-monotone (leqF s) (leqG s)
    }
  }
```

23.2 Categoric.SortIndexedProduct.OrderedCategory

```
SIPOrderedCategory :  (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
                      → OrderedCategory {i} j k₁ k₂ Obj
                      → OrderedCategory {i} j k₁ k₂ (Sort → Obj)
SIPOrderedCategory Sort Base = let open OrderedCategory Base in record
  {orderedSemigroupoid = SIPOrderedSemigroupoid Sort orderedSemigroupoid
  ; idOp = Category.idOp (SIPCategory Sort category)
  }
```

The property isCoreflexive in from Categoric.OrderedCategory (Sect. 9.2) reflects without further effort, unlike isSubidentity in ordered semigroupoids, see Sect. 23.32, where SIPisSubidReflect is defined as additionally using a decidable equality on Sort.

```
private
  module SIPsubidReflect (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
                          (base : OrderedCategory j k₁ k₂ Obj)
    where
      OC = SIPOrderedCategory Sort base
      ObjSIP = Sort → Obj
      MorSIP = OrderedCategory.Mor OC
      SIPisCoreflexiveReflect : {A : ObjSIP} {p : MorSIP A A}
                              → OrderedCategory.isCoreflexive OC p
                              → (s : Sort) → OrderedCategory.isCoreflexive base (p s)
      SIPisCoreflexiveReflect {A} {p} p⊑Id s = p⊑Id s
      SIPisSubidReflect′ : {A : ObjSIP} {p : MorSIP A A}
                         → OrderedCategory.isSubidentity OC p
                         → (s : Sort) → OrderedCategory.isSubidentity base (p s)
      SIPisSubidReflect′ {A} {p} subid s =
        OrderedCategory.coreflexiveIsSubidentity base
          (SIPisCoreflexiveReflect
            (OrderedCategory.subidentityIsCoreflexive OC subid) s)


open SIPsubidReflect public using (SIPisCoreflexiveReflect; SIPisSubidReflect′)
```

23.3 Categoric.SortIndexedProduct.MeetOp

```
SIPMeetOp : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
          → {base : OrderedSemigroupoid {i} j k₁ k₂ Obj}
          → (meetOp : MeetOp base)
          → MeetOp (SIPOrderedSemigroupoid Sort base)
SIPMeetOp Sort meetOp = let open MeetOp meetOp in record
  {meet = λ R S → record
    {value = λ s → R s ⊓ S s
    ; proof = record
      {bound₁ = λ s → ⊓-lower₁
      ; bound₂ = λ s → ⊓-lower₂
      ; universal = λ X⊑R X⊑S s → ⊓-universal (X⊑R s) (X⊑S s)
      }
    }
  }
```

23.4 Categoric.SortIndexedProduct.LSLSemigroupoid

```
SIPLSLSemigroupoid : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
                   → LSLSemigroupoid {i} j k₁ k₂ Obj
                   → LSLSemigroupoid {i} j k₁ k₂ (Sort → Obj)
SIPLSLSemigroupoid Sort Base = let open LSLSemigroupoid Base in record
  {orderedSemigroupoid = SIPOrderedSemigroupoid Sort orderedSemigroupoid
  ; meetOp              = SIPMeetOp Sort meetOp
  }
```

23.5 Categoric.SortIndexedProduct.JoinOp

```
SIPJoinOp : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
          → {base : OrderedSemigroupoid {i} j k₁ k₂ Obj}
          → (joinOp : JoinOp base)
          → JoinOp (SIPOrderedSemigroupoid Sort base)
SIPJoinOp Sort joinOp = let open JoinOp joinOp in record
  {join = λ R S → record
    {value = λ s → R s ⊔ S s
    ; proof = record
      {bound₁ = λ s → ⊔-upper₁
      ; bound₂ = λ s → ⊔-upper₂
      ; universal = λ R⊑X S⊑X s → ⊔-universal (R⊑X s) (S⊑X s)
      }
    }
  }
```

```
SIPJoinCompDistrL : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
                  → {base : OrderedSemigroupoid {i} j k₁ k₂ Obj}
                  → {joinOp : JoinOp base}
                  → (joinCompDistrL : JoinCompDistrL joinOp)
                  → JoinCompDistrL (SIPJoinOp Sort joinOp)
SIPJoinCompDistrL Sort joinCompDistrL = let open JoinCompDistrL joinCompDistrL in record
  {⨾-⊔-subdistribL = λ s → ⨾-⊔-subdistribL}
```

```
SIPJoinCompDistrR : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
                  → {base : OrderedSemigroupoid {i} j k₁ k₂ Obj}
                  → {joinOp : JoinOp base}
                  → (joinCompDistrR : JoinCompDistrR joinOp)
                  → JoinCompDistrR (SIPJoinOp Sort joinOp)
SIPJoinCompDistrR Sort joinCompDistrR = let open JoinCompDistrR joinCompDistrR in record
  {⨾-⊔-subdistribR = λ s → ⨾-⊔-subdistribR}
```

23.6 Categoric.SortIndexedProduct.USLSemigroupoid

```
SIPUSLSemigroupoid : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
                   → USLSemigroupoid {i} j k₁ k₂ Obj
                   → USLSemigroupoid {i} j k₁ k₂ (Sort → Obj)
SIPUSLSemigroupoid Sort Base = let open USLSemigroupoid Base in record
  {orderedSemigroupoid = SIPOrderedSemigroupoid Sort orderedSemigroupoid
  ; joinOp = SIPJoinOp Sort joinOp
  ; joinCompDistrL = SIPJoinCompDistrL Sort joinCompDistrL
  ; joinCompDistrR = SIPJoinCompDistrR Sort joinCompDistrR
  }
```

23.7 Categoric.SortIndexedProduct.USLCategory

```
SIPUSLCategory : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
               → USLCategory {i} j k₁ k₂ Obj
               → USLCategory {i} j k₁ k₂ (Sort → Obj)
SIPUSLCategory Sort Base = let open USLCategory Base in record
  {orderedCategory = SIPOrderedCategory Sort orderedCategory
```

```
  ; joinOp            =  SIPJoinOp            Sort joinOp
  ; joinCompDistrL   =  SIPJoinCompDistrL  Sort joinCompDistrL
  ; joinCompDistrR   =  SIPJoinCompDistrR  Sort joinCompDistrR
  }
```

23.8 Categoric.SortIndexedProduct.LatticeSemigroupoid

```
SIPLatticeSemigroupoid :  (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
                          → LatticeSemigroupoid {i} j k₁ k₂ Obj
                          → LatticeSemigroupoid {i} j k₁ k₂ (Sort → Obj)
SIPLatticeSemigroupoid Sort Base  =  let open LatticeSemigroupoid Base in record
  {orderedSemigroupoid  =  SIPOrderedSemigroupoid Sort orderedSemigroupoid
  ; meetOp              =  SIPMeetOp              Sort meetOp
  ; joinOp              =  SIPJoinOp              Sort joinOp
  }
```

23.9 Categoric.SortIndexedProduct.HomLatticeDistr

```
SIPHomLatticeDistr :  (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
                      → {base : LatticeSemigroupoid {i} j k₁ k₂ Obj}
                      → (homLatDistr : HomLatticeDistr base)
                      → HomLatticeDistr (SIPLatticeSemigroupoid Sort base)
SIPHomLatticeDistr Sort homLatDistr  =  let open HomLatticeDistr homLatDistr in record
  {⊓-⊔-subdistribR  =  λ s → ⊓-⊔-subdistribR
  }
```

23.10 Categoric.SortIndexedProduct.DistrLatSemigroupoid

```
SIPDistrLatSemigroupoid : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
                          → DistrLatSemigroupoid {i} j k₁ k₂ Obj
                          → DistrLatSemigroupoid {i} j k₁ k₂ (Sort → Obj)
SIPDistrLatSemigroupoid Sort Base  =  let open DistrLatSemigroupoid Base in record
  {latticeSemigroupoid   =  SIPLatticeSemigroupoid Sort latticeSemigroupoid
  ; homLatDistr          =  SIPHomLatticeDistr     Sort homLatDistr
  ; joinCompDistrL       =  SIPJoinCompDistrL      Sort joinCompDistrL
  ; joinCompDistrR       =  SIPJoinCompDistrR      Sort joinCompDistrR
  }
```

23.11 Categoric.SortIndexedProduct.DomainSemigroupoid

```
SIPLeftClosOp :  (Sort : Set) {i j k : Level} {Obj : Set i} {base : Semigroupoid j k Obj}
                 → LeftClosOp base → LeftClosOp (SIPSemigroupoid Sort base)
SIPLeftClosOp Sort {Obj = Obj} {base = base} leftClosOp  =  let
  SG  =  SIPSemigroupoid Sort base
  ObjSIP  =  Sort → Obj
  MorSIP  =  Semigroupoid.Mor SG
  open LeftClosOp leftClosOp
  open Semigroupoid base
```

```
  domSIP : {A B : ObjSIP} → (R : MorSIP A B) → MorSIP A A
  domSIP = λ {A} {B} R s → dom (R s)
  in record
    {dom       = domSIP   -- λ {A} {B} R s → dom (R s)
    ; dom-cong = λ R≈S s → dom-cong (R≈S s)   -- : ∀ {R S} → R ≈ S → dom R ≈ dom S
    ; D1 = λ s → D1   -- : ∀ {R} → dom R ⨾ R ≈ R
    ; L2 = λ s → L2   -- : ∀ {R} → dom (dom R) ≈ dom R
    ; L3 = λ s → L3   -- : ∀ {R S} → dom R ⨾ dom (R ⨾ S) ≈ dom (R ⨾ S)
    ; D4 = λ s → D4   -- : ∀ {R S} → dom R ⨾ dom S ≈ dom S ⨾ dom R
    }


SIPPredomainOp : (Sort : Set) {i j k : Level} {Obj : Set i} {base : Semigroupoid j k Obj}
                 → PredomainOp base → PredomainOp (SIPSemigroupoid Sort base)
SIPPredomainOp Sort {Obj = Obj} {base = base} preDomainOp = let
  SG = SIPSemigroupoid Sort base
  ObjSIP = Sort → Obj
  MorSIP = Semigroupoid.Mor SG
  open PredomainOp preDomainOp
  open Semigroupoid base
  domSIP : {A B : ObjSIP} → (R : MorSIP A B) → MorSIP A A
  domSIP = λ {A} {B} R s → dom (R s)
  in record
    {dom       = domSIP   -- λ {A} {B} R s → dom (R s)
    ; dom-cong = λ R≈S s → dom-cong (R≈S s)   -- : ∀ {R S} → R ≈ S → dom R ≈ dom S
    ; D1 = λ s → D1   -- : ∀ {R} → dom R ⨾ R ≈ R
    ; D3 = λ s → D3   -- : ∀ {R S} → dom (dom R ⨾ S) ≈ dom R ⨾ dom S
    ; D4 = λ s → D4   -- : ∀ {R S} → dom R ⨾ dom S ≈ dom S ⨾ dom R
    }


SIPDomainOp : (Sort : Set) {i j k : Level} {Obj : Set i} {base : Semigroupoid j k Obj}
              → DomainOp base → DomainOp (SIPSemigroupoid Sort base)
SIPDomainOp Sort {Obj = Obj} {base = base} domainOp = let
  SG = SIPSemigroupoid Sort base
  ObjSIP = Sort → Obj
  MorSIP = Semigroupoid.Mor SG
  open DomainOp domainOp
  open Semigroupoid base
  domSIP : {A B : ObjSIP} → (R : MorSIP A B) → MorSIP A A
  domSIP = λ {A} {B} R s → dom (R s)
  in record
    {dom       = domSIP   -- λ {A} {B} R s → dom (R s)
    ; dom-cong = λ R≈S s → dom-cong (R≈S s)   -- : ∀ {R S} → R ≈ S → dom R ≈ dom S
    ; D1 = λ s → D1   -- : ∀ {R} → dom R ⨾ R ≈ R
    ; D2 = λ s → D2   -- : ∀ {R S} → dom (R ⨾ dom S) ≈ dom (R ⨾ S)
    ; D3 = λ s → D3   -- : ∀ {R S} → dom (dom R ⨾ S) ≈ dom R ⨾ dom S
    ; D4 = λ s → D4   -- : ∀ {R S} → dom R ⨾ dom S ≈ dom S ⨾ dom R
    }
```

23.12 Categoric.SortIndexedProduct.OCD

```
SIPdomainOP′ : (Sort : Set)
               {i j k₁ k₂ : Level} {Obj : Set i}
               (base : OrderedCategory j k₁ k₂ Obj)
             → let osg = OrderedCategory.orderedSemigroupoid base
               in OSGDomainOp osg
```

```
                   → OSGDomainOp (SIPOrderedSemigroupoid Sort osg)
SIPdomainOP′ Sort {Obj = Obj} base domainOp = let
  OC = SIPOrderedCategory Sort base
  ObjSIP = Sort → Obj
  MorSIP = OrderedCategory.Mor OC
  open OSGDomainOp domainOp
  open OrderedCategory base
  domSIP : {A B : ObjSIP} → (R : MorSIP A B) → MorSIP A A
  domSIP R s = dom (R s)
  in record
    {dom                = domSIP   -- λ {A} {B} R s → dom (R s)
    ; domSubIdentity    = (λ s → proj₁ domSubIdentity)
                        , (λ s → proj₂ domSubIdentity)
    ; dom-⨾-idempotent  = λ s → dom-⨾-idempotent
    ; domPreserves⊑     = λ Q⊑R s → domPreserves⊑ (Q⊑R s)
    ; domLeastPreserver = λ subid idem R⊑d⨾R s
                        → domLeastPreserver (SIPisSubidReflect′ Sort base subid s) (idem s) (R⊑d⨾R s)
    ; domLocality       = λ s → domLocality
    }


SIPOCD : (Sort : Set)
       → {i j k₁ k₂ : Level} {Obj : Set i}
       → OCD {i} j k₁ k₂ Obj
       → OCD {i} j k₁ k₂ (Sort → Obj)
SIPOCD Sort Base = let open OCD Base in record
  {orderedCategory = SIPOrderedCategory Sort orderedCategory
  ; domainOp       = SIPdomainOP′      Sort orderedCategory domainOp
  }


SIPrangeOP′ : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
            → (base : OrderedCategory j k₁ k₂ Obj)
            → let osg = OrderedCategory.orderedSemigroupoid base
                in OSGRangeOp osg
            → OSGRangeOp (SIPOrderedSemigroupoid Sort osg)
SIPrangeOP′ Sort {Obj = Obj} base rangeOp = let
  OC = SIPOrderedCategory Sort base
  ObjSIP = Sort → Obj
  MorSIP = OrderedCategory.Mor OC
  open OSGRangeOp rangeOp
  open OrderedCategory base
  ranSIP : {A B : ObjSIP} → (R : MorSIP A B) → MorSIP B B
  ranSIP R s = ran (R s)
  in record
    {ran                = ranSIP   -- λ {A} {B} R s → ran (R s)
    ; ranSubIdentity    = (λ s → proj₁ ranSubIdentity)
                        , (λ s → proj₂ ranSubIdentity)
    ; ran-⨾-idempotent  = λ s → ran-⨾-idempotent
    ; ranPreserves⊑     = λ Q⊑R s → ranPreserves⊑ (Q⊑R s)
    ; ranLeastPreserver = λ subid idem R⊑R⨾d s
                        → ranLeastPreserver (SIPisSubidReflect′ Sort base subid s) (idem s) (R⊑R⨾d s)
    ; ranLocality       = λ s → ranLocality
    }


SIPOCR : (Sort : Set)
       → {i j k₁ k₂ : Level} {Obj : Set i}
       → OCR {i} j k₁ k₂ Obj
       → OCR {i} j k₁ k₂ (Sort → Obj)
```

```
SIPOCR Sort Base = let open OCR Base in record
  {orderedCategory = SIPOrderedCategory Sort orderedCategory
  ; rangeOp        = SIPrangeOP′        Sort orderedCategory rangeOp
  }

SIPOCDR : (Sort : Set)
          → {i j k₁ k₂ : Level} {Obj : Set i}
          → OCDR {i} j k₁ k₂ Obj
          → OCDR {i} j k₁ k₂ (Sort → Obj)
SIPOCDR Sort Base = let open OCDR Base in record
  {orderedCategory = SIPOrderedCategory Sort orderedCategory
  ; domainOp       = SIPdomainOP′       Sort orderedCategory domainOp
  ; rangeOp        = SIPrangeOP′        Sort orderedCategory rangeOp
  }
```

23.13 Categoric.SortIndexedProduct.OSGC

```
SIPOSGC : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
          → OSGC {i} j k₁ k₂ Obj
          → OSGC {i} j k₁ k₂ (Sort → Obj)
SIPOSGC Sort Base = record
  {OSGC_Base = let open OSGC Base in record
    {orderedSemigroupoid = SIPOrderedSemigroupoid Sort orderedSemigroupoid
    ; convOp = ConvSemigroupoid.convOp (SIPConvSemigroupoid Sort convSemigroupoid)
    ; ˘-monotone = λ R⊑S s → ˘-monotone (R⊑S s)
    }
  }
```

23.14 Categoric.SortIndexedProduct.OCC-Base

```
SIPOCC-Base : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
              → OCC-Base {i} j k₁ k₂ Obj
              → OCC-Base {i} j k₁ k₂ (Sort → Obj)
SIPOCC-Base Sort Base = let open OCC-Base Base in record
  {osgc = SIPOSGC Sort osgc
  ; idOp = Category.idOp (SIPCategory Sort category)
  }
```

23.15 Categoric.SortIndexedProduct.OCC

```
SIPOCC : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
         → OCC {i} j k₁ k₂ Obj
         → OCC {i} j k₁ k₂ (Sort → Obj)
SIPOCC Sort Base = record {OCC_Base = SIPOCC-Base Sort (OCC.OCC_Base Base)}
```

23.16 Categoric.SortIndexedProduct.LeftResOp

```
SIPLeftResOp : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
               → {base : OrderedSemigroupoid {i} j k₁ k₂ Obj}
```

```
                  → (leftResOp : LeftResOp base)
                  → LeftResOp (SIPOrderedSemigroupoid Sort base)
SIPLeftResOp Sort leftResOp = let open LeftResOp leftResOp in record
  {_/_ = λ S R s → S s / R s
  ; /-cancel-outer = λ s → /-cancel-outer
  ; /-universal = λ Q⨾R⊑S s → /-universal (Q⨾R⊑S s)
  }
```

23.17 Categoric.SortIndexedProduct.RightResOp

```
SIPRightResOp : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
                  → {base : OrderedSemigroupoid {i} j k₁ k₂ Obj}
                  → (rightResOp : RightResOp base)
                  → RightResOp (SIPOrderedSemigroupoid Sort base)
SIPRightResOp Sort rightResOp = let open RightResOp rightResOp in record
  {_\_ = λ Q S s → Q s \ S s
  ; \-cancel-outer = λ s → \-cancel-outer
  ; \-universal = λ Q⨾R⊑S s → \-universal (Q⨾R⊑S s)
  }
```

23.18 Categoric.SortIndexedProduct.SyqOp

```
SIPSyqOp : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
             → {base : OSGC {i} j k₁ k₂ Obj}
             → (syqOp : SyqOp base)
             → SyqOp (SIPOSGC Sort base)
SIPSyqOp Sort syqOp = let open SyqOp syqOp in record
  {_╳_ = λ Q S s → Q s ╳ S s
  ; ╳-cong = λ Q₁≈Q₂ S₁≈S₂ s → ╳-cong (Q₁≈Q₂ s) (S₁≈S₂ s)
  ; ╳-cancel-left = λ s → ╳-cancel-left
  ; ╳-cancel-right = λ s → ╳-cancel-right
  ; ╳-universal = λ Q⨾R⊑S R⨾S˘⊑Q˘ s → ╳-universal (Q⨾R⊑S s) (R⨾S˘⊑Q˘ s)
  }
```

23.19 Categoric.SortIndexedProduct.USLSGC

```
SIPUSLSGC : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
              → USLSGC {i} j k₁ k₂ Obj
              → USLSGC {i} j k₁ k₂ (Sort → Obj)
SIPUSLSGC Sort Base = let open USLSGC Base in record
  {osgc = SIPOSGC Sort osgc
  ; joinOp = SIPJoinOp Sort joinOp
  ; joinCompDistrL = SIPJoinCompDistrL Sort joinCompDistrL
  ; joinCompDistrR = SIPJoinCompDistrR Sort joinCompDistrR
  }
```

23.20 Categoric.SortIndexedProduct.USLCC

```
SIPUSLCC : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
             → USLCC {i} j k₁ k₂ Obj
```

```
          → USLCC {i} j k₁ k₂ (Sort → Obj)
SIPUSLCC Sort Base = let open USLCC Base in record
  {occ             = SIPOCC              Sort occ
  ; joinOp         = SIPJoinOp           Sort joinOp
  ; joinCompDistrL = SIPJoinCompDistrL   Sort joinCompDistrL
  ; joinCompDistrR = SIPJoinCompDistrR   Sort joinCompDistrR
  }
```

23.21 Categoric.SortIndexedProduct.Allegory

```
SIPAllegory : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
          → Allegory {i} j k₁ k₂ Obj
          → Allegory {i} j k₁ k₂ (Sort → Obj)
SIPAllegory Sort Base = let open Allegory Base in record
  {occ       = SIPOCC     Sort occ
  ; meetOp   = SIPMeetOp Sort meetOp
  ; Dedekind = λ s → Dedekind
  }
```

23.22 Categoric.SortIndexedProduct.Collagory

```
SIPCollagory : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
          → Collagory {i} j k₁ k₂ Obj
          → Collagory {i} j k₁ k₂ (Sort → Obj)
SIPCollagory Sort Base = let open Collagory Base in record
  {allegory        = SIPAllegory          Sort allegory
  ; joinOp         = SIPJoinOp            Sort joinOp
  ; homLatDistr    = SIPHomLatticeDistr   Sort homLatDistr
  ; joinCompDistrL = SIPJoinCompDistrL    Sort joinCompDistrL
  ; joinCompDistrR = SIPJoinCompDistrR    Sort joinCompDistrR
  }
```

23.23 Categoric.SortIndexedProduct.ZeroMor

```
open OrderedSemigroupoid
open LeastMor
isLeastMorSIP : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
          → (base : OrderedSemigroupoid j k₁ k₂ Obj)
          → {A B : Sort → Obj}
          → let OSG = SIPOrderedSemigroupoid Sort base
            in {b : Mor OSG A B} → ((s : Sort) → isLeastMor base (b s))
          → isLeastMor OSG b
isLeastMorSIP Sort base {A} {B} {b} isLeast-b-s R t = isLeast-b-s t (R t)

SIPBotMor : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
          → {base : OrderedSemigroupoid {i} j k₁ k₂ Obj}
          → (botMor : BotMor base)
          → BotMor (SIPOrderedSemigroupoid Sort base)
SIPBotMor Sort botMor = let open BotMor botMor in record
  {leastMor = λ {A} {B} → record
```

```
      {mor  =  λ s → ⊥ {A s} {B s}
      ; proof  =  λ R s → ⊥-⊑ {A s} {B s} {R s}
      }
   }

SIPLeftZeroLaw :  (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
                  → {base : OrderedSemigroupoid {i} j k₁ k₂ Obj}
                  → {botMor : BotMor base}
                  → (leftZeroLaw : LeftZeroLaw botMor)
                  → LeftZeroLaw (SIPBotMor Sort botMor)
SIPLeftZeroLaw Sort leftZeroLaw  =  let open LeftZeroLaw leftZeroLaw in record
   {leftZero⊑  =  λ {A} {B} {C} {R} s → leftZero⊑ {A s} {B s} {C s} {R s}
   }

SIPRightZeroLaw :  (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
                   → {base : OrderedSemigroupoid {i} j k₁ k₂ Obj}
                   → {botMor : BotMor base}
                   → (rightZeroLaw : RightZeroLaw botMor)
                   → RightZeroLaw (SIPBotMor Sort botMor)
SIPRightZeroLaw Sort rightZeroLaw  =  let open RightZeroLaw rightZeroLaw in record
   {rightZero⊑  =  λ {A} {B} {C} {R} s → rightZero⊑ {A s} {B s} {C s} {R s}
   }

SIPZeroMor :  (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
              → {base : OrderedSemigroupoid {i} j k₁ k₂ Obj}
              → (zeroMor : ZeroMor base)
              → ZeroMor (SIPOrderedSemigroupoid Sort base)
SIPZeroMor Sort zeroMor  =  let open ZeroMor zeroMor in record
   {botMor        =  SIPBotMor        Sort botMor
   ; leftZeroLaw   =  SIPLeftZeroLaw   Sort leftZeroLaw
   ; rightZeroLaw  =  SIPRightZeroLaw  Sort rightZeroLaw
   }
```

23.24 Categoric.SortIndexedProduct.DistrAllegory

```
SIPDistrAllegory :  (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
                    → DistrAllegory {i} j k₁ k₂ Obj
                    → DistrAllegory {i} j k₁ k₂ (Sort → Obj)
SIPDistrAllegory Sort Base  =  let open DistrAllegory Base in record
   {collagory  =  SIPCollagory Sort collagory
   ; zeroMor   =  SIPZeroMor  Sort zeroMor
   }
```

23.25 Categoric.SortIndexedProduct.DivAllegory

```
SIPDivAllegory :  (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
                  → DivAllegory {i} j k₁ k₂ Obj
                  → DivAllegory {i} j k₁ k₂ (Sort → Obj)
SIPDivAllegory Sort Base  =  let open DivAllegory Base in record
   {distrAllegory  =  SIPDistrAllegory Sort distrAllegory
   ; leftResOp     =  SIPLeftResOp     Sort leftResOp
   ; rightResOp    =  SIPRightResOp    Sort rightResOp
   ; syqOp         =  SIPSyqOp         Sort syqOp
   }
```

23.26 Categoric.SortIndexedProduct.TransClosOp

```
SIPTransClosOp : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
                 → {base : USLSemigroupoid {i} j k₁ k₂ Obj}
                 → (transClosOp : TransClosOp base)
                 → TransClosOp (SIPUSLSemigroupoid Sort base)
SIPTransClosOp Sort transClosOp = let open TransClosOp transClosOp in record
  {_⁺ = λ R s → (R s) ⁺
  ; ⁺-recDef₁ = λ {_} {R} s → ⁺-recDef₁
  ; ⁺-recDef₂ = λ {_} {R} s → ⁺-recDef₂
  ; ⁺-leftInd  = λ {_} {R} {_} {S} R⨾S⊑S s → ⁺-leftInd (R⨾S⊑S s)
  ; ⁺-rightInd = λ {_} {R} {_} {Q} Q⨾R⊑Q s → ⁺-rightInd (Q⨾R⊑Q s)
  }
```

23.27 Categoric.SortIndexedProduct.KleeneSemigroupoid

```
SIPKleeneSemigroupoid : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
                        → KleeneSemigroupoid {i} j k₁ k₂ Obj
                        → KleeneSemigroupoid {i} j k₁ k₂ (Sort → Obj)
SIPKleeneSemigroupoid Sort Base = let open KleeneSemigroupoid Base in record
  {uslSemigroupoid = SIPUSLSemigroupoid Sort uslSemigroupoid
  ; transClosOp    = SIPTransClosOp     Sort transClosOp
  }
```

23.28 Categoric.SortIndexedProduct.KSGC

```
SIPKSGC : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
  → KSGC {i} j k₁ k₂ Obj
  → KSGC {i} j k₁ k₂ (Sort → Obj)
SIPKSGC Sort Base = let open KSGC Base in record
  {uslsgc       = SIPUSLSGC      Sort uslsgc
  ; transClosOp = SIPTransClosOp Sort transClosOp
  }
```

23.29 Categoric.SortIndexedProduct.StarOp

```
SIPStarOp : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
            → {base : USLCategory {i} j k₁ k₂ Obj}
            → (starOp : StarOp base)
            → StarOp (SIPUSLCategory Sort base)
SIPStarOp Sort starOp = let open StarOp starOp in record
  {_* = λ R s → (R s) *
  ; isStar = λ R → record
    {*-recDef   = λ s → *-recDef
    ; *-leftInd  = λ {_} {S} R⨾S⊑S s → *-leftInd (R⨾S⊑S s)
    ; *-rightInd = λ {_} {Q} Q⨾R⊑Q s → *-rightInd (Q⨾R⊑Q s)
    }
  }
```

23.30 Categoric.SortIndexedProduct.KleeneCategory

```
SIPKleeneCategory : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
                  → KleeneCategory {i} j k₁ k₂ Obj
                  → KleeneCategory {i} j k₁ k₂ (Sort → Obj)
SIPKleeneCategory Sort Base = let open KleeneCategory Base in record
  {uslCategory    = SIPUSLCategory Sort uslCategory
  ; zeroMor       = SIPZeroMor     Sort zeroMor
  ; starOp        = SIPStarOp      Sort starOp
  }
```

23.31 Categoric.SortIndexedProduct.KCC

```
SIPKCC : (Sort : Set) {i j k₁ k₂ : Level} {Obj : Set i}
       → KCC {i} j k₁ k₂ Obj
       → KCC {i} j k₁ k₂ (Sort → Obj)
SIPKCC Sort Base = let open KCC Base in record
  {uslcc    = SIPUSLCC   Sort uslcc
  ; zeroMor = SIPZeroMor Sort zeroMor
  ; starOp  = SIPStarOp  Sort starOp
  }
```

23.32 Categoric.SortIndexedProduct.OSGSubIdReflect

For all components of a SIP subidentity to be subidentities again, we need a decidable propositional equality on Sort, to be able to assemble a morphism with given target and one given component.

Let us consider reflection of isLeftSubidentity first, and recall its definition in ordered semigroupoids (Sect. 9.1), where no identity morphisms are assumed:

```
isLeftSubidentity : {A : Obj} → (p : Mor A A) → Set (i ⊍ j ⊍ k₂)
isLeftSubidentity {A} p = {B : Obj} {R : Mor A B} → p ⨾ R ⊑ R
```

Given a sort-indexed morphism p and a sort s, when showing that p s is a left sub-identity, we therefore have to accept an arbitrary base object B and base morphism R, and use these to construct a sort-indexed object B_1 and a sort-indexed morphism R_1 to supply to the proof of isLeftSubidentity p, before we can use the conclusion of that to extract the desired inclusion as the component for sort index s.

Since the context is that of only an ordered semigroupoid, the only base morphism known to start at A t for the sorts t besides s is p t. Therefore we essentially want to define B_1 and R_1 in the following way:

```
B₁ t = if ⌊ s ≟ t ⌋ then B else A t
R₁ t = if ⌊ s ≟ t ⌋ then R else p t
```

Since the type of R involves A s instead of A t, the use of Boolean if_then_else_ looses too much information, and we use the functions withDec and withDec-subst from Relation.Decidable.Utils (Sect. 2.6) instead.

At the end of the proof below, we see the resulting expression subid {B_1} {R_1} s, which however instead of the expected type claim B R (which is p s ⨾ R ⊑ R) has the type claim (B_2 (s ≟ s)) (R_1 s), and the remainder of the proof is concerned with this type adaptation. I would be grateful for information how this could be simplified.

```
private
  module SIPsubidReflectL
```

```
    (Sort : Set) (_≟_ : Decidable (_≡_ {A = Sort}))
    {i j k₁ k₂ : Level} {Obj : Set i}
    (base : OrderedSemigroupoid j k₁ k₂ Obj)
  where
  OSG = SIPOrderedSemigroupoid Sort base
  open OrderedSemigroupoid base
  open SGSIP Sort semigroupoid using (SIPObj; SIPMor)
  SIPisLeftSubidReflect : {A : SIPObj}
                    {p : SIPMor A A}
    →               OrderedSemigroupoid.isLeftSubidentity OSG p
    →               (s : Sort)
    →               OrderedSemigroupoid.isLeftSubidentity base (p s)
  SIPisLeftSubidReflect {A} {p} subid s {B} {R} = let
      T : {t : Sort} → s ≡ t → Obj
      T = λ _ → B
      E : (t : Sort) → ¬ (s ≡ t) → Obj
      E = λ t _ → A t
      P : (t : Sort) → Obj → Set j
      P = λ t → Mor (A t)
      B₀ : {t : Sort} → Dec (s ≡ t) → Obj
      B₀ {t} x = withDec x T (E t)
      B₁ : (t : Sort) → Obj
      B₁ t = B₀ {t} (s ≟ t)
      B₁s≡B : B₁ s ≡ B
      B₁s≡B = withDec-contract {d = s ≟ s} {T = T} {E = E s}
      B₂ : Dec (s ≡ s) → Obj
      B₂ = B₀ {s}
      B₂x≡B : (x : Dec (s ≡ s)) → B₂ x ≡ B   -- [ WK: unused ]
      B₂x≡B x = cong B₂ (Dec-x≡x-irrelevance {d = x} {e = yes refl})
      B≡B₂s≟s : B ≡ B₂ (s ≟ s)               -- [ WK: unused ]
      B≡B₂s≟s = sym (B₂x≡B (s ≟ s))
      TT : (t : Sort) → s ≡ t → P t B
      TT = λ t s≡t → subst (λ u → P u B) s≡t R
      EE : (t : Sort) → ¬ (s ≡ t) → P t (A t)
      EE = λ t s≢t → p t
      R₁ : (t : Sort) → P t (B₁ t)
      R₁ = λ t → withDec-subst (s ≟ t) {P = P t} T (E t) (TT t) (EE t)
      M : Dec (s ≡ s) → Set j
      M = λ x → P s (B₂ x)
      DI : (x : Dec (s ≡ s)) → (s ≟ s) ≡ x
      DI x = Dec-x≡x-irrelevance {d = s ≟ s} {e = x}
      DI₁ : (s ≟ s) ≡ yes refl
      DI₁ = DI (yes refl)
      R₁-contract₁ : R₁ s ≡ subst (P s) (sym B₁s≡B) R
      R₁-contract₁ = withDec-subst-contract {d = s ≟ s}
                {P = P s} {T = T} {E = E s} {t = TT s} {e = EE s}
      R₁'-contract : subst M {s ≟ s} {yes refl} DI₁ (R₁ s) ≡ R
      R₁'-contract = let open ≡-Reasoning in begin
                subst M {s ≟ s} {yes refl} DI₁ (R₁ s)
        ≡⟨ cong (subst M {s ≟ s} {yes refl} DI₁) R₁-contract₁ ⟩
                subst M {s ≟ s} {yes refl} DI₁
                    (subst (P s) {B} {B₁ s} (sym B₁s≡B) R)
        ≡⟨ ≡-subst-comp B₂ (P s) {s ≟ s} {yes refl} {B} DI₁ (sym B₁s≡B) R ⟩
                subst (P s) {B} {B₂ (yes refl)}
```

```
                      (trans (sym B₁s≡B) (cong B₂ DI₁)) R
        ≡⟨ ≡-subst-contract (P s) (trans (sym B₁s≡B) (cong B₂ DI₁)) R ⟩
               R
        ■
      claim : (B : Obj) → (R : P s B) → Set k₂
      claim B R = p s ⨾ R ⊑ R
    in
        -- the type signatures "∋" in the following are only for documentation.
    (claim B R) ∋
    (subst (claim B) R₁'-contract
      ((claim (B₂ (yes refl)) (subst M {s ≟ s} {yes refl} DI₁ (R₁ s))) ∋
        (subst (λ (x : Dec (s ≡ s)) → claim (B₂ x) (subst M (DI x) (R₁ s)))
               {s ≟ s} {yes refl} DI₁
               ((claim (B₂ (s ≟ s)) (subst M (DI (s ≟ s)) (R₁ s))) ∋
                 (subst (claim (B₂ (s ≟ s)))
                   (sym (≡-subst-contract M {s ≟ s} (DI (s ≟ s)) (R₁ s)))
                   ((claim (B₂ (s ≟ s)) (R₁ s)) ∋ (subid {B₁} {R₁} s))
               ))
        ))
      )
```

For right subidentities, we obtain the corresponding property via duality:

```
private
  module SIPsubidReflectR
    (Sort : Set) (_≟_ : Decidable (_≡_ {A = Sort}))
    {i j k₁ k₂ : Level} {Obj : Set i}
    (base : OrderedSemigroupoid j k₁ k₂ Obj)
    where
      open SGSIP Sort (OrderedSemigroupoid.semigroupoid base) using (SIPObj)
      open SIPsubidReflectL Sort _≟_ base using (OSG; SIPisLeftSubidReflect)
      open SIPsubidReflectL Sort _≟_ (oppositeOrderedSemigroupoid base) public using ()
        renaming
          (SIPisLeftSubidReflect to
            SIPisRightSubidReflect   -- : {A : SIPObj} {p : SIPMor A A}
                                     -- → OrderedSemigroupoid.isRightSubidentity OSG p
                                     -- → (s : Sort)
                                     -- → OrderedSemigroupoid.isRightSubidentity base (p s)
          )
      SIPisSubidReflect : {A : SIPObj}
          {p : OrderedSemigroupoid.Mor OSG A A}
        → OrderedSemigroupoid.isSubidentity OSG p
        → (s : Sort)
        → OrderedSemigroupoid.isSubidentity base (p s)
      SIPisSubidReflect subid s = (λ {B} {R} → SIPisLeftSubidReflect (proj₁ subid) s)
                                , (λ {B} {R} → SIPisRightSubidReflect (proj₂ subid) s)

open SIPsubidReflectL public using (SIPisLeftSubidReflect)
open SIPsubidReflectR public using (SIPisRightSubidReflect; SIPisSubidReflect)
```

23.33 Categoric.SortIndexedProduct.OSGD

For all components of a SIP subidentity to be subidentities again, we need a decidable propositional equality on Sort, see Sect. 23.32.

```
SIPdomainOP : (Sort : Set) (_≟_ : Decidable (_≡_ {A = Sort}))
                  {i j k₁ k₂ : Level} {Obj : Set i}
                  {base : OrderedSemigroupoid j k₁ k₂ Obj}
               → OSGDomainOp base
               → OSGDomainOp (SIPOrderedSemigroupoid Sort base)
SIPdomainOP Sort _≟_ {Obj = Obj} {base = base} domainOp = let
  OSG = SIPOrderedSemigroupoid Sort base
  ObjSIP = Sort → Obj
  MorSIP = OrderedSemigroupoid.Mor OSG
  open OSGDomainOp domainOp
  open OrderedSemigroupoid base
  domSIP : {A B : ObjSIP} → (R : MorSIP A B) → MorSIP A A
  domSIP = λ {A} {B} R s → dom (R s)
  in record
    {dom                = domSIP   -- λ {A} {B} R s → dom (R s)
    ; domSubIdentity    = (λ s → proj₁ domSubIdentity)
                        , (λ s → proj₂ domSubIdentity)
    ; dom-⨾-idempotent  = λ s → dom-⨾-idempotent
    ; domPreserves⊑     = λ Q⊑R s → domPreserves⊑ (Q⊑R s)
    ; domLeastPreserver = λ subid idem R⊑d⨾R s
                        → domLeastPreserver (SIPisSubidReflect Sort _≟_ base subid s) (idem s) (R⊑d⨾R s)
    ; domLocality       = λ s → domLocality
    }


SIPOSGD : (Sort : Set) (_≟_ : Decidable (_≡_ {A = Sort}))
          → {i j k₁ k₂ : Level} {Obj : Set i}
          → OSGD {i} j k₁ k₂ Obj
          → OSGD {i} j k₁ k₂ (Sort → Obj)
SIPOSGD Sort _≟_ Base = let open OSGD Base in record
  {orderedSemigroupoid = SIPOrderedSemigroupoid Sort orderedSemigroupoid
  ; domainOp           = SIPdomainOP          Sort _≟_ domainOp
  }


SIPrangeOP : (Sort : Set) (_≟_ : Decidable (_≡_ {A = Sort}))
                 {i j k₁ k₂ : Level} {Obj : Set i}
                 {base : OrderedSemigroupoid j k₁ k₂ Obj}
             → OSGRangeOp base
             → OSGRangeOp (SIPOrderedSemigroupoid Sort base)
SIPrangeOP Sort _≟_ {Obj = Obj} {base = base} rangeOp = let
  OSG = SIPOrderedSemigroupoid Sort base
  ObjSIP = Sort → Obj
  MorSIP = OrderedSemigroupoid.Mor OSG
  open OSGRangeOp rangeOp
  open OrderedSemigroupoid base
  ranSIP : {A B : ObjSIP} → (R : MorSIP A B) → MorSIP B B
  ranSIP = λ {A} {B} R s → ran (R s)
  in record
    {ran                = ranSIP   -- λ {A} {B} R s → ran (R s)
    ; ranSubIdentity    = (λ s → proj₁ ranSubIdentity)
                        , (λ s → proj₂ ranSubIdentity)
    ; ran-⨾-idempotent  = λ s → ran-⨾-idempotent
    ; ranPreserves⊑     = λ Q⊑R s → ranPreserves⊑ (Q⊑R s)
    ; ranLeastPreserver = λ subid idem R⊑R⨾d s
                        → ranLeastPreserver (SIPisSubidReflect Sort _≟_ base subid s) (idem s) (R⊑R⨾d s)
    ; ranLocality       = λ s → ranLocality
    }
```

```
SIPOSGR : (Sort : Set) (_≟_ : Decidable (_≡_ {A = Sort}))
        → {i j k₁ k₂ : Level} {Obj : Set i}
        → OSGR {i} j k₁ k₂ Obj
        → OSGR {i} j k₁ k₂ (Sort → Obj)
SIPOSGR Sort _≟_ Base = let open OSGR Base in record
  {orderedSemigroupoid = SIPOrderedSemigroupoid Sort orderedSemigroupoid
  ; rangeOp            = SIPrangeOP             Sort _≟_ rangeOp
  }

SIPOSGDR : (Sort : Set) (_≟_ : Decidable (_≡_ {A = Sort}))
         → {i j k₁ k₂ : Level} {Obj : Set i}
         → OSGDR {i} j k₁ k₂ Obj
         → OSGDR {i} j k₁ k₂ (Sort → Obj)
SIPOSGDR Sort _≟_ Base = let open OSGDR Base in record
  {orderedSemigroupoid = SIPOrderedSemigroupoid Sort orderedSemigroupoid
  ; domainOp           = SIPdomainOP            Sort _≟_ domainOp
  ; rangeOp            = SIPrangeOP             Sort _≟_ rangeOp
  }
```

23.34 Categoric.SortIndexedProduct.SemiAllegory

```
SIPSemiAllegory : (Sort : Set) (_≟_ : Decidable (_≡_ {A = Sort}))
                  {i j k₁ k₂ : Level} {Obj : Set i}
                → SemiAllegory {i} j k₁ k₂ Obj
                → SemiAllegory {i} j k₁ k₂ (Sort → Obj)
SIPSemiAllegory Sort _≟_ Base = let open SemiAllegory Base in record
  {osgc      = SIPOSGC      Sort osgc
  ; meetOp   = SIPMeetOp    Sort meetOp
  ; domainOp = SIPdomainOP Sort _≟_ domainOp
  ; Dedekind = λ s → Dedekind
  }
```

23.35 Categoric.SortIndexedProduct.SemiCollagory

```
SIPSemiCollagory : (Sort : Set) (_≟_ : Decidable (_≡_ {A = Sort}))
                   {i j k₁ k₂ : Level} {Obj : Set i}
                 → SemiCollagory {i} j k₁ k₂ Obj
                 → SemiCollagory {i} j k₁ k₂ (Sort → Obj)
SIPSemiCollagory Sort _≟_ Base = let open SemiCollagory Base in record
  {semiAllegory     = SIPSemiAllegory      Sort _≟_ semiAllegory
  ; joinOp          = SIPJoinOp            Sort joinOp
  ; homLatDistr     = SIPHomLatticeDistr Sort homLatDistr
  ; joinCompDistrL  = SIPJoinCompDistrL  Sort joinCompDistrL
  ; joinCompDistrR  = SIPJoinCompDistrR Sort joinCompDistrR
  }
```

23.36 Categoric.SortIndexedProduct.LeftRestrResOp

```
SIPLeftRestrResOp : (Sort : Set) (_≟_ : Decidable (_≡_ {A = Sort}))
                  → {i j k₁ k₂ : Level} {Obj : Set i}
```

```
                              → {base : OSGDR {i} j k₁ k₂ Obj}
                              → (leftRestrResOp : LeftRestrResOp base)
                              → LeftRestrResOp (SIPOSGDR Sort _≟_ base)
SIPLeftRestrResOp Sort _≟_ leftRestrResOp = let open LeftRestrResOp leftRestrResOp in record
  {_⫽_ = λ S R s → S s ⫽ R s
  ; ⫽-cancel-outer = λ s → ⫽-cancel-outer
  ; ⫽-restr         = λ s → ⫽-restr
  ; ⫽-universal     = λ Q⨾R⊑S ranQ⊑domR s → ⫽-universal (Q⨾R⊑S s) (ranQ⊑domR s)
  }
```

23.37 Categoric.SortIndexedProduct.RightRestrResOp

```
SIPRightRestrResOp : (Sort : Set) (_≟_ : Decidable (_≡_ {A = Sort}))
                               → {i j k₁ k₂ : Level} {Obj : Set i}
                               → {base : OSGDR {i} j k₁ k₂ Obj}
                               → (rightRestrResOp : RightRestrResOp base)
                               → RightRestrResOp (SIPOSGDR Sort _≟_ base)
SIPRightRestrResOp Sort _≟_ rightRestrResOp = let open RightRestrResOp rightRestrResOp in record
  {_⑊_ = λ Q S s → Q s ⑊ S s
  ; ⑊-cancel-outer = λ s → ⑊-cancel-outer
  ; ⑊-restr         = λ s → ⑊-restr
  ; ⑊-universal     = λ Q⨾R⊑S domR⊑ranQ s → ⑊-universal (Q⨾R⊑S s) (domR⊑ranQ s)
  }
```

Bibliography

Rudolf Berghammer, Gunther Schmidt, and Hans Zierer. Symmetric quotients. Technical Report TUM-INFO 8620, Technische Universität München, Fakultät für Informatik, 1986. 18 p.

Rudolf Berghammer, Gunther Schmidt, and Hans Zierer. Symmetric quotients and domain constructions. *Inform. Process. Lett.*, 33(3):163–168, 1989.

Rudolf Berghammer, Ali Jaoua, and Bernhard Möller, editors. *Relations and Kleene Algebra in Computer Science — 11th International Conference on Relational Methods in Computer Science, and 6th International Conference on Applications of Kleene Algebra, RelMiCS/AKA 2009, Doha, Qatar, November 1–5, 2009. Proceedings*, volume 5827 of *LNCS*, 2009. Springer. doi: 10.1007/978-3-642-04639-1.

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

Nils Anders Danielsson et al. Agda standard library, version 0.7, January 2013. `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Libraries.StandardLibrary`.

Jules Desharnais and Bernhard Möller. Characterizing determinacy in Kleene algebras. *Information Sciences*, 139: 253–273, 2001.

Jules Desharnais, Bernhard Möller, and Georg Struth. Kleene algebra with domain. *ACM Transactions on Computational Logic*, 7(4):798–833, 2006.

Jules Desharnais, Peter Jipsen, and Georg Struth. Domain and antidomain semigroups. In Berghammer et al. (2009), pages 73–87. doi: 10.1007/978-3-642-04639-1.

Dan Dougherty and Claudio Gutiérrez. Normal forms and reduction for theories of binary relations. In Leo Bachmair, editor, *Rewriting Techniques and Applications, Proc. RTA 2000*, volume 1833 of *LNCS*, pages 95–109. Springer, 2000.

Peter J. Freyd and Andre Scedrov. *Categories, Allegories*, volume 39 of *North-Holland Mathematical Library*. North-Holland, Amsterdam, 1990. ISBN 0-444-70368-3 and 0-444-70367-5 (pbk).

Hitoshi Furusawa and Wolfram Kahl. A study on symmetric quotients. Technical Report 1998-06, Fakultät für Informatik, Universität der Bundeswehr München, December 1998.

Carlos Gonzalía. *Relations in Dependent Type Theory*. Ph.D. thesis, also as Technical Report No. 14D, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg University, 2006.

Jinrong Han. Proofs of relational semigroupoids in Isabelle/Isar. M.Sc. thesis, McMaster University, Department of Computing and Software, 2008.

Gérard Huet and Amokrane Saïbi. Constructive category theory. In *Proceedings of the Joint CLICS-TYPES Workshop on Categories and Type Theory*, 1998. doi: 10.1.1.39.4193.

Gérard Huet and Amokrane Saïbi. Constructive category theory. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, language, and interaction: Essays in honour of Robin Milner*, Foundations Of Computing Series, pages 239–275. MIT Press, 2000. ISBN 0-262-16188-5.

Wolfram Kahl. Refactoring heterogeneous relation algebras around ordered categories and converse. *J. Relational Methods in Comp. Sci.*, 1:277–313, 2004. URL `http://www.jormics.org/`.

Wolfram Kahl. Semigroupoid interfaces for programming with relations in Haskell. In Renate Schmidt and Georg Struth, editors, *Relations and Kleene Algebra in Computer Science, RelMiCS/AKA 2006*, volume 4136 of *LNCS*, pages 235–250. Springer, 2006. doi: http://dx.doi.org/10.1007/11828563_16. URL `http://link.springer.de/link/service/series/0558/tocs/t4136.htm`.

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. Collagories for relational adhesive rewriting. In Berghammer et al. (2009), pages 211–226. doi: 10.1007/978-3-642-04639-1.

Wolfram Kahl. Determinisation of relational substitutions in ordered categories with domain. *J. Logic and Algebraic Programming*, 79:812–829, 2010. doi: 10.1016/j.jlap.2010.07.017.

Wolfram Kahl. Collagories: Relation-algebraic reasoning for gluing constructions. *J. Logic and Algebraic Programming*, 80(6):297–338, 2011a. doi: 10.1016/j.jlap.2011.04.006.

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, 2011b. 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.

Akira Kanda. Constructive category theory (no. 1). In Jozef Gruska and Michal Chytil, editors, *Mathematical Foundations of Computer Science, MFCS '81*, volume 118 of *LNCS*, pages 563–577. Springer, 1981. ISBN 3-540-10856-4. doi: 10.1007/3-540-10856-4_125.

Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Inform. and Comput.*, 110(2):366–390, 1994a.

Dexter Kozen. On action algebras. In J. van Eijck and A. Visser, editors, *Logic and Information Flow*, pages 78–88. MIT Press, 1994b.

Dexter Kozen. Typed Kleene algebra. Technical Report 98-1669, Computer Science Department, Cornell University, March 1998.

Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

Shin-Cheng Mu, Hsiang-Shang Ko, and Patrik Jansson. Algebra of programming in Agda: Dependent types for relational program derivation. *J. Functional Programming*, 19(5):545–579, September 2009. doi: 10.1017/S0956796809007345. See also AoPA at `http://www.iis.sinica.edu.tw/~scm/2008/aopa/`.

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.

Vaughan Pratt. Action logic and pure induction. In J. van Eijck, editor, *JELIA 1990: Proc. European Workshop on Logics in Artificial Intelligence*, volume 478 of *LNCS*, pages 97–120. Springer, 1991.

Gunther Schmidt and Thomas Ströhlein. *Relations and Graphs, Discrete Mathematics for Computer Scientists*. EATCS-Monographs on Theoret. Comput. Sci. Springer, 1993. ISBN 3-540-56254-0, 0-387-56254-0.

Gunther Schmidt, Claudia Hattensperger, and Michael Winter. Heterogeneous relation algebra. In Chris Brink, Wolfram Kahl, and Gunther Schmidt, editors, *Relational Methods in Computer Science*, Advances in Computing Science, chapter 3, pages 39–53. Springer, Wien, New York, 1997.

J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science. Prentice Hall, 1989. URL http://spivey.oriel.ox.ac.uk/mike/zrm/. Out of print; available via URL: http://spivey.oriel.ox.ac.uk/mike/zrm/.

Hans Zierer. Relation-algebraic domain constructions. *Theoretical Computer Science*, 87:163–188, 1991.