Theories and Data Structures

"Two-Sides of the Same Coin", or "Library Design by Adjunction"

Jacques Carette, Musa Al-hassy, Wolfram Kahl

August 28, 2017

Abstract

We aim to show how common data-structures naturally arise from elementary mathematical theories.

In particular, we answer the following questions:

- Why do lists pop-up more frequently to the average programmer than, say, their duals: bags?
- More simply, why do unit and empty types occur so naturally? What about enumerations/sums and records/products?
- Why is it that dependent sums and products do not pop-up expicitly to the average programmer? They arise naturally all the time as tuples and as classes.
- How do we get the usual toolbox of functions and helpful combinators for a particular data type? Are they "built into" the type?
- Is it that the average programmer works in the category of classical Sets, with functions and propositional equality? Does this result in some "free constructions" not easily made computable since mathematicians usually work in the category of Setoids but tend to quotient to arrive in Sets? —where quotienting is not computably feasible, in Sets at-least; and why is that?

$\boxed{\textbf{???}}$

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

Contents

1 Introduction

???

2 Overview

???

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

https://github.com/JacquesCarette/TheoriesAndDataStructures

Part I

Helpers

3 Obtaining Forgetful Functors

We aim to realise a "toolkit" for an data-structure by considering a free construction and proving it adjoint to a forgetful functor. Since the majority of our theories are built on the category Set, we begin my making a helper method to produce the forgetful functors from as little information as needed about the mathematical structure being studied.

Indeed, it is a common scenario where we have an algebraic structure with a single carrier set and we are interested in the categories of such structures along with functions preserving the structure.

We consider a type of "algebras" built upon the category of Sets —in that, every algebra has a carrier set and every homomorphism is a essentially a function between carrier sets where the composition of homomorphisms is essentially the composition of functions and the identity homomorphism is essentially the identity function.

Such algebras consistute a category from which we obtain a method to Forgetful functor builder for single-sorted algebras to Sets.

```
module Forget where
open import Level

open import Categories.Category using (Category)
open import Categories.Functor  using (Functor)
open import Categories.Agda     using (Sets)

open import Function2
open import Function
open import EqualityCombinators
```

[MA: *For one reason or another, the module head is not making the imports smaller.* **]**

A OneSortedAlg is essentially the details of a forgetful functor from some category to Sets,

```
record OneSortedAlg (ℓ : Level) : Set (suc (suc ℓ)) where
  field
    Alg      : Set (suc ℓ)
    Carrier  : Alg → Set ℓ
    Hom      : Alg → Alg → Set ℓ
```

```
mor          : {A B : Alg} → Hom A B → (Carrier A → Carrier B)
comp         : {A B C : Alg} → Hom B C → Hom A B → Hom A C
.comp-is-∘   : {A B C : Alg} {g : Hom B C} {f : Hom A B} → mor (comp g f) ≐ mor g ∘ mor f
Id           : {A : Alg} → Hom A A
.Id-is-id    : {A : Alg} → mor (Id {A}) ≐ id
```

The aforementioned claim that algebras and their structure preserving morphisms form a category can be realised due to the coherency conditions we requested viz the morphism operation on homomorphisms is functorial.

```
open import Relation.Binary.SetoidReasoning
oneSortedCategory : (ℓ : Level) → OneSortedAlg ℓ → Category (suc ℓ) ℓ ℓ
oneSortedCategory ℓ A = record
  {Obj   = Alg
  ; _⇒_  = Hom
  ; _≡_  = λ F G → mor F ≐ mor G
  ; id   = Id
  ; _∘_  = comp
  ; assoc = λ {A B C D} {F} {G} {H} → begin⟨ ≐-setoid (Carrier A) (Carrier D) ⟩
    mor (comp (comp H G) F) ≈⟨ comp-is-∘                          ⟩
    mor (comp H G) ∘ mor F  ≈⟨ ∘-≐-cong₁ _ comp-is-∘              ⟩
    mor H ∘ mor G ∘ mor F   ≈˘⟨ ∘-≐-cong₂ (mor H) comp-is-∘ ⟩
    mor H ∘ mor (comp G F)  ≈˘⟨ comp-is-∘                         ⟩
    mor (comp H (comp G F)) ∎
  ; identityˡ = λ {{f = f}} → comp-is-∘ ⟨≐≐⟩ Id-is-id ∘ mor f}
  ; identityʳ = λ {{f = f}} → comp-is-∘ ⟨≐≐⟩ ≡.cong (mor f) ∘ Id-is-id}
  ; equiv    = record {IsEquivalence ≐-isEquivalence}
  ; ∘-resp-≡  = λ f≈h g≈k → comp-is-∘ ⟨≐≐⟩ ∘-resp-≐ f≈h g≈k ⟨≐≐⟩ ≐-sym comp-is-∘
  }
  where open OneSortedAlg A; open import Relation.Binary using (IsEquivalence)
```

The fact that the algebras are built on the category of sets is captured by the existence of a forgetful functor.

```
mkForgetful : (ℓ : Level) (A : OneSortedAlg ℓ) → Functor (oneSortedCategory ℓ A) (Sets ℓ)
mkForgetful ℓ A = record
  {F₀             = Carrier
  ; F₁            = mor
  ; identity      = Id-is-id $ᵢ
  ; homomorphism  = comp-is-∘ $ᵢ
  ; F-resp-≡      = _$ᵢ
  }
  where open OneSortedAlg A
```

That is, the constituents of a OneSortedAlgebra suffice to produce a category and a so-called presheaf as well.

4 Equality Combinators

Here we export all equality related concepts, including those for propositional and function extensional equality.

```
module EqualityCombinators where
open import Level
```

4.1 Propositional Equality

We use one of Agda's features to qualify all propositional equality properties by "≡." for the sake of clarity and to avoid name clashes with similar other properties.

```
import Relation.Binary.PropositionalEquality
module ≡ = Relation.Binary.PropositionalEquality
open ≡ using (_≡_) public
```

We also provide two handy-dandy combinators for common uses of transitivity proofs.

```
_⟨≡≡⟩_ = ≡.trans
_⟨≡≡˘⟩_ : {a : Level} {A : Set a} {x y z : A} → x ≡ y → z ≡ y → x ≡ z
x≈y ⟨≡≡˘⟩ z≈y = x≈y ⟨≡≡⟩ ≡.sym z≈y
```

4.2 Function Extensionality

We bring into scope pointwise equality, _≐_, and provide a proof that it constitutes an equivalence relation —where the source and target of the functions being compared are left implicit.

```
open ≡ using () renaming (_→-setoid_ to ≐-setoid; _≗_ to _≐_) public
open import Relation.Binary using (IsEquivalence; Setoid)
module _ {a b : Level} {A : Set a} {B : Set b} where
  ≐-isEquivalence : IsEquivalence (_≐_ {A = A} {B})
  ≐-isEquivalence = record {Setoid (≐-setoid A B)}
  open IsEquivalence ≐-isEquivalence public
    renaming (refl to ≐-refl; sym to ≐-sym; trans to ≐-trans)
  open import Equiv public using (∘-resp-≐)   -- To do: port this over here!
    renaming (cong∘l to ∘-≐-cong₂; cong∘r to ∘-≐-cong₁)
infixr 5 _⟨≐≐⟩_
_⟨≐≐⟩_ = ≐-trans
```

Note that the precedence of this last operator is lower than that of function composition so as to avoid superfluous parenthesis.

Here is an implicit version of extensional —we use it as a transitionary tool since the standard library and the category theory library differ on their uses of implicit versus explicit variable usage.

```
infixr 5 _≐ᵢ_
_≐ᵢ_ : {a b : Level} {A : Set a} {B : A → Set b}
  (f g : (x : A) → B x) → Set (a ⊔ b)
f ≐ᵢ g = ∀ {x} → f x ≡ g x
```

4.3 Equiv

We form some combinators for HoTT like reasoning.

```
cong₂D : ∀ {a b c} {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}
  → (x₂≡x₁ : x₂ ≡ x₁) → ≡.subst B x₂≡x₁ y₂ ≡ y₁ → f x₁ y₁ ≡ f x₂ y₂
cong₂D f ≡.refl ≡.refl = ≡.refl
open import Equiv public using (_≃_; id≃; sym≃; trans≃; qinv)
infix 3 _□
infixr 2 _≃⟨_⟩_
_≃⟨_⟩_ : {x y z : Level} (X : Set x) {Y : Set y} {Z : Set z}
```

```
  → X ≃ Y → Y ≃ Z → X ≃ Z
X ≃⟨ X≃Y ⟩ Y≃Z = trans≃ X≃Y Y≃Z
_□ : {x : Level} (X : Set x) → X ≃ X
X □ = id≃
```

[**MA:** *Consider moving pertinent material here from Equiv.lagda at the end.*]

4.4 Making symmetry calls less intrusive

It is common that we want to use an equality within a calculation as a right-to-left rewrite rule which is accomplished by utilizing its symmetry property. We simplify this rendition, thereby saving an explicit call and parenthesis in-favour of a less hinder-some notation.

Among other places, I want to use this combinator in module Forget's proof of associativity for oneSortedCategory

```
module _ {c l : Level} {S : Setoid c l} where
  open import Relation.Binary.SetoidReasoning using (_≈⟨_⟩_)
  open import Relation.Binary.EqReasoning using (_IsRelatedTo_)
  open Setoid S
  infixr 2 _≈˘⟨_⟩_
  _≈˘⟨_⟩_ : ∀ (x {y z} : Carrier) → y ≈ x → _IsRelatedTo_ S y z → _IsRelatedTo_ S x z
  x ≈˘⟨ y≈x ⟩ y≈z = x ≈⟨ sym y≈x ⟩ y≈z
```

A host of similar such combinators can be found within the RATH-Agda library.

4.5 More Equational Reasoning for Setoid

A few convenient combinators for equational reasoning in Setoid.

```
module SetoidCombinators {ℓS ℓs : Level} (S : Setoid ℓS ℓs) where
  open Setoid S
  _⟨≈≈⟩_ = trans
  _⟨≈˘≈⟩_ : {a b c : Carrier} → b ≈ a → b ≈ c → a ≈ c
  _⟨≈˘≈⟩_ = λ b≈a b≈c → sym b≈a ⟨≈≈⟩ b≈c
  _⟨≈≈˘⟩_ : {a b c : Carrier} → a ≈ b → c ≈ b → a ≈ c
  _⟨≈≈˘⟩_ = λ a≈b c≈b → a≈b ⟨≈≈⟩ sym c≈b
  _⟨≈˘≈˘⟩_ : {a b c : Carrier} → b ≈ a → c ≈ b → a ≈ c
  _⟨≈˘≈˘⟩_ = λ b≈a c≈b → b≈a ⟨≈˘≈⟩ sym c≈b
```

4.6 Localising Equality

Convenient syntax for when we need to specify which Setoid's equality we are talking about.

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

5 Properties of Sums and Products

This module is for those domain-ubiquitous properties that, disappointingly, we could not locate in the standard library. —The standard library needs some sort of "table of contents *with* subsection" to make it easier to know of what is available.

This module re-exports (some of) the contents of the standard library's Data.Product and Data.Sum.

```
module DataProperties where

open import Level renaming (suc to lsuc; zero to lzero)
open import Function using (id; _∘_; const)
open import EqualityCombinators

open import Data.Product public using (_×_; proj₁; proj₂; Σ; _,_; swap; uncurry) renaming (map to _×₁_; <_,_> to ⟨_,_⟩)
open import Data.Sum public using (inj₁; inj₂; [_,_]) renaming (map to _⊎₁_)
open import Data.Nat using (ℕ; zero; suc)
```

Precedence Levels

The standard library assigns precedence level of 1 for the infix operator _⊎_, which is rather odd since infix operators ought to have higher precedence that equality combinators, yet the standard library assigns _≈⟨_⟩_ a precedence level of 2. The usage of these two —e.g. in CommMonoid.lagda— causes an annoying number of parentheses and so we reassign the level of the infix operator to avoid such a situation.

```
infixr 3 _⊎_
_⊎_ = Data.Sum._⊎_
```

5.1 Generalised Bot and Top

To avoid a flurry of lift's, and for the sake of clarity, we define level-polymorphic empty and unit types.

```
open import Level

data ⊥ {ℓ : Level} : Set ℓ where

⊥-elim : {a ℓ : Level} {A : Set a} → ⊥ {ℓ} → A
⊥-elim ()

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

5.2 Sums

Just as _⊎_ takes types to types, its "map" variant _⊎₁_ takes functions to functions and is a functorial congruence: It preserves identity, distributes over composition, and preserves extenstional equality.

```
⊎-id : {a b : Level} {A : Set a} {B : Set b} → id ⊎₁ id ≐ id {A = A ⊎ B}
⊎-id = [ ≐-refl , ≐-refl ]

⊎-∘ : {a b c a' b' c' : Level}
  {A : Set a} {A' : Set a'} {B : Set b} {B' : Set b'} {C' : Set c} {C : Set c'}
  {f : A → A'} {g : B → B'} {f' : A' → C} {g' : B' → C'}
  → (f' ∘ f) ⊎₁ (g' ∘ g) ≐ (f' ⊎₁ g') ∘ (f ⊎₁ g)    -- aka "the exchange rule for sums"
⊎-∘ = [ ≐-refl , ≐-refl ]

⊎-cong : {a b c d : Level} {A : Set a} {B : Set b} {C : Set c} {D : Set d} {f f' : A → C} {g g' : B → D}
```

```
  → f ≐ f' → g ≐ g' → f ⊎₁ g ≐ f' ⊎₁ g'
⊎-cong f≈f' g≈g' = [ ∘-≐-cong₂ inj₁ f≈f' , ∘-≐-cong₂ inj₂ g≈g' ]
```

Composition post-distributes into casing,

```
∘-[,] : {a b c d : Level} {A : Set a} {B : Set b} {C : Set c} {D : Set d} {f : A → C} {g : B → C} {h : C → D}
  → h ∘ [ f , g ] ≐ [ h ∘ f , h ∘ g ]  -- aka "fusion"
∘-[,] = [ ≐-refl , ≐-refl ]
```

It is common that a data-type constructor D : Set → Set allows us to extract elements of the underlying type and so we have a natural transfomation D ⟶ **I**, where **I** is the identity functor. These kind of results will occur for our other simple data-strcutres as well. In particular, this is the case for D A = 2 × A = A ⊎ A:

```
from⊎ : {ℓ : Level} {A : Set ℓ} → A ⊎ A → A
from⊎ = [ id , id ]
  -- from⊎ is a natural transformation
  --
from⊎-nat : {a b : Level} {A : Set a} {B : Set b} {f : A → B} → f ∘ from⊎ ≐ from⊎ ∘ (f ⊎₁ f)
from⊎-nat = [ ≐-refl , ≐-refl ]
  -- from⊎ is injective and so is pre-invertible,
  --
from⊎-preInverse : {a b : Level} {A : Set a} {B : Set b} → id ≐ from⊎ {A = A ⊎ B} ∘ (inj₁ ⊎₁ inj₂)
from⊎-preInverse = [ ≐-refl , ≐-refl ]
```

[**MA: insert:** A brief mention about co-monads?]

5.3 Products

Dual to from⊎, a natural transformation 2 ×_ ⟶ **I**, is diag, the transformation **I** ⟶ $\_^2$.

```
diag : {ℓ : Level} {A : Set ℓ} (a : A) → A × A
diag a = a , a
```

[**MA: insert:** A brief mention of Haskell's const, which is diag curried. Also something about K combinator?]

Z-style notation for sums,

```
Σ:• : {a b : Level} (A : Set a) (B : A → Set b) → Set (a ⊔ b)
Σ:• = Data.Product.Σ
infix -666 Σ:•
syntax Σ:• A (λ x → B) = Σ x : A • B


open import Data.Nat.Properties
suc-inj : ∀ {i j} → ℕ.suc i ≡ ℕ.suc j → i ≡ j
suc-inj = cancel-+-left (ℕ.suc ℕ.zero)
```

or

```
suc-inj {0} _≡_.refl = _≡_.refl
suc-inj {ℕ.suc i} _≡_.refl = _≡_.refl
```

Part II

Variations on Sets

6 Two Sorted Structures

So far we have been considering algebraic structures with only one underlying carrier set, however programmers are faced with a variety of different types at the same time, and the graph structure between them, and so we consider briefly consider two sorted structures by starting the simplest possible case: Two type and no required interaction whatsoever between them.

```
module Structures.TwoSorted where

open import Level renaming (suc to lsuc; zero to lzero)
open import Categories.Category    using (Category)
open import Categories.Functor     using (Functor)
open import Categories.Adjunction using (Adjunction)
open import Categories.Agda        using (Sets)
open import Function               using (id; _∘_; const)
open import Function2              using (_$ᵢ)

open import Forget
open import EqualityCombinators
open import DataProperties
```

6.1 Definitions

A TwoSorted type is just a pair of sets in the same universe —in the future, we may consider those in different levels.

```
record TwoSorted ℓ : Set (lsuc ℓ) where
  constructor MkTwo
  field
    One : Set ℓ
    Two : Set ℓ

open TwoSorted
```

Unastionishngly, a morphism between such types is a pair of functions between the *multiple* underlying carriers.

```
record Hom {ℓ} (Src Tgt : TwoSorted ℓ) : Set ℓ where
  constructor MkHom
  field
    one : One Src → One Tgt
    two : Two Src → Two Tgt

open Hom
```

6.2 Category and Forgetful Functors

We are using pairs of object and pairs of morphisms which are known to form a category:

```
Twos : (ℓ : Level) → Category (lsuc ℓ) ℓ ℓ
Twos ℓ = record
```

```
  {Obj        = TwoSorted ℓ
  ; _⇒_       = Hom
  ; _≡_       = λ F G → one F ≐ one G × two F ≐ two G
  ; id        = MkHom id id
  ; _∘_       = λ F G → MkHom (one F ∘ one G) (two F ∘ two G)
  ; assoc     = ≐-refl , ≐-refl
  ; identityˡ = ≐-refl , ≐-refl
  ; identityʳ = ≐-refl , ≐-refl
  ; equiv     = record
     {refl  = ≐-refl , ≐-refl
     ; sym  = λ {(oneEq , twoEq) → ≐-sym oneEq , ≐-sym twoEq}
     ; trans = λ {(oneEq₁ , twoEq₁) (oneEq₂ , twoEq₂) → ≐-trans oneEq₁ oneEq₂ , ≐-trans twoEq₁ twoEq₂}
     }
  ; ∘-resp-≡ = λ {(g≈₁k , g≈₂k) (f≈₁h , f≈₂h) → ∘-resp-≐ g≈₁k f≈₁h , ∘-resp-≐ g≈₂k f≈₂h}
  }
```

The naming Twos is to be consistent with the category theory library we are using, which names the category of sets and functions by Sets.

We can forget about the first sort or the second to arrive at our starting category and so we have two forgetful functors.

```
Forget : (ℓ : Level) → Functor (Twos ℓ) (Sets ℓ)
Forget ℓ = record
  {F₀             = TwoSorted.One
  ; F₁            = Hom.one
  ; identity      = ≡.refl
  ; homomorphism  = ≡.refl
  ; F-resp-≡ = λ {(F≈₁G , F≈₂G) {x} → F≈₁G x}
  }
Forget² : (ℓ : Level) → Functor (Twos ℓ) (Sets ℓ)
Forget² ℓ = record
  {F₀             = TwoSorted.Two
  ; F₁            = Hom.two
  ; identity      = ≡.refl
  ; homomorphism  = ≡.refl
  ; F-resp-≡ = λ {(F≈₁G , F≈₂G) {x} → F≈₂G x}
  }
```

6.3 Free and CoFree

Given a type, we can pair it with the empty type or the singelton type and so we have a free and a co-free constructions. Intuitively, the first is free since the singelton type is the smallest type we can adjoin to obtain a Twos object, whereas ⊤ is the "largest" type we adjoin to obtain a Twos object. This is one way that the unit and empty types naturaly arise.

```
Free : (ℓ : Level) → Functor (Sets ℓ) (Twos ℓ)
Free ℓ = record
  {F₀             = λ A → MkTwo A ⊥
  ; F₁            = λ f → MkHom f id
  ; identity      = ≐-refl , ≐-refl
  ; homomorphism  = ≐-refl , ≐-refl
  ; F-resp-≡ = λ f≈g → (λ x → f≈g {x}) , ≐-refl
  }
```

```
Cofree : (ℓ : Level) → Functor (Sets ℓ) (Twos ℓ)
Cofree ℓ = record
  {F₀            = λ A → MkTwo A ⊤
  ; F₁           = λ f → MkHom f id
  ; identity     = ≐-refl , ≐-refl
  ; homomorphism = ≐-refl , ≐-refl
  ; F-resp-≡  =  λ f≈g → (λ x → f≈g {x}) , ≐-refl
  }
  -- Dually, ( also shorter due to eta reduction )
Free² : (ℓ : Level) → Functor (Sets ℓ) (Twos ℓ)
Free² ℓ = record
  {F₀            = MkTwo ⊥
  ; F₁           = MkHom id
  ; identity     = ≐-refl , ≐-refl
  ; homomorphism = ≐-refl , ≐-refl
  ; F-resp-≡  =  λ f≈g → ≐-refl , λ x → f≈g {x}
  }
Cofree² : (ℓ : Level) → Functor (Sets ℓ) (Twos ℓ)
Cofree² ℓ = record
  {F₀            = MkTwo ⊤
  ; F₁           = MkHom id
  ; identity     = ≐-refl , ≐-refl
  ; homomorphism = ≐-refl , ≐-refl
  ; F-resp-≡  =  λ f≈g → ≐-refl , λ x → f≈g {x}
  }
```

6.4 Adjunction Proofs

Now for the actual proofs that the Free and Cofree functors are deserving of their names.

```
Left : (ℓ : Level) → Adjunction (Free ℓ) (Forget ℓ)
Left ℓ = record
  {unit = record
    {η = λ _ → id
    ; commute = λ _ → ≡.refl
    }
  ; counit = record
    {η = λ _ → MkHom id (λ {()})
    ; commute = λ f → ≐-refl , (λ {()})
    }
  ; zig = ≐-refl , (λ {()})
  ; zag = ≡.refl
  }
Right : (ℓ : Level) → Adjunction (Forget ℓ) (Cofree ℓ)
Right ℓ = record
  {unit = record
    {η = λ _ → MkHom id (λ _ → tt)
    ; commute = λ _ → ≐-refl , ≐-refl
    }
  ; counit = record {η = λ _ → id; commute = λ _ → ≡.refl}
  ; zig    = ≡.refl
  ; zag    = ≐-refl , λ {tt → ≡.refl}
  }
  -- Dually,
```

```
Left² : (ℓ : Level) → Adjunction (Free² ℓ) (Forget² ℓ)
Left² ℓ = record
  {unit = record
    {η = λ _ → id
    ; commute = λ _ → ≡.refl
    }
  ; counit = record
    {η = λ _ → MkHom (λ {()}) id
    ; commute = λ f → (λ {()}) , ≐-refl
    }
  ; zig = (λ {()}) , ≐-refl
  ; zag = ≡.refl
  }
Right² : (ℓ : Level) → Adjunction (Forget² ℓ) (Cofree² ℓ)
Right² ℓ = record
  {unit = record
    {η = λ _ → MkHom (λ _ → tt) id
    ; commute = λ _ → ≐-refl , ≐-refl
    }
  ; counit = record {η = λ _ → id; commute = λ _ → ≡.refl}
  ; zig    = ≡.refl
  ; zag    = (λ {tt → ≡.refl}) , ≐-refl
  }
```

6.5 Merging is adjoint to duplication

The category of sets contains products and so TwoSorted algebras can be represented there and, moreover, this is adjoint to duplicating a type to obtain a TwoSorted algebra.

```
  -- The category of Sets has products and so the TwoSorted type can be reified there.
Merge : (ℓ : Level) → Functor (Twos ℓ) (Sets ℓ)
Merge ℓ = record
  {F₀             = λ S → One S × Two S
  ; F₁            = λ F → one F ×₁ two F
  ; identity      = ≡.refl
  ; homomorphism  = ≡.refl
  ; F-resp-≡ = λ {(F≈₁G , F≈₂G) {x , y} → ≡.cong₂ _,_ (F≈₁G x) (F≈₂G y)}
  }
  -- Every set gives rise to its square as a TwoSorted type.
Dup : (ℓ : Level) → Functor (Sets ℓ) (Twos ℓ)
Dup ℓ = record
  {F₀             = λ A → MkTwo A A
  ; F₁            = λ f → MkHom f f
  ; identity      = ≐-refl , ≐-refl
  ; homomorphism  = ≐-refl , ≐-refl
  ; F-resp-≡ = λ F≈G → diag (λ _ → F≈G)
  }
```

Then the proof that these two form the desired adjunction

```
Right₂ : (ℓ : Level) → Adjunction (Dup ℓ) (Merge ℓ)
Right₂ ℓ = record
  {unit   = record {η = λ _ → diag; commute = λ _ → ≡.refl}
  ; counit = record {η = λ _ → MkHom proj₁ proj₂; commute = λ _ → ≐-refl , ≐-refl}
```

```
  ; zig      = ≐-refl , ≐-refl
  ; zag      = ≡.refl
  }
```

6.6 Duplication also has a left adjoint

The category of sets admits sums and so an alternative is to represet a TwoSorted algebra as a sum, and moreover this is adjoint to the aforementioned duplication functor.

```
Choice : (ℓ : Level) → Functor (Twos ℓ) (Sets ℓ)
Choice ℓ = record
  {F₀                = λ S → One S ⊎ Two S
  ; F₁               = λ F → one F ⊎₁ two F
  ; identity         = ⊎-id $ᵢ
  ; homomorphism  = λ {{x = x} → ⊎-∘ x}
  ; F-resp-≡  = λ F≈G {x} → uncurry ⊎-cong F≈G x
  }
Left₂ : (ℓ : Level) → Adjunction (Choice ℓ) (Dup ℓ)
Left₂ ℓ = record
  {unit      = record {η = λ _ → MkHom inj₁ inj₂; commute = λ _ → ≐-refl , ≐-refl}
  ; counit   = record {η = λ _ → from⊎; commute = λ _ {x} → (≡.sym ∘ from⊎-nat) x}
  ; zig      = λ {{_} {x} → from⊎-preInverse x}
  ; zag      = ≐-refl , ≐-refl
  }
```

7 Binary Heterogeneous Relations — **[MA:** *What* named *data structure do these correspond to in programming?* **]**

We consider two sorted algebras endowed with a binary heterogeneous relation. An example of such a structure is a graph, or network, which has a sort for edges and a sort for nodes and an incidence relation.

```
module Structures.Rel where
open import Level renaming (suc to lsuc; zero to lzero; _⊔_ to _⊍_)
open import Categories.Category    using (Category)
open import Categories.Functor     using (Functor)
open import Categories.Adjunction using (Adjunction)
open import Categories.Agda        using (Sets)
open import Function               using (id; _∘_; const)
open import Function2              using (_$ᵢ)
open import Forget
open import EqualityCombinators
open import DataProperties
open import Structures.TwoSorted using (TwoSorted; Twos; MkTwo) renaming (Hom to TwoHom; MkHom to MkTwoHom)
```

7.1 Definitions

We define the structure involved, along with a notational convenience:

```
record HetroRel ℓ ℓ′ : Set (lsuc (ℓ ⊍ ℓ′)) where
  constructor MkHRel
```

```
  field
    One : Set ℓ
    Two : Set ℓ
    Rel : One → Two → Set ℓ′
open HetroRel
relOp = HetroRel.Rel
syntax relOp A x y = x ⟨ A ⟩ y
```

Then define the strcture-preserving operations,

```
record Hom {ℓ ℓ′} (Src Tgt : HetroRel ℓ ℓ′) : Set (ℓ ⊍ ℓ′) where
  constructor MkHom
  field
    one : One Src → One Tgt
    two : Two Src → Two Tgt
    shift : {x : One Src} {y : Two Src} → x ⟨ Src ⟩ y → one x ⟨ Tgt ⟩ two y
open Hom
```

7.2 Category and Forgetful Functors

That these structures form a two-sorted algebraic category can easily be witnessed.

```
Rels : (ℓ ℓ′ : Level) → Category (lsuc (ℓ ⊍ ℓ′)) (ℓ ⊍ ℓ′) ℓ
Rels ℓ ℓ′ = record
  {Obj        = HetroRel ℓ ℓ′
  ; _⇒_       = Hom
  ; _≡_       = λ F G → one F ≐ one G × two F ≐ two G
  ; id        = MkHom id id id
  ; _∘_       = λ F G → MkHom (one F ∘ one G) (two F ∘ two G) (shift F ∘ shift G)
  ; assoc     = ≐-refl , ≐-refl
  ; identityˡ = ≐-refl , ≐-refl
  ; identityʳ = ≐-refl , ≐-refl
  ; equiv     = record
    {refl  = ≐-refl , ≐-refl
    ; sym   = λ {(oneEq , twoEq) → ≐-sym oneEq , ≐-sym twoEq}
    ; trans = λ {(oneEq₁ , twoEq₁) (oneEq₂ , twoEq₂) → ≐-trans oneEq₁ oneEq₂ , ≐-trans twoEq₁ twoEq₂}
    }
  ; ∘-resp-≡ = λ {(g≈₁k , g≈₂k) (f≈₁h , f≈₂h) → ∘-resp-≐ g≈₁k f≈₁h , ∘-resp-≐ g≈₂k f≈₂h}
  }
```

We can forget about the first sort or the second to arrive at our starting category and so we have two forgetful functors. Moreover, we can simply forget about the relation to arrive at the two-sorted category :-)

```
Forget¹ : (ℓ ℓ′ : Level) → Functor (Rels ℓ ℓ′) (Sets ℓ)
Forget¹ ℓ ℓ′ = record
  {F₀             = HetroRel.One
  ; F₁            = Hom.one
  ; identity      = ≡.refl
  ; homomorphism  = ≡.refl
  ; F-resp-≡ = λ {(F≈₁G , F≈₂G) {x} → F≈₁G x}
  }
Forget² : (ℓ ℓ′ : Level) → Functor (Rels ℓ ℓ′) (Sets ℓ)
Forget² ℓ ℓ′ = record
  {F₀             = HetroRel.Two
```

```
  ; F₁              = Hom.two
  ; identity        = ≡.refl
  ; homomorphism    = ≡.refl
  ; F-resp-≡  =  λ {(F≈₁G , F≈₂G) {x} → F≈₂G x}
  }
  -- Whence, Rels is a subcategory of Twos
Forget³ : (ℓ ℓ′ : Level) → Functor (Rels ℓ ℓ′) (Twos ℓ)
Forget³ ℓ ℓ′ = record
  {F₀               = λ S → MkTwo (One S) (Two S)
  ; F₁              = λ F → MkTwoHom (one F) (two F)
  ; identity        = ≐-refl , ≐-refl
  ; homomorphism    = ≐-refl , ≐-refl
  ; F-resp-≡  =  id
  }
```

7.3 Free and CoFree Functors

Given a (two)type, we can pair it with the empty type or the singleton type and so we have a free and a co-free constructions. Intuitively, the empty type denotes the empty relation which is the smallest relation and so a free construction; whereas, the singleton type denotes the "always true" relation which is the largest binary relation and so a cofree construction.

Candidate adjoints to forgetting the *first* component of a Rels

```
Free¹ : (ℓ ℓ′ : Level) → Functor (Sets ℓ) (Rels ℓ ℓ′)
Free¹ ℓ ℓ′ = record
  {F₀               = λ A → MkHRel A ⊥ (λ {_ ()})
  ; F₁              = λ f → MkHom f id (λ {{y = ()}})
  ; identity        = ≐-refl , ≐-refl
  ; homomorphism    = ≐-refl , ≐-refl
  ; F-resp-≡  =  λ f≈g → (λ x → f≈g {x}) , ≐-refl
  }
  -- (MkRel X ⊥ ⊥ ⟶ Alg) ≅ (X ⟶ One Alg)
Left¹ : (ℓ ℓ′ : Level) → Adjunction (Free¹ ℓ ℓ′) (Forget¹ ℓ ℓ′)
Left¹ ℓ ℓ′ = record
  {unit   = record
    {η = λ _ → id
    ; commute = λ _ → ≡.refl
    }
  ; counit = record
    {η = λ A → MkHom (λ z → z) (λ {()}) (λ {x} {})
    ; commute = λ f → ≐-refl , (λ ())
    }
  ; zig = ≐-refl , (λ ())
  ; zag = ≡.refl
  }
CoFree¹ : (ℓ : Level) → Functor (Sets ℓ) (Rels ℓ ℓ)
CoFree¹ ℓ = record
  {F₀               = λ A → MkHRel A ⊤ (λ _ _ → A)
  ; F₁              = λ f → MkHom f id f
  ; identity        = ≐-refl , ≐-refl
  ; homomorphism    = ≐-refl , ≐-refl
```

```
  ; F-resp-≡  =  λ f≈g → (λ x → f≈g {x}) , ≐-refl
  }
  -- (One Alg ⟶ X) ≅ (Alg ⟶ MkRel X ⊤ (λ _ _ → X)
Right¹ : (ℓ : Level) → Adjunction (Forget¹ ℓ ℓ) (CoFree¹ ℓ)
Right¹ ℓ  =  record
  {unit  =  record
    {η  =  λ _ → MkHom id (λ _ → tt) (λ {x} {y} _ → x)
    ; commute  =  λ _ → ≐-refl , (λ x → ≡.refl)
    }
  ; counit  =  record {η  =  λ _ → id; commute  =  λ _ → ≡.refl}
  ; zig     =  ≡.refl
  ; zag     =  ≐-refl , λ {tt → ≡.refl}
  }
  -- Another cofree functor:
CoFree¹′ : (ℓ : Level) → Functor (Sets ℓ) (Rels ℓ ℓ)
CoFree¹′ ℓ  =  record
  {F₀              =  λ A → MkHRel A ⊤ (λ _ _ → ⊤)
  ; F₁             =  λ f → MkHom f id id
  ; identity       =  ≐-refl , ≐-refl
  ; homomorphism   =  ≐-refl , ≐-refl
  ; F-resp-≡  =  λ f≈g → (λ x → f≈g {x}) , ≐-refl
  }
  -- (One Alg ⟶ X) ≅ (Alg ⟶ MkRel X ⊤ (λ _ _ → ⊤)
Right¹′ : (ℓ : Level) → Adjunction (Forget¹ ℓ ℓ) (CoFree¹′ ℓ)
Right¹′ ℓ  =  record
  {unit  =  record
    {η  =  λ _ → MkHom id (λ _ → tt) (λ {x} {y} _ → tt)
    ; commute  =  λ _ → ≐-refl , (λ x → ≡.refl)
    }
  ; counit  =  record {η  =  λ _ → id; commute  =  λ _ → ≡.refl}
  ; zig     =  ≡.refl
  ; zag     =  ≐-refl , λ {tt → ≡.refl}
  }
```

But wait, adjoints are necessarily unique, up to isomorphism, whence $\mathsf{CoFree}^1 \cong \mathsf{Cofree}^{1\prime}$. Intuitively, the relation part is a "subset" of the given carriers and when one of the carriers is a singleton then the largest relation is the universal relation which can be seen as either the first non-singleton carrier or the "always-true" relation which happens to be formalized by ignoring its arguments and going to a singleton set.

Candidate adjoints to forgetting the *second* component of a Rels

```
Free² : (ℓ : Level) → Functor (Sets ℓ) (Rels ℓ ℓ)
Free² ℓ  =  record
  {F₀              =  λ A → MkHRel ⊥ A (λ ())
  ; F₁             =  λ f → MkHom id f (λ {})
  ; identity       =  ≐-refl , ≐-refl
  ; homomorphism   =  ≐-refl , ≐-refl
  ; F-resp-≡  =  λ F≈G → ≐-refl , (λ x → F≈G {x})
  }
  -- (MkRel ⊥ X ⊥ ⟶ Alg) ≅ (X ⟶ Two Alg)
Left² : (ℓ : Level) → Adjunction (Free² ℓ) (Forget² ℓ ℓ)
Left² ℓ  =  record
  {unit  =  record
```

```
      {η = λ _ → id
      ; commute = λ _ → ≡.refl
      }
   ; counit = record
      {η = λ _ → MkHom (λ ()) id (λ {})
      ; commute = λ f → (λ ()) , ≐-refl
      }
   ; zig = (λ ()) , ≐-refl
   ; zag = ≡.refl
   }
CoFree² : (ℓ : Level) → Functor (Sets ℓ) (Rels ℓ ℓ)
CoFree² ℓ = record
   {F₀             =   λ A → MkHRel ⊤ A (λ _ _ → ⊤)
   ; F₁            =   λ f → MkHom id f id
   ; identity      =   ≐-refl , ≐-refl
   ; homomorphism  =   ≐-refl , ≐-refl
   ; F-resp-≡ = λ F≈G → ≐-refl , (λ x → F≈G {x})
   }
   -- (Two Alg ⟶ X) ≅ (Alg ⟶ ⊤ X ⊤
Right² : (ℓ : Level) → Adjunction (Forget² ℓ ℓ) (CoFree² ℓ)
Right² ℓ = record
   {unit = record
      {η = λ _ → MkHom (λ _ → tt) id (λ _ → tt)
      ; commute = λ f → ≐-refl , ≐-refl
      }
   ; counit = record
      {η = λ _ → id
      ; commute = λ _ → ≡.refl
      }
   ; zig = ≡.refl
   ; zag = (λ {tt → ≡.refl}) , ≐-refl
   }
```

Candidate adjoints to forgetting the *third* component of a Rels

```
Free³ : (ℓ : Level) → Functor (Twos ℓ) (Rels ℓ ℓ)
Free³ ℓ = record
   {F₀             =   λ S → MkHRel (One S) (Two S) (λ _ _ → ⊥)
   ; F₁            =   λ f → MkHom (one f) (two f) id
   ; identity      =   ≐-refl , ≐-refl
   ; homomorphism  =   ≐-refl , ≐-refl
   ; F-resp-≡ = id
   } where open TwoSorted; open TwoHom
   -- (MkTwo X Y → Alg without Rel) ≅ (MkRel X Y ⊥ ⟶ Alg)
Left³ : (ℓ : Level) → Adjunction (Free³ ℓ) (Forget³ ℓ ℓ)
Left³ ℓ = record
   {unit = record
      {η = λ A → MkTwoHom id id
      ; commute = λ F → ≐-refl , ≐-refl
      }
   ; counit = record
      {η = λ A → MkHom id id (λ ())
      ; commute = λ F → ≐-refl , ≐-refl
      }
```

```
  ; zig  =  ≐-refl , ≐-refl
  ; zag  =  ≐-refl , ≐-refl
  }
```

```
CoFree³ : (ℓ : Level) → Functor (Twos ℓ) (Rels ℓ ℓ)
CoFree³ ℓ = record
  {F₀             =   λ S → MkHRel (One S) (Two S) (λ _ _ → ⊤)
  ; F₁            =   λ f → MkHom (one f) (two f) id
  ; identity      =   ≐-refl , ≐-refl
  ; homomorphism  =   ≐-refl , ≐-refl
  ; F-resp-≡ = id
  } where open TwoSorted; open TwoHom

  -- (Alg without Rel ⟶ MkTwo X Y) ≅ (Alg ⟶ MkRel X Y ⊤)
Right³ : (ℓ : Level) → Adjunction (Forget³ ℓ ℓ) (CoFree³ ℓ)
Right³ ℓ = record
  {unit = record
    {η = λ A → MkHom id id (λ _ → tt)
    ; commute = λ F → ≐-refl , ≐-refl
    }
  ; counit = record
    {η = λ A → MkTwoHom id id
    ; commute = λ F → ≐-refl , ≐-refl
    }
  ; zig = ≐-refl , ≐-refl
  ; zag = ≐-refl , ≐-refl
  }
CoFree³′ : (ℓ : Level) → Functor (Twos ℓ) (Rels ℓ ℓ)
CoFree³′ ℓ = record
  {F₀             =   λ S → MkHRel (One S) (Two S) (λ _ _ → One S × Two S)
  ; F₁            =   λ F → MkHom (one F) (two F) (one F ×₁ two F)
  ; identity      =   ≐-refl , ≐-refl
  ; homomorphism  =   ≐-refl , ≐-refl
  ; F-resp-≡ = id
  } where open TwoSorted; open TwoHom

  -- (Alg without Rel ⟶ MkTwo X Y) ≅ (Alg ⟶ MkRel X Y X×Y)
Right³′ : (ℓ : Level) → Adjunction (Forget³ ℓ ℓ) (CoFree³′ ℓ)
Right³′ ℓ = record
  {unit = record
    {η = λ A → MkHom id id (λ {x} {y} x~y → x , y)
    ; commute = λ F → ≐-refl , ≐-refl
    }
  ; counit = record
    {η = λ A → MkTwoHom id id
    ; commute = λ F → ≐-refl , ≐-refl
    }
  ; zig = ≐-refl , ≐-refl
  ; zag = ≐-refl , ≐-refl
  }
```

But wait, adjoints are necessarily unique, up to isomorphism, whence $\mathsf{CoFree}^3 \cong \mathsf{CoFree}^{3\prime}$. Intuitively, the relation part is a "subset" of the given carriers and so the largest relation is the universal relation which can be seen as the product of the carriers or the "always-true" relation which happens to be formalized by ignoring its arguments and going to a singleton set.

7.4 ???

It remains to port over results such as Merge, Dup, and Choice from Twos to Rels.

Also to consider: sets with an equivalence relation; whence propositional equality.

The category of sets contains products and so TwoSorted algebras can be represented there and, moreover, this is adjoint to duplicating a type to obtain a TwoSorted algebra.

```
  -- The category of Sets has products and so the TwoSorted type can be reified there.
Merge : (ℓ : Level) → Functor (Twos ℓ) (Sets ℓ)
Merge ℓ = record
  {F₀            = λ S → One S × Two S
  ; F₁           = λ F → one F ×₁ two F
  ; identity     = ≡.refl
  ; homomorphism = ≡.refl
  ; F-resp-≡ = λ {(F≈₁G , F≈₂G) {x , y} → ≡.cong₂ _,_ (F≈₁G x) (F≈₂G y)}
  }
  -- Every set gives rise to its square as a TwoSorted type.
Dup : (ℓ : Level) → Functor (Sets ℓ) (Twos ℓ)
Dup ℓ = record
  {F₀            = λ A → MkTwo A A
  ; F₁           = λ f → MkHom f f
  ; identity     = ≐-refl , ≐-refl
  ; homomorphism = ≐-refl , ≐-refl
  ; F-resp-≡ = λ F≈G → diag (λ _ → F≈G)
  }
```

Then the proof that these two form the desired adjunction

```
Right₂ : (ℓ : Level) → Adjunction (Dup ℓ) (Merge ℓ)
Right₂ ℓ = record
  {unit   = record {η = λ _ → diag; commute = λ _ → ≡.refl}
  ; counit = record {η = λ _ → MkHom proj₁ proj₂; commute = λ _ → ≐-refl , ≐-refl}
  ; zig    = ≐-refl , ≐-refl
  ; zag    = ≡.refl
  }
```

The category of sets admits sums and so an alternative is to represet a TwoSorted algebra as a sum, and moreover this is adjoint to the aforementioned duplication functor.

```
Choice : (ℓ : Level) → Functor (Twos ℓ) (Sets ℓ)
Choice ℓ = record
  {F₀            = λ S → One S ⊎ Two S
  ; F₁           = λ F → one F ⊎₁ two F
  ; identity     = ⊎-id $ᵢ
  ; homomorphism = λ {{x = x} → ⊎-∘ x}
  ; F-resp-≡ = λ F≈G {x} → uncurry ⊎-cong F≈G x
  }
Left₂ : (ℓ : Level) → Adjunction (Choice ℓ) (Dup ℓ)
Left₂ ℓ = record
  {unit    = record {η = λ _ → MkHom inj₁ inj₂; commute = λ _ → ≐-refl , ≐-refl}
  ; counit = record {η = λ _ → from⊎; commute = λ _ {x} → (≡.sym ∘ from⊎-nat) x}
  ; zig    = λ {{_} {x} → from⊎-preInverse x}
  ; zag    = ≐-refl , ≐-refl
  }
```

8 Pointed Algebras: Nullable Types

We consider the theory of *pointed algebras* which consist of a type along with an elected value of that type.[1] Software engineers encounter such scenarios all the time in the case of an object-type and a default value of a "null", or undefined, object. In the more explicit setting of pure functional programming, this concept arises in the form of Maybe, or Option types.

Some programming languages, such as C# for example, provide a **default** keyword to access a default value of a given data type.

[**MA:** **insert:** Haskell's typeclass analogue of default?]

[**MA:** *Perhaps discuss "types as values" and the subtle issue of how pointed algebras are completely different than classes in an imperative setting.*]

```
module Structures.Pointed where

open import Level renaming (suc to lsuc; zero to lzero)
open import Categories.Category using (Category; module Category)
open import Categories.Functor using (Functor)
open import Categories.Adjunction using (Adjunction)
open import Categories.NaturalTransformation using (NaturalTransformation)
open import Categories.Agda using (Sets)
open import Function using (id; _∘_)
open import Data.Maybe using (Maybe; just; nothing; maybe; maybe′)

open import Forget

open import Data.Empty
open import Relation.Nullary

open import EqualityCombinators
```

8.1 Definition

As mentioned before, a Pointed algebra is a type, which we will refer to by Carrier, along with a value, or point, of that type.

```
record Pointed {a} : Set (lsuc a) where
  constructor MkPointed
  field
    Carrier : Set a
    point   : Carrier

open Pointed
```

Unsurprisingly, a "structure preserving operation" on such structures is a function between the underlying carriers that takes the source's point to the target's point.

```
record Hom {ℓ} (X Y : Pointed {ℓ}) : Set ℓ where
  constructor MkHom
  field
    mor          : Carrier X → Carrier Y
    preservation : mor (point X) ≡ point Y

open Hom
```

[1] Note that this definition is phrased as a "dependent product"!

8.2 Category and Forgetful Functors

Since there is only one type, or sort, involved in the definition, we may hazard these structures as "one sorted algebras":

```
oneSortedAlg : ∀ {ℓ} → OneSortedAlg ℓ
oneSortedAlg = record
  {Alg        = Pointed
  ; Carrier   = Carrier
  ; Hom       = Hom
  ; mor       = mor
  ; comp      = λ F G → MkHom (mor F ∘ mor G) (≡.cong (mor F) (preservation G) ⟨≡≡⟩ preservation F)
  ; comp-is-∘ = ≐-refl
  ; Id        = MkHom id ≡.refl
  ; Id-is-id  = ≐-refl
  }
```

From which we immediately obtain a category and a forgetful functor.

```
Pointeds : (ℓ : Level) → Category (lsuc ℓ) ℓ ℓ
Pointeds ℓ = oneSortedCategory ℓ oneSortedAlg

Forget : (ℓ : Level) → Functor (Pointeds ℓ) (Sets ℓ)
Forget ℓ = mkForgetful ℓ oneSortedAlg
```

The naming Pointeds is to be consistent with the category theory library we are using, which names the category of sets and functions by Sets. That is, the category name is the objects' name suffixed with an 's'.

Of-course, as hinted in the introduction, this structure —as are many— is defined in a dependent fashion and so we have another forgetful functor:

```
open import Data.Product
Forgetᴰ : (ℓ : Level) → Functor (Pointeds ℓ) (Sets ℓ)
Forgetᴰ ℓ = record {F₀ = λ P → Σ (Carrier P) (λ x → x ≡ point P)
  ; F₁ = λ {P} {Q} F → λ {(val , val≡ptP) → mor F val , (≡.cong (mor F) val≡ptP ⟨≡≡⟩ preservation F)}
  ; identity = λ {P} → λ {{val , val≡ptP} → ≡.cong (λ x → val , x) (≡.proof-irrelevance _ _)}
  ; homomorphism = λ {P} {Q} {R} {F} {G} → λ {{val , val≡ptP} → ≡.cong (λ x → mor G (mor F val) , x) (≡.proof-irrelevance _ _)}
  ; F-resp-≡ = λ {P} {Q} {F} {G} F≈G → λ {{val , val≡ptP} → {!≡.cong₂ _,_ (F≈G val) ?!}}
  }
```

That is, we "only remember the point".

[**MA: insert:** An adjoint to this functor?]

8.3 A Free Construction

As discussed earlier, the prime example of pointed algebras are the optional types, and this claim can be realised as a functor:

```
Free : (ℓ : Level) → Functor (Sets ℓ) (Pointeds ℓ)
Free ℓ = record
  {F₀            = λ A → MkPointed (Maybe A) nothing
  ; F₁           = λ f → MkHom (maybe (just ∘ f) nothing) ≡.refl
  ; identity     = maybe ≐-refl ≡.refl
  ; homomorphism = maybe ≐-refl ≡.refl
  ; F-resp-≡ = λ F≡G → maybe (∘-resp-≐ (≐-refl {x = just}) (λ x → F≡G {x})) ≡.refl
  }
```

Which is indeed deserving of its name:

```
MaybeLeft : (ℓ : Level) → Adjunction (Free ℓ) (Forget ℓ)
MaybeLeft ℓ = record
  {unit        = record {η = λ _ → just; commute = λ _ → ≡.refl}
  ; counit     = record
     {η        = λ X → MkHom (maybe id (point X)) ≡.refl
     ; commute = maybe ≐-refl ∘ ≡.sym ∘ preservation
     }
  ; zig        = maybe ≐-refl ≡.refl
  ; zag        = ≡.refl
  }
```

[MA: *Develop Maybe explicitly so we can "see" how the utility maybe "pops up naturally".* **]**

While there is a "least" pointed object for any given set, there is, in-general, no "largest" pointed object corresponding to any given set. That is, there is no co-free functor.

```
NoRight : {ℓ : Level} → (CoFree : Functor (Sets ℓ) (Pointeds ℓ)) → ¬ (Adjunction (Forget ℓ) CoFree)
NoRight (record {F₀ = f}) Adjunct = lower (η (counit Adjunct) (Lift ⊥) (point (f (Lift ⊥))))
  where open Adjunction
    open NaturalTransformation
```

9 Dependent Sums

We consider "dependent algebras" which consist of an index set and a family of sets on it. Alternatively, in can be construed as a universe of discourse along with an elected subset of interest. In the latter view the free and cofree constructions products the empty and universal predicates. In the former view, the we have an adjunction involving dependent products.

```
module Structures.Dependent where
open import Level renaming (suc to lsuc; zero to lzero; _⊔_ to _⊍_)
open import Categories.Category using (Category)
open import Categories.Functor using (Functor)
open import Categories.Adjunction using (Adjunction)
open import Function using (id; _∘_; const)
open import Function2 using (_$ᵢ)
open import Forget
open import EqualityCombinators hiding (_≡_; module ≡)
open import DataProperties
import Relation.Binary.HeterogeneousEquality
module ≅ = Relation.Binary.HeterogeneousEquality
open ≅ using (_≅_)
  -- category of sets with heterogenous equality
Sets : (ℓ : Level) → Category (lsuc ℓ) ℓ ℓ
Sets ℓ = record
  {Obj     = Set ℓ
  ; _⇒_    = λ A B → A → B
  ; _≡_    = λ {A} {B} f g → {x : A} → f x ≅ g x
  ; _∘_    = λ f g x → f (g x)
  ; id     = λ x → x
  ; assoc  = ≅.refl
```

```
  ; identityˡ  = ≅.refl
  ; identityʳ  = ≅.refl
  ; equiv      = record {refl = ≅.refl; sym = λ eq → ≅.sym eq; trans = λ p q → ≅.trans p q}
  ; ∘-resp-≡  = λ {{f = f} f≅h g≅k → ≅.trans (≅.cong f g≅k) f≅h}
  }
```

9.1 Definition

A Dependent algebra consists of a carrier acting as an index for another family of functions. An array is an example of this with the index set being the valid indices. Alternatively, the named fields of a class-object are the indices for that class-object.

```
record Dependent a b : Set (lsuc (a ⊍ b)) where
  constructor MkDep
  field
    Sort : Set a
    Carrier : Sort → Set b
open Dependent
```

Alternatively, these can be construed as some universe Index furnished with a constructive predicated Field :-) That is to say, these may also be known as "unary relational algebras".

Moreover, we can construe Index as sort symbols, that is "uninterpreted types" of some universe, and Field is then the interpretation of those symbols as a reification in the ambient type system.

Again, we may name these "many sorted".

```
record Hom {a b} (Src Tgt : Dependent a b) : Set (a ⊍ b) where
  constructor MkHom
  field
    mor : Sort Src → Sort Tgt
    shift : {i : Sort Src} → Carrier Src i → Carrier Tgt (mor i)
open Hom
```

The shift condition may be read, in the predicate case, as: if i is in the predicate in the source, then its images is in the predicate of the target.

Such categories have been studies before under the guide "the category of sets with an elected subset".

9.2 Category and Forgetful Functor

```
Dependents : (ℓ : Level) → Category (lsuc ℓ) ℓ ℓ
Dependents ℓ = record
  {Obj       = Dependent ℓ ℓ
  ; _⇒_     =   Hom
  ; _≡_      =   λ {A} {B} F G → ({x : Sort A} → mor F x ≅ mor G x)
                   × ({s : Sort A} {f : Carrier A s} → shift F f ≅ shift G f)
  ; id       =   MkHom id id
  ; _∘_      =   λ F G → MkHom (mor F ∘ mor G) (shift F ∘ shift G)
  ; assoc    =   ≅.refl , ≅.refl
  ; identityˡ  = ≅.refl , ≅.refl
  ; identityʳ  = ≅.refl , ≅.refl
  ; equiv      = record
    {refl = ≅.refl , ≅.refl
```

```
    ; sym  =  λ {(eq , eq') → ≅.sym eq , ≅.sym eq'}
    ; trans  =  λ {(peq , peq') (qeq , qeq') → ≅.trans peq qeq , ≅.trans peq' qeq'}
    }
  ; ∘-resp-≡  =  λ {{f  =  f} (f≅h , f≅h') (g≅k , g≅k') → ≅.trans (≅.cong (mor f) g≅k) f≅h ,
                    ≅.trans (≅.cong (shift {!f!}) g≅k') f≅h'}
  }
  where open import Relation.Binary
```

We can forget about the first sort or the second to arrive at our starting category and so we have two forgetful functors.

```
Forget : (ℓ : Level) → Functor (Dependents ℓ) (Sets ℓ)
Forget ℓ  =  record
  {F₀              =  Dependent.Sort
  ; F₁             =  Hom.mor
  ; identity       =  ≅.refl
  ; homomorphism   =  ≅.refl
  ; F-resp-≡  =  λ {(F≈G , _) → F≈G}
  }
```

[**MA:** *ToDo :: construct another forgetful functor*]

9.3 Free and CoFree

Given a type, we can pair it with the empty type or the singelton type and so we have a free and a co-free constructions.

```
Free : (ℓ : Level) → Functor (Sets ℓ) (Dependents ℓ)
Free ℓ  =  record
  {F₀              =  λ A → MkDep A (λ _ → ⊥)
  ; F₁             =  λ f → MkHom f id
  ; identity       =  ≅.refl , ≅.refl
  ; homomorphism   =  ≅.refl , ≅.refl
  ; F-resp-≡  =  λ F≈G → F≈G , ≅.refl
  }
Cofree : (ℓ : Level) → Functor (Sets ℓ) (Dependents ℓ)
Cofree ℓ  =  record
  {F₀              =  λ A → MkDep A (λ _ → ⊤)
  ; F₁             =  λ f → MkHom f id
  ; identity       =  ≅.refl , ≅.refl
  ; homomorphism   =  ≅.refl , ≅.refl
  ; F-resp-≡  =  λ f≈g → f≈g , ≅.refl
  }
```

9.4 Left and Right adjunctions

Now for the actual proofs that the Free and Cofree functors are deserving of their names.

```
Left : (ℓ : Level) → Adjunction (Free ℓ) (Forget ℓ)
Left ℓ  =  record
  {unit  =  record
    {η  =  λ _ → id
    ; commute  =  λ _ → ≅.refl
```

```
    }
  ; counit = record
    {η = λ _ → MkHom id (λ {()})
    ; commute = λ f → ≅.refl , (λ {{_} {()}})
    }
  ; zig = ≅.refl , (λ {{_} {()}})
  ; zag = ≅.refl
  }
Right : (ℓ : Level) → Adjunction (Forget ℓ) (Cofree ℓ)
Right ℓ = record
  {unit = record
    {η = λ _ → MkHom id (λ _ → tt)
    ; commute = λ _ → ≅.refl , ≅.refl
    }
  ; counit = record {η = λ _ → id; commute = λ _ → ≅.refl}
  ; zig    = ≅.refl
  ; zag    = ≅.refl , (λ {{_} {tt} → ≅.refl})
  }
```

9.5 DepProd

The category of sets contains products and so Dependent algebras can be represented there and, moreover, this is adjoint to duplicating a type to obtain a TwoSorted algebra.

```
≡-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
```

```
DepProd : (ℓ : Level) → Functor (Dependents ℓ) (Sets ℓ)
DepProd ℓ = record
  {F₀            = λ S → Σ (Sort S) (Carrier S)
  ; F₁           = λ F → mor F ×₁ shift F
  ; identity     = λ {{_} {fst , snd} → ≅.refl}
  ; homomorphism = ≅.refl
  ; F-resp-≡ = λ {(F≈G , eq) → ≅.cong₂ _,_ F≈G eq}   -- This was the troublesome hole; now filled!
  }
```

Begin inactive material

```
  where helper : {a b : Level} {S T : Dependent a b} {F G : Hom S T}
    → (F≈G : mor F ≐ mor G)
    → {i : Sort S} {f : Carrier S i}
    → shift F f ≡ ≡.subst (Carrier T) (≡.sym (F≈G i)) (shift G f)
    helper {S = S} {T} {F} {G} F≈G {i} {f} with ≡.cong (Carrier T) (F≈G i)
    ...| r = {!see RATH's propeq utils, maybe something there helps!}
  -- ! consider using Relation.Binary.HeterogenousEquality . . . !
  -- Every set gives rise to an identity family on itself
ID : (ℓ : Level) → Functor (Sets ℓ) (Dependents ℓ)
ID ℓ = record
  {F₀            = λ A → MkDep A (λ _ → A)
  ; F₁           = λ f → MkHom f f
  ; identity     = ≐-refl
  ; homomorphism = ≐-refl
```

```
  ; F-resp-≡  =  λ F≈G → λ x → F≈G {x}
  }
  -- look into having a free functor from TwoCat, then _×_ pops up!
  -- maybe not, what is the forgetful functor...!
f : {!\unfinished!}
f = {!\unfinished!}
```

Then the proof that these two form the desired adjunction

```
Right₂ : (ℓ : Level) → Adjunction (ID ℓ) (DepProd ℓ)
Right₂ ℓ = record
  {unit    = record {η = λ _ → diag; commute = λ _ → ≡.refl}
  ; counit = record {η = λ _ → MkHom proj₁ (λ {{i , f} _ → f}); commute = λ _ → ≐-refl}
  ; zig    = ≐-refl
  ; zag    = ≡.refl
  }
```

Note that since Σ encompasses both × and ⊎, it may ¬ be that there is another functor co-adjoint to ID —not sure though.

10 Distinguished Subset Algebras

```
module Structures.DistinguishedSubset where
open import Level renaming (suc to lsuc; zero to lzero)
open import Categories.Category using (Category)
open import Categories.Functor  using (Functor)
open import Categories.Adjunction using (Adjunction)
open import Categories.Agda     using (Sets)
open import Function            using (id; _∘_; const)
open import Function2           using (_$ᵢ)
open import Data.Bool           using (Bool; true; false)
open import Relation.Nullary    using (¬_)
open import Forget
open import EqualityCombinators
open import DataProperties
```

```
record Disting a : Set (lsuc a) where
  constructor MkDist
  field
    Index : Set a
    Field : Index → Bool
open Disting
```

Alternatively, these can be construed as some universe Index furnished with a constructive predicated Field :-) That is to say, these may also be known as “unary relational algebras”.

```
record Hom {a} (Src Tgt : Disting a) : Set a where
  constructor MkHom
  field
    mor : Index Src → Index Tgt
    shift : {i : Index Src} → Field Src i ≡ Field Tgt (mor i)
open Hom
```

The shift condition may be read, in the predicate case, as: if i is in the predicate in the source, then its images is in the predicate of the target.

Such categories have been studied before under the guide “the category of sets with a distinguished subset”.

```
DependentCat : (ℓ : Level) → Category (lsuc ℓ) ℓ ℓ
DependentCat ℓ = record
  {Obj       = Disting ℓ
  ; _⇒_   =   Hom
  ; _≡_    =   λ F G → (mor F ≐ mor G)
  ; id         = MkHom id ≡.refl
  ; _∘_       = λ F G → MkHom (mor F ∘ mor G) (shift G ⟨≡≡⟩ shift F)
  ; assoc     = λ _ → ≡.refl
  ; identityˡ  = λ _ → ≡.refl
  ; identityʳ  = λ _ → ≡.refl
  ; equiv      = λ {A} {B} → record
                    {refl  =  ≐-refl
                    ; sym  =  ≐-sym
                    ; trans  =  ≐-trans}   -- record  IsEquivalence ≐-isEquivalence
  ; ∘-resp-≡  = ∘-resp-≐
  }
  where open import Relation.Binary
```

We can forget about the first sort or the second to arrive at our starting category and so we have two forgetful functors.

```
Forget : (ℓ : Level) → Functor (DependentCat ℓ) (Sets ℓ)
Forget ℓ = record
  {F₀                 = Disting.Index
  ; F₁                 = Hom.mor
  ; identity           = ≡.refl
  ; homomorphism  = ≡.refl
  ; F-resp-≡  =  λ F≈G {x} → F≈G x
  }
```

ToDo :: construct another forgetful functor

Given a type, we can pair it with the empty type or the singelton type and so we have a free and a co-free constructions.

```
Free : (ℓ : Level) → Functor (Sets ℓ) (DependentCat ℓ)
Free ℓ = record
  {F₀                 = λ A → MkDist (A × (A → Bool)) (λ {(a , R) → R a})
  ; F₁                 = λ f → MkHom (λ {(a , R) → f a , (λ b → {!!})}) {!!}
  ; identity           = {!!}
  ; homomorphism  = {!!}
  ; F-resp-≡  =  λ F≈G → {!!}
  }
Cofree : (ℓ : Level) → Functor (Sets ℓ) (DependentCat ℓ)
Cofree ℓ = record
  {F₀                 = λ A → MkDist {!!} (λ a → {!!})
  ; F₁                 = λ f → MkHom {!!} {!!}
  ; identity           = {!!}   -- ≐-refl
  ; homomorphism  = {!!}   -- ≐-refl
  ; F-resp-≡  =  λ f≈g → {!!}    -- f≈g x
  }
```

Now for the actual proofs that the Free and Cofree functors are deserving of their names.

```
Left : (ℓ : Level) → Adjunction (Free ℓ) (Forget ℓ)
Left ℓ = record
  {unit = record
    {η = λ _ → {!!}
    ;commute = λ _ → ≡.refl
    }
  ;counit = record
    {η = λ {(MkDist A R) → MkHom (λ a → {!!}) {!!}}
    ;commute = λ f → {!!}   -- ≐-refl
    }
  ;zig = {!!}   -- ≐-refl
  ;zag = {!!}
  }
Right : (ℓ : Level) → Adjunction (Forget ℓ) (Cofree ℓ)
Right ℓ = record
  {unit = record
    {η = λ {(MkDist A R) → MkHom {!!} {!!}}
    ;commute = λ _ → {!!}   -- ≐-refl
    }
  ;counit = record {η = λ _ → {!!};commute = λ f → {!!}}
  ;zig    = ≡.refl
  ;zag    = {!!}   -- ≐-refl
  }
```

The category of sets contains products and so Dependent algebras can be represented there and, moreover, this is adjoint to duplicating a type to obtain a TwoSorted algebra.

```
DepProd : (ℓ : Level) → Functor (DependentCat ℓ) (Sets ℓ)
DepProd ℓ = record
  {F₀              = λ S → Σ (Index S) {!!}
  ;F₁              = λ F → mor F ×₁ {!!}
  ;identity        = ≡.refl
  ;homomorphism    = ≡.refl
  ;F-resp-≡ =
    λ {A} {B} F≈G → {!!}
  }
  -- Every set gives rise to an identity family on itself
ID : (ℓ : Level) → Functor (Sets ℓ) (DependentCat ℓ)
ID ℓ = record
  {F₀              = λ A → MkDist A (λ _ → {!!})
  ;F₁              = λ f → MkHom f {!!}
  ;identity        = {!!}   -- ≐-refl
  ;homomorphism    = {!!}   -- ≐-refl
  ;F-resp-≡ = λ F≈G → {!!}   -- λ x → F≈G x
  }
```

Then the proof that these two form the desired adjunction

```
Right₂ : (ℓ : Level) → Adjunction (ID ℓ) (DepProd ℓ)
Right₂ ℓ = record
  {unit   = record {η = λ _ → diag;commute = λ _ → ≡.refl}
  ;counit = record {η = λ _ → MkHom proj₁ {!!};commute = λ _ → {!!}}
  ;zig    = {!!}   -- ≐-refl
  ;zag    = ≡.refl
  }
```

Note that since Σ encompasses both $\times$ and $\uplus$, it may not be that there is another functor co-adjoint to ID —not sure though.

Part III

Unary Algebras

11 UnaryAlgebra

Unary algebras are tantamount to an OOP interface with a single operation. The associated free structure captures the "syntax" of such interfaces, say, for the sake of delayed evaluation in a particular interface implementation.

This example algebra serves to set-up the approach we take in more involved settings.

[**MA:** *This section requires massive reorganisation.*]

```
module Structures.UnaryAlgebra where

open import Level renaming (suc to lsuc; zero to lzero)

open import Categories.Category using (Category; module Category)
open import Categories.Functor   using (Functor; Contravariant)
open import Categories.Adjunction using (Adjunction)
open import Categories.Agda       using (Sets)
open import Forget

open import Data.Nat using (ℕ; suc; zero)
open import DataProperties

open import Function2
open import Function

open import EqualityCombinators
```

11.1 Definition

A single-sorted Unary algebra consists of a type along with a function on that type. For example, the naturals and addition-by-1 or lists and the reverse operation.

```
record Unary {ℓ} : Set (lsuc ℓ) where
  constructor MkUnary
  field
    Carrier : Set ℓ
    Op : Carrier → Carrier

open Unary

record Hom {ℓ} (X Y : Unary {ℓ}) : Set ℓ where
  constructor MkHom
  field
    mor    : Carrier X → Carrier Y
    pres-op : mor ∘ Op X ≐ᵢ Op Y ∘ mor

open Hom
```

11.2 Category and Forgetful Functor

Along with functions that preserve the elected operation, such algberas form a category.

```
UnaryAlg : {ℓ : Level} → OneSortedAlg ℓ
UnaryAlg = record
  {Alg      = Unary
  ; Carrier = Carrier
  ; Hom     = Hom
  ; mor     = mor
  ; comp    = λ F G → record
    {mor     =   mor F ∘ mor G
    ; pres-op = ≡.cong (mor F) (pres-op G) ⟨≡≡⟩ pres-op F
    }
  ; comp-is-∘ = ≐-refl
  ; Id        =   MkHom id ≡.refl
  ; Id-is-id  =   ≐-refl
  }
Unarys : (ℓ : Level) → Category (lsuc ℓ) ℓ ℓ
Unarys ℓ = oneSortedCategory ℓ UnaryAlg
Forget : (ℓ : Level) → Functor (Unarys ℓ) (Sets ℓ)
Forget ℓ = mkForgetful ℓ UnaryAlg
```

11.3 Free Structure

We now turn to finding a free unary algebra.

Indeed, we do so by simply not "interpreting" the single function symbol that is required as part of the definition. That is, we form the "term algebra" over the signature for unary algebras.

```
data Eventually {ℓ} (A : Set ℓ) : Set ℓ where
  base : A → Eventually A
  step : Eventually A → Eventually A
```

The elements of this type are of the form $\mathsf{step}^{\mathsf{n}}$ (base a) for a : A. This leads to an alternative presentation, Eventually A ≅ Σ n : ℕ • A viz $\mathsf{step}^{\mathsf{n}}$ (base a) ↔ (n , a) —cf Free^2 below. Incidentally, or promisingly, Eventually ⊤ ≅ ℕ.

We will realise this claim later on. For now, we turn to the dependent-eliminator/induction/recursion principle:

```
elim : {ℓ a : Level} {A : Set a} {P : Eventually A → Set ℓ}
  → ({x : A} → P (base x))
  → ({sofar : Eventually A} → P sofar → P (step sofar))
  → (ev : Eventually A) → P ev
elim b s (base x) = b {x}
elim {P = P} b s (step e) = s {e} (elim {P = P} b s e)
```

Given an unary algebra (B, $_B$, $_S$) we can interpret the terms of Eventually A where the injection base is reified by $_B$ and the unary operation step is reified by $_S$.

```
open import Function using (const)
⟦_,_⟧ : {a b : Level} {A : Set a} {B : Set b} (_B : A → B) (_S : B → B) → Eventually A → B
⟦ _B , _S ⟧ = elim (λ {a} → _B a) _S
```

Notice that: The number of $_S$teps is preserved, ⟦ $_B$, $_S$ ⟧ ∘ $\mathsf{step}^{\mathsf{n}}$ ≐ $_S{}^{\mathsf{n}}$ ∘ ⟦ $_B$, $_S$ ⟧. Essentially, ⟦ $_B$, $_S$ ⟧ ($\mathsf{step}^{\mathsf{n}}$ base x) ≈ $_S{}^{\mathsf{n}}$ $_B$ x. A similar general remark applies to elim.

Here is an implicit version of elim,

Eventually is clearly a functor,

```
map : {a b : Level} {A : Set a} {B : Set b} → (A → B) → (Eventually A → Eventually B)
map f = ⟦ base ∘ f , step ⟧
```

Whence the folding operation is natural,

```
⟦⟧-naturality : {a b : Level} {A : Set a} {B : Set b}
  → {B′ S′ : A → A} {B S : B → B} {f : A → B}
  → (basis : B ∘ f ≐i f ∘ B′)
  → (next : S ∘ f ≐i f ∘ S′)
  → ⟦ B , S ⟧ ∘ map f ≐ f ∘ ⟦ B′ , S′ ⟧
⟦⟧-naturality {S = S} basis next = elim basis (λ ind → ≡.cong S ind ⟨≡≡⟩ next)
```

Other instances of the fold include:

```
extract : ∀ {ℓ} {A : Set ℓ} → Eventually A → A
extract = ⟦ id , id ⟧  -- cf from⊎ ;)
```

[**MA:** *Mention comonads?*]

More generally,

```
iterate : ∀ {ℓ} {A : Set ℓ} (f : A → A) → Eventually A → A
iterate f = ⟦ id , f ⟧
  --
  -- that is, iterateE f (step^n base x) ≈ f^n x
iterate-nat : {ℓ : Level} {X Y : Unary {ℓ}} (F : Hom X Y)
  → iterate (Op Y) ∘ map (mor F) ≐ mor F ∘ iterate (Op X)
iterate-nat F = ⟦⟧-naturality {f = mor F} ≡.refl (≡.sym (pres-op F))
```

The induction rule yields identical looking proofs for clearly distinct results:

```
iterate-map-id : {ℓ : Level} {X : Set ℓ} → id {A = Eventually X} ≐ iterate step ∘ map base
iterate-map-id = elim ≡.refl (≡.cong step)
map-id : {a : Level} {A : Set a} → map (id {A = A}) ≐ id
map-id = elim ≡.refl (≡.cong step)
map-∘ : {ℓ : Level} {X Y Z : Set ℓ} {f : X → Y} {g : Y → Z}
  → map (g ∘ f) ≐ map g ∘ map f
map-∘ = elim ≡.refl (≡.cong step)
map-cong : ∀ {o} {A B : Set o} {F G : A → B} → F ≐ G → map F ≐ map G
map-cong eq = elim (≡.cong base ∘ eq $i) (≡.cong step)
```

These results could be generalised to ⟦_,_⟧ if needed.

11.4 The Toolki Appears Naturally: Part 1

That Eventually furnishes a set with its free unary algebra can now be realised.

```
Free : (ℓ : Level) → Functor (Sets ℓ) (Unarys ℓ)
Free ℓ = record
  {F0          = λ A → MkUnary (Eventually A) step
```

```
  ; F₁            = λ f → MkHom (map f) ≡.refl
  ; identity      = map-id
  ; homomorphism  = map-∘
  ; F-resp-≡  =  λ F≈G → map-cong (λ _ → F≈G)
  }
AdjLeft : (ℓ : Level) → Adjunction (Free ℓ) (Forget ℓ)
AdjLeft ℓ = record
  {unit    = record {η = λ _ → base; commute = λ _ → ≡.refl}
  ; counit = record {η = λ A → MkHom (iterate (Op A)) ≡.refl; commute = iterate-nat}
  ; zig    = iterate-map-id
  ; zag    = ≡.refl
  }
```

Notice that the adjunction proof forces us to come-up with the operations and properties about them!

- map: usually functions can be packaged-up to work on syntax of unary algebras.
- map-id: the identity function leaves syntax alone; or: map id can be replaced with a constant time algorithm, namely, id.
- map-∘: sequential substitutions on syntax can be efficiently replaced with a single substitution.
- map-cong: observably indistinguishable substitutions can be used in place of one another, similar to the transparency principle of Haskell programs.
- iterate: given a function f, we have $\mathsf{step}^n\ \mathsf{base}\ x \mapsto f^n\ x$. Along with properties of this operation.

```
_^_ : {a : Level} {A : Set a} (f : A → A) → ℕ → (A → A)
f ↑ zero = id
f ↑ suc n = f ↑ n ∘ f
  -- important property of iteration that allows it to be defined in an alternative fashion
iter-swap : {ℓ : Level} {A : Set ℓ} {f : A → A} {n : ℕ} → (f ↑ n) ∘ f ≐ f ∘ (f ↑ n)
iter-swap {n = zero} = ≐-refl
iter-swap {f = f} {n = suc n} = ∘-≐-cong₁ f iter-swap
  -- iteration of commutable functions
iter-comm : {ℓ : Level} {B C : Set ℓ} {f : B → C} {g : B → B} {h : C → C}
  → (leap-frog : f ∘ g ≐ᵢ h ∘ f)
  → {n : ℕ} → h ↑ n ∘ f ≐ᵢ f ∘ g ↑ n
iter-comm leap {zero} = ≡.refl
iter-comm {g = g} {h} leap {suc n} = ≡.cong (h ↑ n) (≡.sym leap) ⟨≡≡⟩ iter-comm leap
  -- exponentation distributes over product
^-over-× : {a b : Level} {A : Set a} {B : Set b} {f : A → A} {g : B → B}
  → {n : ℕ} → (f ×₁ g) ↑ n ≐ (f ↑ n) ×₁ (g ↑ n)
^-over-× {n = zero} = λ {(x , y) → ≡.refl}
^-over-× {f = f} {g} {n = suc n} = ^-over-× {n = n} ∘ (f ×₁ g)
```

11.5 The Toolki Appears Naturally: Part 2

And now for a different way of looking at the same algebra. We "mark" a piece of data with its depth.

```
Free² : (ℓ : Level) → Functor (Sets ℓ) (Unarys ℓ)
Free² ℓ = record
  {F₀             = λ A → MkUnary (ℕ × A) (suc ×₁ id)
  ; F₁            = λ f → MkHom (id ×₁ f) ≡.refl
  ; identity      = ≐-refl
  ; homomorphism  = ≐-refl
  ; F-resp-≡  =  λ F≈G → λ {(n , x) → ≡.cong₂ _,_ ≡.refl (F≈G {x})}
```

```
  }
  -- tagging operation
at : {a : Level} {A : Set a} → ℕ → A → ℕ × A
at n = λ x → (n , x)

ziggy : {a : Level} {A : Set a} (n : ℕ) → at n ≐ (suc ×₁ id {A = A}) ↑ n ∘ at 0
ziggy zero = ≐-refl
ziggy {A = A} (suc n) = begin⟨ ≐-setoid A (ℕ × A) ⟩
  (suc ×₁ id) ∘ at n                          ≈⟨ ∘-≐-cong₂ (suc ×₁ id) (ziggy n) ⟩
  (suc ×₁ id) ∘ (suc ×₁ id {A = A}) ↑ n ∘ at 0 ≈⟨ ∘-≐-cong₁ (at 0) (≐-sym iter-swap) ⟩
  (suc ×₁ id {A = A}) ↑ n ∘ (suc ×₁ id) ∘ at 0 ■
  where open import Relation.Binary.SetoidReasoning

AdjLeft² : ∀ o → Adjunction (Free² o) (Forget o)
AdjLeft² o = record
  {unit        = record {η = λ _ → at 0; commute = λ _ → ≡.refl}
  ; counit     = record
    {η         = λ A → MkHom (uncurry (Op A ^_)) (λ {{n , a} → iter-swap a})
    ; commute  = λ F → uncurry (λ x y → iter-comm (pres-op F))
    }
  ; zig        = uncurry ziggy
  ; zag        = ≡.refl
  }
```

Notice that the adjunction proof forces us to come-up with the operations and properties about them!

- iter-comm: $\boxed{???}$
- _^_: $\boxed{???}$
- iter-swap: $\boxed{???}$
- ziggy: $\boxed{???}$

12 Involutive Algebras: Sum and Product Types

Free and cofree constructions wrt these algebras "naturally" give rise to the notion of sum and product types.

```
module Structures.InvolutiveAlgebra where

open import Level renaming (suc to lsuc; zero to lzero)
open import Categories.Category  using (Category; module Category)
open import Categories.Functor   using (Functor; Contravariant)
open import Categories.Adjunction using (Adjunction)
open import Categories.Agda      using (Sets)
open import Categories.Monad     using (Monad)
open import Categories.Comonad   using (Comonad)

open import Function
open import Function2 using (_$ᵢ)

open import DataProperties
open import EqualityCombinators
```

12.1 Definition

```
record Inv {ℓ} : Set (lsuc ℓ) where
  field
    A : Set ℓ
```

```
    _° : A → A
    involutive : ∀ (a : A) → a ° ° ≡ a
open Inv renaming (A to Carrier; _° to inv)

record Hom {ℓ} (X Y : Inv {ℓ}) : Set ℓ where
  open Inv X; open Inv Y renaming (_° to _O)
  field
    mor : Carrier X → Carrier Y
    pres : (x : Carrier X) → mor (x °) ≡ (mor x) O
open Hom
```

12.2 Category and Forgetful Functor

[MA: *can regain via onesortedalgebra construction* **]**

```
Involutives : (ℓ : Level) → Category _ ℓ ℓ
Involutives ℓ = record
  {Obj       = Inv
  ; _⇒_      = Hom
  ; _≡_      = λ F G → mor F ≐ mor G
  ; id       = record {mor = id; pres = ≐-refl}
  ; _∘_      = λ F G → record
    {mor     = mor F ∘ mor G
    ; pres   = λ a → ≡.cong (mor F) (pres G a) ⟨≡≡⟩ pres F (mor G a)
    }
  ; assoc    = ≐-refl
  ; identityˡ = ≐-refl
  ; identityʳ = ≐-refl
  ; equiv    = record {IsEquivalence ≐-isEquivalence}
  ; ∘-resp-≡ = ∘-resp-≐
  }
  where open Hom; open import Relation.Binary using (IsEquivalence)

Forget : (o : Level) → Functor (Involutives o) (Sets o)
Forget _ = record
  {F₀              = Carrier
  ; F₁             = mor
  ; identity       = ≡.refl
  ; homomorphism   = ≡.refl
  ; F-resp-≡ = _$ᵢ
  }
```

12.3 Free Adjunction: Part 1 of a toolkit

The double of a type has an involution on it by swapping the tags:

```
swap₊ : {ℓ : Level} {X : Set ℓ} → X ⊎ X → X ⊎ X
swap₊ = [ inj₂ , inj₁ ]

swap² : {ℓ : Level} {X : Set ℓ} → swap₊ ∘ swap₊ ≐ id {A = X ⊎ X}
swap² = [ ≐-refl , ≐-refl ]
```

```
2 ×_ : {ℓ : Level} {X Y : Set ℓ}
  → (X → Y)
```

```
  → X ⊎ X → Y ⊎ Y
2 × f = f ⊎₁ f
2 ×-over-swap : {ℓ : Level} {X Y : Set ℓ} {f : X → Y}
  → 2 × f ∘ swap₊ ≐ swap₊ ∘ 2 × f
2 ×-over-swap = [ ≐-refl , ≐-refl ]
2 ×-id≈id : {ℓ : Level} {X : Set ℓ} → 2 × id ≐ id {A = X ⊎ X}
2 ×-id≈id = [ ≐-refl , ≐-refl ]
2 ×-∘ : {ℓ : Level} {X Y Z : Set ℓ} {f : X → Y} {g : Y → Z}
  → 2 × (g ∘ f) ≐ 2 × g ∘ 2 × f
2 ×-∘ = [ ≐-refl , ≐-refl ]
2 ×-cong : {ℓ : Level} {X Y : Set ℓ} {f g : X → Y}
  → f ≐ᵢ g
  → 2 × f ≐ 2 × g
2 ×-cong F≈G = [ (λ _ → ≡.cong inj₁ F≈G) , (λ _ → ≡.cong inj₂ F≈G) ]
Left : (ℓ : Level) → Functor (Sets ℓ) (Involutives ℓ)
Left ℓ = record
  {F₀             = λ A → record {A = A ⊎ A; _° = swap₊; involutive = swap²}
  ;F₁             = λ f → record {mor = 2 × f; pres = 2 ×-over-swap}
  ;identity       = 2 ×-id≈id
  ;homomorphism   = 2 ×-∘
  ;F-resp-≡       = 2 ×-cong
  }
```

Notice that the adjunction proof forces us to come-up with the operations and properties about them!

- 2 ×: usually functions can be packaged-up to work on syntax of unary algebras.
- 2 ×-id≈id: the identity function leaves syntax alone; or: map id can be replaced with a constant time algorithm, namely, id.
- 2 ×-∘: sequential substitutions on syntax can be efficiently replaced with a single substitution.
- 2 ×-cong: observably indistinguishable substitutions can be used in place of one another, similar to the transparency principle of Haskell programs.
- 2 ×-over-swap: $\boxed{???}$
- swap_+: $\boxed{???}$
- swap^2: $\boxed{???}$
- $\boxed{???}$

There are actually two left adjoints. It seems the choice of inj_1 / inj_2 is free. But that choice does force the order of id _° in map⊎ (else zag does not hold).

```
AdjLeft : (ℓ : Level) → Adjunction (Left ℓ) (Forget ℓ)
AdjLeft ℓ = record
  {unit = record {η = λ _ → inj₁; commute = λ _ → ≡.refl}
  ;counit = record
    {η = λ A → record
      {mor   = [ id , inv A ]  -- ≡ from⊎ ∘ map⊎ idF _°
      ;pres  = [ ≐-refl , ≡.sym ∘ involutive A ]
      }
    ;commute = λ F → [ ≐-refl , ≡.sym ∘ pres F ]
    }
  ;zig = [ ≐-refl , ≐-refl ]
  ;zag = ≡.refl
  }
  -- but there's another!
AdjLeft₂ : (ℓ : Level) → Adjunction (Left ℓ) (Forget ℓ)
```

```
AdjLeft₂ ℓ = record
  {unit = record {η = λ _ → inj₂; commute = λ _ → ≡.refl}
  ; counit = record
    {η = λ A → record
      {mor = [ inv A , id ]        -- ≡ from⊎ ∘ map⊎ _° idF
      ; pres = [ ≡.sym ∘ involutive A , ≐-refl ]
      }
    ; commute = λ F → [ ≡.sym ∘ pres F , ≐-refl ]
    }
  ; zig = [ ≐-refl , ≐-refl ]
  ; zag = ≡.refl
  }
```

[MA: *ToDo :: extract functions out of adjunction proofs!* **]**

Notice that the adjunction proof forces us to come-up with the operations and properties about them!

- **???**

12.4 CoFree Adjunction

```
  -- for the proofs below, we "cheat" and let η for records make things easy.
Right : (ℓ : Level) → Functor (Sets ℓ) (Involutives ℓ)
Right ℓ = record
  {F₀ = λ B → record {A = B × B; _° = swap; involutive = ≐-refl}
  ; F₁ = λ g → record {mor = g ×₁ g; pres = ≐-refl}
  ; identity        = ≐-refl
  ; homomorphism = ≐-refl
  ; F-resp-≡      = λ F≡G a → ≡.cong₂ _,_ (F≡G {proj₁ a}) F≡G
  }
AdjRight : (ℓ : Level) → Adjunction (Forget ℓ) (Right ℓ)
AdjRight ℓ = record
  {unit = record
    {η = λ A → record
      {mor = ⟨ id , inv A ⟩
      ; pres = ≡.cong₂ _,_ ≡.refl ∘ involutive A
      }
    ; commute = λ f → ≡.cong₂ _,_ ≡.refl ∘ ≡.sym ∘ pres f
    }
  ; counit =    record {η = λ _ → proj₁; commute = λ _ → ≡.refl}
  ; zig    =    ≡.refl
  ; zag    =    ≐-refl
  }
  -- MA: and here's another ;)
AdjRight₂ : (ℓ : Level) → Adjunction (Forget ℓ) (Right ℓ)
AdjRight₂ ℓ = record
  {unit = record
    {η = λ A → record
      {mor     = ⟨ inv A , id ⟩
      ; pres = flip (≡.cong₂ _,_) ≡.refl ∘ involutive A
      }
    ; commute = λ f → flip (≡.cong₂ _,_) ≡.refl ∘ ≡.sym ∘ pres f
    }
  ; counit =    record {η = λ _ → proj₂; commute = λ _ → ≡.refl}
  ; zig    =    ≡.refl
```

```
  ; zag    =    ≐-refl
  }
```

Note that we have TWO proofs for AdjRight since we can construe A×A as {(a , a°) | a ∈ A} or as {(a° , a) | a ∈ A} —similarly for why we have two AdjLeft proofs.

[**MA:** *ToDo :: extract functions out of adjunction proofs!*]

Notice that the adjunction proof forces us to come-up with the operations and properties about them!

- ???

12.5 Monad constructions

```
SetMonad : {o : Level} → Monad (Sets o)
SetMonad {o} = Adjunction.monad (AdjLeft o)

InvComonad : {o : Level} → Comonad (Involutives o)
InvComonad {o} = Adjunction.comonad (AdjLeft o)
```

[**MA:** *Prove that free functors are faithful, see Semigroup, and mention monad constructions elsewhere?*]

13 Indexed Unary Algebras

```
module Structures.IndexedUnaryAlgebra where

open import Level renaming (suc to lsuc; zero to lzero; _⊔_ to _⊍_)
open import Categories.Category using (Category; module Category)
open import Categories.Functor using (Functor; Contravariant)
open import Categories.Adjunction using (Adjunction)
open import Categories.Agda using (Sets)
open import Function hiding (_$_)

open import Data.List

open import Forget
open import Function2
  -- open import Structures.Pointed using (Pointeds; Pointed) renaming (Hom to PHom ; MkHom to MkPHom)

open import EqualityCombinators

open import Data.Product using (_×_; _,_)
```

An *I indexed unary algebra* consists of a carrier Q and for each index i : I a unary morphism Op_i : Q → Q on the carrier. In general, an operation of type I × Q → Q is also known as an *I action on Q*, and pop-up in the study of groups and vector spaces.

```
record UnaryAlg {a} (I : Set a) (ℓ : Level) : Set (lsuc ℓ ⊍ a) where
  constructor MkAlg
  field
    Carrier : Set ℓ
    Op : {i : I} → Carrier → Carrier

    -- action form
  _·_ : I → Carrier → Carrier
  i · c = Op {i} c

open UnaryAlg
```

Henceforth we work with a given indexing set,

```
module _ {a} (I : Set a) where    -- Musa: Most likely ought to name this module.
```

Give two unary algebras, over the same indexing set, a morphism between them is a function of their underlying carriers that respects the actions.

```
record Hom {ℓ} (X Y : UnaryAlg I ℓ) : Set (lsuc ℓ ⊍ a) where
  constructor MkHom
  infixr 5 mor
  field
    mor          : Carrier X → Carrier Y
    preservation : {i : I} → mor ∘ Op X {i} ≐ Op Y {i} ∘ mor

open Hom using (mor)
open Hom using () renaming (mor to _$_)   -- override application to take a Hom

  -- arguments can usually be inferred, so implicit variant
preservation : {ℓ : Level} {X Y : UnaryAlg I ℓ} (F : Hom X Y)
  → {i : I} {x : Carrier X} → F $ Op X {i} x ≡ Op Y {i} (F $ x)
preservation F = Hom.preservation F _
```

Notice that the preservation proof looks like a usual homomorphism condition —after excusing the implicits. Rendered in action notation, it would take the shape ∀ {i x} → mor (i · x) ≡ i · mor x with the mor "leap-frogging" over the action. Admiteddly this form is also common and then mor is called an "equivaraint" function, yet this sounds like a new unfamilar concept than it really it: Homomorphism.

Unsuprisngly, the indexed unary algebra's form a category.

```
UnaryAlgCat : (ℓ : Level) → Category (lsuc ℓ ⊍ a) (lsuc ℓ ⊍ a) ℓ
UnaryAlgCat ℓ = record
  {Obj    = UnaryAlg I ℓ
  ; _⇒_   = Hom
  ; _≡_   = λ F G → mor F ≐ mor G
  ; id    = λ {A} → MkHom id ≐-refl
  ; _∘_   = λ {A} {B} {C} F G → MkHom (mor F ∘ mor G) (λ {i} x → let open ≡.≡-Reasoning {A = Carrier C} in begin
    (mor F ∘ mor G ∘ Op A) x
        ≡⟨ ≡.cong (mor F) (preservation G) ⟩
    (mor F ∘ Op B ∘ mor G) x
        ≡⟨ preservation F ⟩
    (Op C ∘ mor F ∘ mor G) x
        ■)
  ; assoc     = ≐-refl
  ; identityˡ = ≐-refl
  ; identityʳ = ≐-refl
  ; equiv = record {IsEquivalence ≐-isEquivalence}
  ; ∘-resp-≡  = λ {A} {B} {C} {F} {G} {H} {K} F≈G H≈K x → let open ≡.≡-Reasoning {A = Carrier C} in begin
    (mor F ∘ mor H) x
        ≡⟨ F≈G _ ⟩
    (mor G ∘ mor H) x
        ≡⟨ ≡.cong (mor G) (H≈K _) ⟩
    (mor G ∘ mor K) x
        ■
  }
  where
    open import Relation.Binary using (IsEquivalence)
```

Needless to say, we can ignore the extra structure to arrive at the underlying carrier.

```
Forget : (ℓ : Level) → Functor (UnaryAlgCat ℓ) (Sets ℓ)
Forget ℓ = record
```

```
{F₀             = Carrier
;F₁             = mor
;identity       = ≡.refl
;homomorphism   = ≡.refl
;F-resp-≡  = λ F≈G {x} → F≈G x
}
```

For each I indexed unary algebra (A, Op) with an elected element a_0 : A, there is a unique homomorpism fold : (List I, _::_) ⟶ (A, Op) sending [] ↦ a_0.

```
module _ (Q : UnaryAlg I a) (q₀ : Carrier Q) where
  open import Data.Unit
  I* : UnaryAlg I a
  I* = MkAlg (List I) (λ {x} xs → x :: xs)

  fold₀ : List I → Carrier Q
  fold₀ []         = q₀
  fold₀ (x :: xs)  = Op Q {x} (fold₀ xs)

  fold : Hom I* Q
  fold = MkHom fold₀ ≐-refl

  fold-point : fold $ [] ≡ q₀
  fold-point = ≡.refl

  fold-unique : (F : Hom I* Q) (point : F $ [] ≡ q₀) → mor F ≐ mor fold
  fold-unique F point [] = point
  fold-unique F point (x :: xs) = let open ≡.≡-Reasoning {A = Carrier Q} in begin
    mor F (x :: xs)
      ≡⟨ preservation F ⟩
    Op Q {x} (mor F xs)
      ≡⟨ induction-hypothesis ⟩
    Op Q {x} (fold₀ xs)
      ∎
      where induction-hypothesis = ≡.cong (Op Q) (fold-unique F point xs)
```

Perhaps it would be better to consider POINTED indexed unary algebras, where this result may be phrased more concisely.

WK: A signature with no constant symbols results in there being no closed terms and so the term algebra is just the empty set of no closed terms quotiented by the given equations and the resulting algebra has an empty carrier.

Free : build over generators – cf Multiset construction in CommMonoid.lagda Initial : does not require generators

ToDo :: mimic the multiset construction here for generators S "over" IndexedUnaryAlgebras. WK claims it may have carrier S × List I; then the non-indexed case is simply List ⊤ ≅ ℕ.

Part IV

Boom Hierarchy

14 Magmas: Binary Trees

Needless to say Binary Trees are a ubiquitous concept in programming. We look at the associate theory and see that they are easy to use since they are a free structure and their associate tool kit of combinators are a result of the proof that they are indeed free. $\boxed{\textbf{???}}$

```
module Structures.Magma where
open import Level renaming (suc to lsuc; zero to lzero)
open import Categories.Category using (Category)
open import Categories.Functor using (Functor)
open import Categories.Adjunction using (Adjunction)
open import Categories.Agda using (Sets)
open import Function using (const; id; _∘_; _$_)
open import Data.Empty
open import Function2 using (_$ᵢ)
open import Forget
open import EqualityCombinators
```

14.1 Definition

A Free Magma is a binary tree.

```
record Magma ℓ : Set (lsuc ℓ) where
  constructor MkMagma
  field
    Carrier : Set ℓ
    Op : Carrier → Carrier → Carrier
open Magma
bop = Magma.Op
syntax bop M x y = x ⟨ M ⟩ y
record Hom {ℓ} (X Y : Magma ℓ) : Set ℓ where
  constructor MkHom
  field
    mor : Carrier X → Carrier Y
    preservation : {x y : Carrier X} → mor (x ⟨ X ⟩ y) ≡ mor x ⟨ Y ⟩ mor y
open Hom
```

14.2 Category and Forgetful Functor

```
MagmaAlg : {ℓ : Level} → OneSortedAlg ℓ
MagmaAlg {ℓ} = record
  {Alg        = Magma ℓ
  ; Carrier   = Carrier
  ; Hom       = Hom
  ; mor       = mor
  ; comp      = λ F G → record
    {mor           = mor F ∘ mor G
    ; preservation = ≡.cong (mor F) (preservation G) ⟨≡≡⟩ preservation F
    }
  ; comp-is-∘ = ≐-refl
  ; Id        = MkHom id ≡.refl
  ; Id-is-id  = ≐-refl
  }
Magmas : (ℓ : Level) → Category (lsuc ℓ) ℓ ℓ
Magmas ℓ = oneSortedCategory ℓ MagmaAlg
Forget : (ℓ : Level) → Functor (Magmas ℓ) (Sets ℓ)
Forget ℓ = mkForgetful ℓ MagmaAlg
```

14.3 Syntax

[**MA:** *Mention free functor and free monads? Syntax.*]

```
data Tree {a : Level} (A : Set a) : Set a where
Leaf : A → Tree A
Branch : Tree A → Tree A → Tree A
rec : {ℓ ℓ′ : Level} {A : Set ℓ} {X : Tree A → Set ℓ′}
  → (leaf : (a : A) → X (Leaf a))
  → (branch : (l r : Tree A) → X l → X r → X (Branch l r))
  → (t : Tree A) → X t
rec lf br (Leaf x) = lf x
rec lf br (Branch l r) = br l r (rec lf br l) (rec lf br r)
⟦_,_⟧ : {a b : Level} {A : Set a} {B : Set b} (L : A → B) (B : B → B → B) → Tree A → B
⟦ L , B ⟧ = rec L (λ _ _ x y → B x y)
map : ∀ {a b} {A : Set a} {B : Set b} → (A → B) → Tree A → Tree B
map f = ⟦ Leaf ∘ f , Branch ⟧   -- cf UnaryAlgebra's map for Eventually
  -- implicits variant of rec
indT : ∀ {a c} {A : Set a} {P : Tree A → Set c}
  → (base : {x : A} → P (Leaf x))
  → (ind : {l r : Tree A} → P l → P r → P (Branch l r))
  → (t : Tree A) → P t
indT base ind = rec (λ a → base) (λ l r → ind)


id-as-⟦⟧ : {ℓ : Level} {A : Set ℓ} → ⟦ Leaf , Branch ⟧ ≐ id {A = Tree A}
id-as-⟦⟧ = indT ≡.refl (≡.cong₂ Branch)
map-∘ : {ℓ : Level} {X Y Z : Set ℓ} {f : X → Y} {g : Y → Z} → map (g ∘ f) ≐ map g ∘ map f
map-∘ = indT ≡.refl (≡.cong₂ Branch)
map-cong : {ℓ : Level} {A B : Set ℓ} {f g : A → B}
  → f ≐ᵢ g
  → map f ≐ map g
map-cong = λ F≈G → indT (≡.cong Leaf F≈G) (≡.cong₂ Branch)
TreeF : (ℓ : Level) → Functor (Sets ℓ) (Magmas ℓ)
TreeF ℓ = record
  {F₀             = λ A → MkMagma (Tree A) Branch
  ;F₁             = λ f → MkHom (map f) ≡.refl
  ;identity       = id-as-⟦⟧
  ;homomorphism   = map-∘
  ;F-resp-≡       = map-cong
  }
eval : {ℓ : Level} (M : Magma ℓ) → Tree (Carrier M) → Carrier M
eval M = ⟦ id , Op M ⟧
eval-naturality : {ℓ : Level} {M N : Magma ℓ} (F : Hom M N)
  → eval N ∘ map (mor F) ≐ mor F ∘ eval M
eval-naturality {ℓ} {M} {N} F = indT ≡.refl $ λ pf₁ pf₂ → ≡.cong₂ (Op N) pf₁ pf₂ ⟨≡≡˘⟩ preservation F
  -- 'eval Trees' has a pre-inverse.
as-id : {ℓ : Level} {A : Set ℓ} → id {A = Tree A} ≐ ⟦ id , Branch ⟧ ∘ map Leaf
as-id = indT ≡.refl (≡.cong₂ Branch)
TreeLeft : (ℓ : Level) → Adjunction (TreeF ℓ) (Forget ℓ)
TreeLeft ℓ = record
  {unit   =   record {η = λ _ → Leaf; commute = λ _ → ≡.refl}
  ;counit =   record
    {η          = λ A → MkHom (eval A) ≡.refl
```

```
    ; commute = eval-naturality
    }
  ; zig  = as-id
  ; zag  = ≡.refl
  }
```

Notice that the adjunction proof forces us to come-up with the operations and properties about them!

- id-as-⟦⟧: $\boxed{???}$
- map: usually functions can be packaged-up to work on trees.
- map-id: the identity function leaves syntax alone; or: map id can be replaced with a constant time algorithm, namely, id.
- map-∘: sequential substitutions on syntax can be efficiently replaced with a single substitution.
- map-cong: observably indistinguishable substitutions can be used in place of one another, similar to the transparency principle of Haskell programs.
- eval : $\boxed{???}$
- eval-naturality : $\boxed{???}$
- as-id : $\boxed{???}$

Looks like there is no right adjoint, because its binary constructor would have to anticipate all magma _*_, so that singleton (x * y) has to be the same as Binary x y.

How does this relate to the notion of "co-trees" —infinitely long trees? –similar to the lists vs streams view.

15 Semigroups: Non-empty Lists

```
module Structures.Semigroup where

open import Level renaming (suc to lsuc; zero to lzero)

open import Categories.Category using (Category)
open import Categories.Functor   using (Functor; Faithful)
open import Categories.Adjunction using (Adjunction)
open import Categories.Agda      using (Sets)
open import Function             using (const; id; _∘_)
open import Data.Product         using (_×_; _,_)

open import Function2 using (_$ᵢ)
open import EqualityCombinators
open import Forget
```

15.1 Definition

A Free Semigroup is a Non-empty list

```
record Semigroup {a} : Set (lsuc a) where
  constructor MkSG
  infixr 5 _*_
  field
    Carrier : Set a
    _*_   : Carrier → Carrier → Carrier
    assoc : {x y z : Carrier} → x * (y * z) ≡ (x * y) * z

open Semigroup renaming (_*_ to Op)
bop = Semigroup._*_
```

```
syntax bop A x y = x ⟨ A ⟩ y

record Hom {ℓ} (Src Tgt : Semigroup {ℓ}) : Set ℓ where
  constructor MkHom
  field
    mor  : Carrier Src → Carrier Tgt
    pres : {x y : Carrier Src} → mor (x ⟨ Src ⟩ y) ≡ (mor x) ⟨ Tgt ⟩ (mor y)

open Hom
```

15.2 Category and Forgetful Functor

```
SGAlg : {ℓ : Level} → OneSortedAlg ℓ
SGAlg = record
  {Alg        = Semigroup
  ; Carrier   = Semigroup.Carrier
  ; Hom       = Hom
  ; mor       = Hom.mor
  ; comp      = λ F G → MkHom (mor F ∘ mor G) (≡.cong (mor F) (pres G) ⟨≡≡⟩ pres F)
  ; comp-is-∘ = ≐-refl
  ; Id        = MkHom id ≡.refl
  ; Id-is-id  = ≐-refl
  }

SemigroupCat : (ℓ : Level) → Category (lsuc ℓ) ℓ ℓ
SemigroupCat ℓ = oneSortedCategory ℓ SGAlg

Forget : (ℓ : Level) → Functor (SemigroupCat ℓ) (Sets ℓ)
Forget ℓ = mkForgetful ℓ SGAlg

Forget-isFaithful : {ℓ : Level} → Faithful (Forget ℓ)
Forget-isFaithful F G F≈G = λ x → F≈G {x}
```

15.3 Free Structure

The non-empty lists constitute a free semigroup algebra.

They can be presented as $\mathsf{X} \times \mathsf{List}\ \mathsf{X}$ or via $\Sigma\, \mathsf{n} : \mathbb{N} \bullet \Sigma\, \mathsf{xs} : \mathsf{Vec}\ \mathsf{n}\ \mathsf{X} \bullet \mathsf{n} \not\equiv 0$. A more direct presentation would be:

```
data List₁ {ℓ : Level} (A : Set ℓ) : Set ℓ where
  [_]  : A → List₁ A
  _::_ : A → List₁ A → List₁ A

rec : {ℓ ℓ′ : Level} {Y : Set ℓ} {X : List₁ Y → Set ℓ′}
  → (wrap : (y : Y) → X [ y ])
  → (cons : (y : Y) (ys : List₁ Y) → X ys → X (y :: ys))
  → (ys : List₁ Y) → X ys
rec w c [ x ]    = w x
rec w c (x :: xs) = c x xs (rec w c xs)

[]-injective : {ℓ : Level} {A : Set ℓ} {x y : A} → [ x ] ≡ [ y ] → x ≡ y
[]-injective ≡.refl = ≡.refl
```

One would expect the second constructor to be an binary operator that we would somehow (setoids!) cox into being associative. However, were we to use an operator, then we would lose canonocity. (Why is it important?)

In some sense, by choosing this particular typing, we are insisting that the operation is right associative.

This is indeed a semigroup,

```
_++_ : {ℓ : Level} {X : Set ℓ} → List₁ X → List₁ X → List₁ X
xs + ys = rec (_:: ys) (λ x xs' res → x :: res) xs
++-assoc : {ℓ : Level} {X : Set ℓ} {xs ys zs : List₁ X}
  → xs + (ys + zs) ≡ (xs + ys) + zs
++-assoc {xs = xs} {ys} {zs} = rec {X = λ xs → xs + (ys + zs) ≡ (xs + ys) + zs} ≐-refl (λ x xs' ind → ≡.cong (x ::_) ind) xs
List₁SG : {ℓ : Level} (X : Set ℓ) → Semigroup {ℓ}
List₁SG X = MkSG (List₁ X) _++_ ++-assoc
```

We can interpret the syntax of a List₁ in any semigroup provided we have a function between the carriers. That is to say, a function of sets is freely lifted to a homomorphism of semigroups.

```
⟦_,_⟧ : {ℓ ℓ′ : Level} {X : Set ℓ} {Y : Set ℓ′}
  → (wrap : X → Y)
  → (op : Y → Y → Y)
  → (List₁ X → Y)
⟦ w , o ⟧ = rec w (λ x xs res → o (w x) res)
  -- lift
list₁ : {ℓ : Level} {X : Set ℓ} {S : Semigroup {ℓ}}
  → (X → Carrier S) → Hom (List₁SG X) S
list₁ {X = X} {S = S} f = MkHom ⟦ f , Op S ⟧ ⟦⟧-over-++
  where H = ⟦ f , Op S ⟧
    ⟦⟧-over-++ : {xs ys : List₁ X} → H (xs + ys) ≡ (H xs) ⟨ S ⟩ (H ys)
    ⟦⟧-over-++ {xs} {ys} = rec {X = λ xs → H (xs + ys) ≡ (H xs) ⟨ S ⟩ (H ys)}
      ≐-refl (λ x xs' ind → ≡.cong (Op S (f x)) ind ⟨≡≡⟩ assoc S) xs
```

In particular, the map operation over lists is:

```
map : {a b : Level} {A : Set a} {B : Set b} → (A → B) → List₁ A → List₁ B
map f = ⟦ [_] ∘ f , _++_ ⟧
```

At the dependent level, we have the induction principle,

```
ind : {a b : Level} {A : Set a} {P : List₁ A → Set b}
     → (base : {x : A} → P [ x ])
     → (ind : {x : A} {xs : List₁ A} → P [ x ] → P xs → P (x :: xs))
     → (xs : List₁ A) → P xs
ind base ind = rec (λ y → base) (λ y ys → ind base)
  -- ind {P = P} base ind [ x ] = base
  -- ind {P = P} base ind (x :: xs) = ind {x} {xs} (base {x}) (ind {P = P} base ind xs)
```

For example, map preserves identity:

```
map-id : {a : Level} {A : Set a} → map id ≐ id {A = List₁ A}
map-id = ind ≡.refl (λ {x} {xs} refl ind → ≡.cong (x ::_) ind)
map-∘ : {ℓ : Level} {A B C : Set ℓ} {f : A → B} {g : B → C}
  → map (g ∘ f) ≐ map g ∘ map f
map-∘ {f = f} {g} = ind ≡.refl (λ {x} {xs} refl ind → ≡.cong ((g (f x)) ::_) ind)
map-cong : {ℓ : Level} {A B : Set ℓ} {f g : A → B}
  → f ≐ g → map f ≐ map g
map-cong {f = f} {g} f≐g = ind (≡.cong [_] (f≐g _))
                               (λ {x} {xs} refl ind → ≡.cong₂ _::_ (f≐g x) ind)
```

15.4 Adjunction Proof

```
Free : (ℓ : Level) → Functor (Sets ℓ) (SemigroupCat ℓ)
Free ℓ = record
  {F₀             = List₁SG
  ; F₁            = λ f → list₁ ([_] ∘ f)
  ; identity      = map-id
  ; homomorphism  = map-∘
  ; F-resp-≡ = λ F≈G → map-cong (λ x → F≈G {x})
  }
Free-isFaithful : {ℓ : Level} → Faithful (Free ℓ)
Free-isFaithful F G F≈G {x} = []-injective (F≈G [ x ])
TreeLeft : (ℓ : Level) → Adjunction (Free ℓ) (Forget ℓ)
TreeLeft ℓ = record
  {unit = record {η = λ _ → [_]; commute = λ _ → ≡.refl}
  ; counit = record
    {η = λ S → list₁ id
    ; commute = λ {X} {Y} F → rec ≐-refl (λ x xs ind → ≡.cong (Op Y (mor F x)) ind ⟨≡≡˘⟩ pres F)
    }
  ; zig = rec ≐-refl (λ x xs ind → ≡.cong (x ::_) ind)
  ; zag = ≡.refl
  }
```

ToDo :: Discuss streams and their realisation in Agda.

15.5 Non-empty lists are trees

```
open import Structures.Magma renaming (Hom to MagmaHom)
open MagmaHom using () renaming (mor to morₘ)
ForgetM : (ℓ : Level) → Functor (SemigroupCat ℓ) (Magmas ℓ)
ForgetM ℓ = record
  {F₀             = λ S → MkMagma (Carrier S) (Op S)
  ; F₁            = λ F → MkHom (mor F) (pres F)
  ; identity      = ≐-refl
  ; homomorphism  = ≐-refl
  ; F-resp-≡ = id
  }
ForgetM-isFaithful : {ℓ : Level} → Faithful (ForgetM ℓ)
ForgetM-isFaithful F G F≈G = λ x → F≈G x
```

Even though there's essentially no difference between the homsets of MagmaCat and SemigroupCat, I "feel" that there ought to be no free functor from the former to the latter. More precisely, I feel that there cannot be an associative "extension" of an arbitrary binary operator; see _⟪_ below.

```
open import Relation.Nullary
open import Categories.NaturalTransformation hiding (id; _≡_)
NoLeft : {ℓ : Level} (FreeM : Functor (Magmas lzero) (SemigroupCat lzero)) → Faithful FreeM → ¬ (Adjunction FreeM (ForgetM lzero)
NoLeft FreeM faithfull Adjunct = ohno (inj-is-injective crash)
  where open Adjunction Adjunct
    open NaturalTransformation
    open import Data.Nat
    open Functor
     {-We expect a free functor to be injective on morphisms, otherwise if it collides functions then it is enforcing equations and t
```

```
_⟪_ : ℕ → ℕ → ℕ
x ⟪ y = x * y + 1
  -- (x ⟪ y) ⟪ z ≡ x * y * z + z + 1
  -- x ⟪ (y ⟪ z) ≡ x * y * z + x + 1
  --
  -- Taking z , x := 1 , 0 yields 2 ≡ 1
  --
  -- The following code realises this pseudo-argument correctly.
ohno : ¬ (2 ≡.≡ 1)
ohno ()
𝒩 : Magma lzero
𝒩 = MkMagma ℕ _⟪_
𝒩 : Semigroup
𝒩 = Functor.F₀ FreeM 𝒩
_⊕_ = Magma.Op (Functor.F₀ (ForgetM lzero) 𝒩)
inj : MagmaHom 𝒩 (Functor.F₀ (ForgetM lzero) 𝒩)
inj = η unit 𝒩
inj₀ = MagmaHom.mor inj
  -- the components of the unit are monic precisely when the left adjoint is faithful
.work : {X Y : Magma lzero} {F G : MagmaHom X Y}
  → morₘ (η unit Y) ∘ morₘ F ≐ morₘ (η unit Y) ∘ morₘ G
  → morₘ F ≐ morₘ G
work {X} {Y} {F} {G} ηF≈ηG =
  let ℳ₀ = Functor.F₀ FreeM
      ℳ   = Functor.F₁ FreeM
      _∘ₘ_ = Category._∘_ (Magmas lzero)
      εY    = mor (η counit (ℳ₀ Y))
      ηY    = η unit Y
  in faithfull F G (begin⟨ ≐-setoid (Carrier (ℳ₀ X)) (Carrier (ℳ₀ Y)) ⟩
  mor (ℳ F) ≈⟨ ∘-≐-cong₁ (mor (ℳ F)) zig ⟩
  (εY ∘ mor (ℳ ηY)) ∘ mor (ℳ F) ≡⟨ ≡.refl ⟩
  εY ∘ (mor (ℳ ηY) ∘ mor (ℳ F)) ≈⟨ ∘-≐-cong₂ εY (≐-sym (homomorphism FreeM)) ⟩
  εY ∘ mor (ℳ (ηY ∘ₘ F)) ≈⟨ ∘-≐-cong₂ εY (F-resp-≡ FreeM ηF≈ηG) ⟩
  εY ∘ mor (ℳ (ηY ∘ₘ G)) ≈⟨ ∘-≐-cong₂ εY (homomorphism FreeM) ⟩
  εY ∘ (mor (ℳ ηY) ∘ mor (ℳ G)) ≡⟨ ≡.refl ⟩
  (εY ∘ mor (ℳ ηY)) ∘ mor (ℳ G) ≈⟨ ∘-≐-cong₁ (mor (ℳ G)) (≐-sym zig) ⟩
  mor (ℳ G) ∎)
  where open import Relation.Binary.SetoidReasoning
postulate inj-is-injective : {x y : ℕ} → inj₀ x ≡ inj₀ y → x ≡ y
open import Data.Unit
𝒯 : Magma lzero
𝒯 = MkMagma ⊤ (λ _ _ → tt)
  --
  -- ⋆ It may be that monics do ¬ correspond to the underlying/mor function being injective for MagmaCat.
  -- ! .cminj-is-injective : {x y : ℕ} → {!!}   -- inj₀ x ≡ inj₀ y → x ≡ y
  -- ! cminj-is-injective {x} {y} = work {𝒯} {𝒩} {F = MkHom (λ x → 0) (λ {{tt} {tt} → {!!}})} {G = {!!}} {!!}
  --
  -- ToDo! . . . perhaps this lives in the libraries someplace?
bad : Hom (Functor.F₀ FreeM (Functor.F₀ (ForgetM _) 𝒩)) 𝒩
bad = η counit 𝒩
crash : inj₀ 2 ≡ inj₀ 1
crash = let open ≡.≡-Reasoning {A = Carrier 𝒩} in begin
  inj₀ 2
    ≡⟨ ≡.refl ⟩
```

```
inj₀ ((0 ⟪ 666) ⟪ 1)
  ≡⟨ MagmaHom.preservation inj ⟩
inj₀ (0 ⟪ 666) ⊕ inj₀ 1
  ≡⟨ ≡.cong (_⊕ inj₀ 1) (MagmaHom.preservation inj) ⟩
(inj₀ 0 ⊕ inj₀ 666) ⊕ inj₀ 1
  ≡⟨ ≡.sym (assoc 𝒩) ⟩
inj₀ 0 ⊕ (inj₀ 666 ⊕ inj₀ 1)
  ≡⟨ ≡.cong (inj₀ 0 ⊕_) (≡.sym (MagmaHom.preservation inj)) ⟩
inj₀ 0 ⊕ inj₀ (666 ⟪ 1)
  ≡⟨ ≡.sym (MagmaHom.preservation inj) ⟩
inj₀ (0 ⟪ (666 ⟪ 1))
  ≡⟨ ≡.refl ⟩
inj₀ 1
  ■
```

16 Monoids: Lists

```
module Structures.Monoid where
open import Level renaming (zero to lzero; suc to lsuc)
open import Data.List using (List; _::_; []; _++_; foldr; map)
open import Categories.Category using (Category)
open import Categories.Functor  using (Functor)
open import Categories.Adjunction using (Adjunction)
open import Categories.Agda     using (Sets)
open import Function            using (id; _∘_; const)
open import Function2           using (_$ᵢ)
open import Forget
open import EqualityCombinators
open import DataProperties
```

16.1 Some remarks about recursion principles

(To be relocated elsewhere)

```
open import Data.List
rcList : {X : Set} {Y : List X → Set} (g₁ : Y []) (g₂ : (x : X) (xs : List X) → Y xs → Y (x :: xs)) → (xs : List X) → Y xs
rcList g₁ g₂ [] = g₁
rcList g₁ g₂ (x :: xs) = g₂ x xs (rcList g₁ g₂ xs)
open import Data.Nat hiding (_*_)
rcℕ : {ℓ : Level} {X : ℕ → Set ℓ} (g₁ : X zero) (g₂ : (n : ℕ) → X n → X (suc n)) → (n : ℕ) → X n
rcℕ g₁ g₂ zero = g₁
rcℕ g₁ g₂ (suc n) = g₂ n (rcℕ g₁ g₂ n)
```

Each constructor $\mathsf{c} : \mathsf{Srcs} \to \mathsf{Type}$ becomes an argument $(\mathsf{ss} : \mathsf{Srcs}) \to \mathsf{X}\ \mathsf{ss} \to \mathsf{X}\ (\mathsf{c}\ \mathsf{ss})$, more or less :-) to obtain a "recursion theorem" like principle. The second piece $\mathsf{X}\ \mathsf{ss}$ may not be possible due to type considerations. Really, the induction principle is just the *dependent* version of folding/recursion!

Observe that if we instead use arguments of the form $\{\mathsf{ss} : \mathsf{Srcs}\} \to \mathsf{X}\ \mathsf{ss} \to \mathsf{X}\ (\mathsf{c}\ \mathsf{ss})$ then, for one reason or another, the dependent type X needs to be supplies explicity –yellow Agda! Hence, it behooves us to use explicits in this case. Sometimes, the yellow cannot be avoided.

16.2 Definition

```
record Monoid ℓ : Set (lsuc ℓ) where
  field
    Carrier : Set ℓ
    Id      : Carrier
    _*_     : Carrier → Carrier → Carrier
    leftId  : {x : Carrier} → Id * x ≡ x
    rightId : {x : Carrier} → x * Id ≡ x
    assoc   : {x y z : Carrier} → (x * y) * z ≡ x * (y * z)

open Monoid

record Hom {ℓ} (Src Tgt : Monoid ℓ) : Set ℓ where
  constructor MkHom
  open Monoid Src renaming (_*_ to _*₁_)
  open Monoid Tgt renaming (_*_ to _*₂_)
  field
    mor : Carrier Src → Carrier Tgt
    pres-Id : mor (Id Src) ≡ Id Tgt
    pres-Op : {x y : Carrier Src} → mor (x *₁ y) ≡ mor x *₂ mor y

open Hom
```

16.3 Category

```
MonoidAlg : {ℓ : Level} → OneSortedAlg ℓ
MonoidAlg {ℓ} = record
  {Alg         = Monoid ℓ
  ; Carrier    = Carrier
  ; Hom        = Hom {ℓ}
  ; mor        = mor
  ; comp       = λ F G → record
    {mor       = mor F ∘ mor G
    ; pres-Id  = ≡.cong (mor F) (pres-Id G) ⟨≡≡⟩ pres-Id F
    ; pres-Op  = ≡.cong (mor F) (pres-Op G) ⟨≡≡⟩ pres-Op F
    }
  ; comp-is-∘  = ≐-refl
  ; Id         = MkHom id ≡.refl ≡.refl
  ; Id-is-id   = ≐-refl
  }

MonoidCat : (ℓ : Level) → Category (lsuc ℓ) ℓ ℓ
MonoidCat ℓ = oneSortedCategory ℓ MonoidAlg
```

16.4 Forgetful Functors $\boxed{???}$

```
  -- Forget all structure, and maintain only the underlying carrier
Forget : (ℓ : Level) → Functor (MonoidCat ℓ) (Sets ℓ)
Forget ℓ = mkForgetful ℓ MonoidAlg

  -- ToDo :: forget to the underlying semigroup

  -- ToDo :: forget to the underlying pointed

  -- ToDo :: forget to the underlying magma

  -- ToDo :: forget to the underlying binary relation, with x ∼ y :≡ (∀ z → x * z ≡ y * z)
    -- the monoid-indistuighability equivalence relation
```

17 Structures.CommMonoid

```
module Structures.CommMonoid where
open import Level renaming (zero to lzero; suc to lsuc; _⊔_ to _⊍_) hiding (lift)
open import Relation.Binary using (Setoid; Rel; _Preserves₂_⟶_⟶_; IsEquivalence)
open import Categories.Category using (Category)
open import Categories.Functor  using (Functor)
open import Categories.Agda     using (Setoids)
open import Data.Product using (Σ; proj₁; proj₂; _,_)
open import Function.Equality using (Π; _⟶_; id; _∘_)
open import Relation.Binary.Sum
import Algebra.FunctionProperties as AFP
open AFP using (Op₂)
```

17.1 Definitions

Some of this is borrowed from the standard library's Algebra.Structures and Algebra. But un-nested and made direct.

Splitting off the properties is useful when defining structures which are commutative-monoid-like, but differ in other ways. The core properties can be re-used.

```
record IsCommutativeMonoid {a ℓ} {A : Set a} (_≈_ : Rel A ℓ)
  (_•_ : Op₂ A) (ε : A) : Set (a ⊍ ℓ) where
  open AFP _≈_
  field
    left-unit  : LeftIdentity ε _•_
    right-unit : RightIdentity ε _•_
    assoc      : Associative _•_
    comm       : Commutative _•_
    _⟨•⟩_ : Congruent₂ _•_
```

There are many equivalent ways of defining a CommMonoid. But it boils down to this: Agda's dependent records are **telescopes**. Sometimes, one wants to identify a particular initial sub-telescope that should be shared between two instances. This is hard (impossible?) to do with holistic records. But if split, via Σ, this becomes easy.

For our purposes, it is very convenient to split the Setoid part of the definition.

```
record CommMonoid {ℓ} {o} (X : Setoid ℓ o) : Set (lsuc ℓ ⊍ lsuc o) where
  constructor MkCommMon
  open Setoid X renaming (Carrier to X₀)
  field
    e    : X₀
    _*_ : X₀ → X₀ → X₀
    isCommMonoid : IsCommutativeMonoid _≈_ _*_ e
  module ≈ = Setoid X
  _⟨≈⟩_ = trans
infix -666 eq-in
eq-in = CommMonoid.≈._≈_
syntax eq-in M x y = x ≈ y : M   -- ghost colon
record Hom {ℓ} {o} (A B : Σ (Setoid ℓ o) CommMonoid) : Set (ℓ ⊍ o) where
  constructor MkHom
```

```
  open Setoid (proj₁ A) using () renaming (_≈_ to _≈₁_; Carrier to A₀)
  open Setoid (proj₁ B) using () renaming (_≈_ to _≈₂_)
  open CommMonoid (proj₂ A) using () renaming (e to e₁; _*_ to _*₁_)
  open CommMonoid (proj₂ B) using () renaming (e to e₂; _*_ to _*₂_)
  field mor : proj₁ A ⟶ proj₁ B
  private mor₀ = Π._⟨$⟩_ mor
  field
    pres-e : mor₀ e₁ ≈₂ e₂
    pres-* : {x y : A₀} → mor₀ (x *₁ y) ≈₂ mor₀ x *₂ mor₀ y
  open Π mor public
```

Notice that the last line in the record, **open** Π mor **public**, lifts the setoid-homomorphism operation _⟨$⟩_ and cong to work on our monoid homomorphisms directly.

17.2 Category and Forgetful Functor

```
MonoidCat : (ℓ o : Level) → Category (lsuc ℓ ⊍ lsuc o) (o ⊍ ℓ) (o ⊍ ℓ)
MonoidCat ℓ o = record
  {Obj = Σ (Setoid ℓ o) CommMonoid
  ; _⇒_ = Hom
  ; _≡_ = λ {{_} {_ , B} F G → ∀ {x} → F ⟨$⟩ x ≈ G ⟨$⟩ x : B}
  ; id = λ {{A , _} → MkHom id (refl A) (refl A)}
  ; _∘_ = λ {{C = _ , C} F G → let open CommMonoid C in record
    {mor    = mor F ∘ mor G
    ; pres-e = (cong F (pres-e G)) ⟨≈⟩ (pres-e F)
    ; pres-* = (cong F (pres-* G)) ⟨≈⟩ (pres-* F)
    }}
  ; assoc = λ {{D = D , _} → refl D}
  ; identityˡ = λ {{_} {B , _} → refl B}
  ; identityʳ = λ {{_} {B , _} → refl B}
  ; equiv = λ {{_} {B , _} → record
    {refl = refl B
    ; sym = λ F≈G → sym B F≈G
    ; trans = λ F≈G G≈H → trans B F≈G G≈H}
    }
  ; ∘-resp-≡ = λ {{C = C , _} {f = F} F≈F' G≈G' → trans C (cong F G≈G') F≈F'}
  }
  where open Hom; open Setoid
```

```
Forget : (ℓ o : Level) → Functor (MonoidCat ℓ o) (Setoids ℓ o)
Forget ℓ o = record
  {F₀           = λ C → record {Setoid (proj₁ C)}
  ; F₁          = λ F → record {Hom F}
  ; identity    = λ {A} → ≈.refl (proj₂ A)
  ; homomorphism = λ {_} {_} {C} → ≈.refl (proj₂ C)
  ; F-resp-≡ = λ F≈G {x} → F≈G {x}
  }
  where open CommMonoid using (module ≈)
```

18 Structures.CommMonoidTerm

```
module Structures.CommMonoidTerm where
```

```
open import Level renaming (zero to lzero; suc to lsuc; _⊔_ to _⊎_) hiding (lift)
open import Relation.Binary using (Setoid; IsEquivalence;
  Reflexive; Symmetric; Transitive)
open import Categories.Category using (Category)
open import Categories.Functor   using (Functor)
open import Categories.Adjunction using (Adjunction)
open import Categories.Agda       using (Setoids)
open import Function.Equality using (Π; _⟶_; id; _∘_)
open import Function2 using (_$ᵢ)
open import Function   using () renaming (id to id₀; _∘_ to _⊚_)
open import Data.List using (List; []; _++_; _∷_; foldr) renaming (map to mapL)
open import Forget
open import EqualityCombinators
open import DataProperties
open import Equiv using (_≃_; id≃; sym≃; trans≃; _⊎≃_; _⟨≃≃⟩_; ≃-setoid; ≃IsEquiv)

record CommMonoid {ℓ} {o} : Set (lsuc ℓ ⊎ lsuc o) where
  constructor MkCommMon
  field setoid : Setoid ℓ o
  open Setoid setoid public
  field
    e          : Carrier
    _*_        : Carrier → Carrier → Carrier
    left-unit  : {x : Carrier} → e * x ≈ x
    right-unit : {x : Carrier} → x * e ≈ x
    assoc      : {x y z : Carrier} → (x * y) * z ≈ x * (y * z)
    comm       : {x y : Carrier} → x * y ≈ y * x
    _⟨*⟩_  : {x y z w : Carrier} → x ≈ y → z ≈ w → x * z ≈ y * w
  module ≈ = Setoid setoid
open CommMonoid hiding (_≈_)
infix -666 eq-in
eq-in = CommMonoid._≈_
syntax eq-in M x y = x ≈ y : M   -- ghost colon
record Hom {ℓ} {o} (A B : CommMonoid {ℓ} {o}) : Set (ℓ ⊎ o) where
  constructor MkHom
  open CommMonoid A using () renaming (e to e₁; _*_ to _*₁_; _≈_ to _≈₁_)
  open CommMonoid B using () renaming (e to e₂; _*_ to _*₂_; _≈_ to _≈₂_)
  field mor   : setoid A ⟶ setoid B
  private mor₀ = Π._⟨$⟩_ mor
  field
    pres-e : mor₀ e₁ ≈₂ e₂
    pres-* : {x y : Carrier A} → mor₀ (x *₁ y) ≈₂ mor₀ x *₂ mor₀ y
  open Π mor public
open Hom
```

Notice that the last line in the record, **open** Π mor **public**, lifts the setoid-homomorphism operation _⟨$⟩_ and cong to work on our monoid homomorphisms directly.

```
MonoidCat : (ℓ o : Level) → Category (lsuc ℓ ⊎ lsuc o) (o ⊎ ℓ) (o ⊎ ℓ)
MonoidCat ℓ o = record
  {Obj = CommMonoid {ℓ} {o}
  ; _⇒_ = Hom
  ; _≡_ = λ {A} {B} F G → {x : Carrier A} → F ⟨$⟩ x ≈ G ⟨$⟩ x : B
```

```
  ; id  =  λ {A} → MkHom id (≈.refl A) (≈.refl A)
  ; _∘_  =  λ {_} {_} {C} F G → record
     {mor    =  mor F ∘ mor G
     ; pres-e  =  ≈.trans C (cong F (pres-e G)) (pres-e F)
     ; pres-*  =  ≈.trans C (cong F (pres-* G)) (pres-* F)
     }
  ; assoc  =  λ {{D  =  D} → ≈.refl D}
  ; identityˡ  =  λ {A} {B} {F} {x} → ≈.refl B
  ; identityʳ  =  λ {A} {B} {F} {x} → ≈.refl B
  ; equiv  =  λ {A} {B} → record
     {refl   =  λ {F} {x} → ≈.refl B
     ; sym  =  λ {F} {G} F≈G {x} → ≈.sym B F≈G
     ; trans  =  λ {F} {G} {H} F≈G G≈H {x} → ≈.trans B F≈G G≈H
     }
  ; ∘-resp-≡  =  λ {A} {B} {C} {F} {F'} {G} {G'} F≈F' G≈G' {x} → ≈.trans C (cong F G≈G') F≈F'
  }
```

```
Forget : (ℓ o : Level) → Functor (MonoidCat ℓ o) (Setoids ℓ o)
Forget ℓ o  =  record
  {F₀                =  λ C → record {CommMonoid C}
  ; F₁               =  λ F → record {Hom F}
  ; identity         =  λ {A} → ≈.refl A
  ; homomorphism  =  λ {A} {B} {C} → ≈.refl C
  ; F-resp-≡  =  λ F≈G {x} → F≈G {x}
  }
```

A “multiset on type X” is a commutative monoid with a to it from X. For now, we make no constraints on the map, however it may be that future proof obligations will require it to be an injection —which is reasonable.

```
record Multiset {ℓ o : Level} (X : Setoid ℓ o) : Set (lsuc ℓ ⊍ lsuc o) where
  field
     commMonoid : CommMonoid {ℓ} {ℓ ⊍ o}
     singleton : Setoid.Carrier X → CommMonoid.Carrier commMonoid
  open CommMonoid commMonoid public
open Multiset
```

A “multiset homomorphism” is a way to lift arbitrary (setoid) functions on the carriers to be homomorphisms on the underlying commutative monoid structure.

```
record MultisetHom {ℓ} {o} {X Y : Setoid ℓ o} (A : Multiset X) (B : Multiset Y) : Set (ℓ ⊍ o) where
  constructor MKMSHom
  field
     lift : (X ⟶ Y) → Hom (commMonoid A) (commMonoid B)
open MultisetHom
```

```
module _ {ℓ o : Level} (X : Setoid ℓ o) where
  X₀  =  Setoid.Carrier X
  infix 5 _•_
  infix 3 _≈ₜ_
     -- syntax of monoids over X
  data Term : Set ℓ where
     inj : X₀ → Term
     ε : Term
     _•_ : Term → Term → Term
```

```
  open Setoid X using () renaming (_≈_ to _≈ₓ_)
  data _≈ₜ_ : Term → Term → Set (ℓ ⊍ o) where
      -- This is an equivalence relation
    ≈ₜ-refl : {t : Term} → t ≈ₜ t
    ≈ₜ-sym   : {s t : Term} → s ≈ₜ t → t ≈ₜ s
    ≈ₜ-trans : {s t u : Term} → s ≈ₜ t → t ≈ₜ u → s ≈ₜ u
      -- where the commutative monoid laws hold
    •-cong : {s t u v : Term} → s ≈ₜ t → u ≈ₜ v → s • u ≈ₜ t • v
    •-assoc : {s t u : Term} → (s • t) • u ≈ₜ s • (t • u)
    •-comm : {s t : Term} → s • t ≈ₜ t • s
    •-leftId : {s : Term} → ε • s ≈ₜ s
    •-rightId : {s : Term} → s • ε ≈ₜ s
      -- and it contains all equailities of the underlying setoid
    embed : {x y : X₀} → x ≈ₓ y → inj x ≈ₜ inj y
        --
        -- This means that we do NOT have unique proofs.
        -- For example, inj x ≈ₜ inj x can be proven in two ways: ≈ₜ-refl or embed ≈ₓ-refl.
        --
        -- This may bite us in the butt; not necessarily though...
  LM : Setoid ℓ (o ⊍ ℓ)
  LM = record
        {Carrier = Term
        ; _≈_ = _≈ₜ_
        ; isEquivalence = record
           {refl  =  ≈ₜ-refl
           ; sym  =  ≈ₜ-sym
           ; trans = ≈ₜ-trans
           }
        }
  ListMS : Multiset X
  ListMS = record
      {commMonoid = record
        {setoid       = LM
        ; e           = ε
        ; _*_         = _•_
        ; left-unit   = •-leftId
        ; right-unit = •-rightId
        ; assoc       = •-assoc
        ; comm        = •-comm
        ; _⟨*⟩_ = •-cong
        }
      ; singleton = inj
      }
      where
  -- Term is functorial
module _ {ℓ o : Level} {X Y : Setoid ℓ o} where
  term-lift : (X ⟶ Y) → Term X → Term Y
  term-lift F (inj x) = inj (Π._⟨$⟩_ F x)
  term-lift F ε = ε
  term-lift F (s • t) = term-lift F s • term-lift F t
  term-cong : (F : X ⟶ Y) → {s t : Term X} → _≈ₜ_ X s t → _≈ₜ_ Y (term-lift F s) (term-lift F t)
  term-cong F ≈ₜ-refl = ≈ₜ-refl
  term-cong F (≈ₜ-sym eq) = ≈ₜ-sym (term-cong F eq)
  term-cong F (≈ₜ-trans eq eq₁) = ≈ₜ-trans (term-cong F eq) (term-cong F eq₁)
  term-cong F (•-cong eq eq₁) = •-cong (term-cong F eq) (term-cong F eq₁)
```

```
   term-cong F •-assoc = •-assoc
   term-cong F •-comm = •-comm
   term-cong F •-leftId = •-leftId
   term-cong F •-rightId = •-rightId
   term-cong F (embed x≈y) = embed (Π.cong F x≈y)

     -- Setoid morphism
   term : (F : X ⟶ Y) → (LM X) ⟶ (LM Y)
   term F = record {_⟨$⟩_ = term-lift F; cong = term-cong F}

   -- proofs that it is functorial; must pattern-match and expand. This is the 'cost' of
   -- going with a term language. Can't put them in the above module either, because
   -- the implicit premises are different.
term-id : ∀ {ℓ o} {X : Setoid ℓ o} {x : Term X} → term-lift id x ≈ x : commMonoid (ListMS X)
term-id {x = inj x} = ≈ₜ-refl
term-id {x = ε} = ≈ₜ-refl
term-id {x = x • x₁} = •-cong (term-id {x = x}) (term-id {x = x₁})

term-Hom : ∀ {ℓ o} {X Y Z : Setoid ℓ o} {f : X ⟶ Y} {g : Y ⟶ Z} {x : Term X} →
   (term-lift (g ∘ f) x) ≈ (term-lift g (term-lift f x)) : commMonoid (ListMS Z)
term-Hom {x = inj x} = ≈ₜ-refl
term-Hom {x = ε} = ≈ₜ-refl
term-Hom {x = x • x₁} = •-cong term-Hom term-Hom

term-resp-F : ∀ {ℓ o} {X Y : Setoid ℓ o} {f g : X ⟶ Y} {x : Term X} →
   ({z : Setoid.Carrier X} → Setoid._≈_ Y (f Π.⟨$⟩ z) (g Π.⟨$⟩ z)) → term-lift f x ≈ term-lift g x : commMonoid (ListMS Y)
term-resp-F {x = inj x} F = embed F
term-resp-F {x = ε} F = ≈ₜ-refl
term-resp-F {x = x • x₁} F = •-cong (term-resp-F F) (term-resp-F F)

ListCMHom : ∀ {ℓ o} (X Y : Setoid ℓ o) → MultisetHom (ListMS X) (ListMS Y)
ListCMHom X Y = MKMSHom (λ F → record
        {mor    = term F
        ; pres-e = ≈ₜ-refl
        ; pres-* = ≈ₜ-refl
        })

   -- We have a fold over the syntax
fold : ∀ {ℓ o} {X : Setoid ℓ o} {B : Set ℓ} →
   let A = Setoid.Carrier X in
   (A → B) → B → (B → B → B) → Term X → B
fold f b g (inj x) = f x
fold f b g ε = b
fold f b g (m • m₁) = g (fold f b g m) (fold f b g m₁)

   -- and an induction principle
ind : ∀ {ℓ o p} → {X : Setoid ℓ o} (P : Term X → Set p) →
   ((x : Setoid.Carrier X) → P (inj x)) → P ε →
   ({t₁ t₂ : Term X} → P t₁ → P t₂ → P (t₁ • t₂)) →
   (t : Term X) → P t
ind P base e₁ bin (inj x) = base x
ind P base e₁ bin ε = e₁
ind P base e₁ bin (t • t₁) = bin (ind P base e₁ bin t) (ind P base e₁ bin t₁)

   -- but the above can be really hard to use in some cases, such as:

fold-resp-≈ : ∀ {ℓ o}
   (CM : CommMonoid {ℓ} {o}) → let X = CommMonoid.setoid CM in {i j : Term X} →
   (i ≈ j : commMonoid (ListMS X)) → (fold id₀ (e CM) (_*_ CM) i) ≈ (fold id₀ (e CM) (_*_ CM) j) : CM
fold-resp-≈ cm ≈ₜ-refl = CommMonoid.refl cm
fold-resp-≈ cm (≈ₜ-sym pf) = CommMonoid.sym cm (fold-resp-≈ cm pf)
fold-resp-≈ cm (≈ₜ-trans pf pf₁) = CommMonoid.trans cm (fold-resp-≈ cm pf) (fold-resp-≈ cm pf₁)
fold-resp-≈ cm (•-cong pf pf₁) = CommMonoid._⟨*⟩_ cm (fold-resp-≈ cm pf) (fold-resp-≈ cm pf₁)
```

```
fold-resp-≈ cm •-assoc = CommMonoid.assoc cm
fold-resp-≈ cm •-comm = CommMonoid.comm cm
fold-resp-≈ cm •-leftId = CommMonoid.left-unit cm
fold-resp-≈ cm •-rightId = CommMonoid.right-unit cm
fold-resp-≈ cm (embed x₁) = x₁
```

It is really important to note that the induction above is on the proof witness.

```
fold-resp-lift : ∀ {ℓ o} {X Y : CommMonoid {ℓ} {o}} (f : Hom X Y) {i : Term (setoid X)} →
  CommMonoid._≈_ Y (fold id₀ (e Y) (_*_ Y) (term-lift (mor f) i)) (mor f Π.⟨$⟩ (fold id₀ (e X) (_*_ X) i))
fold-resp-lift {Y = Y} f {inj x} = CommMonoid.refl Y
fold-resp-lift {Y = Y} f {ε} = CommMonoid.sym Y (pres-e f)
fold-resp-lift {Y = Y} f {i • j} = CommMonoid.trans Y
  (CommMonoid._⟨*⟩_ Y (fold-resp-lift f {i}) (fold-resp-lift f {j}))
  (CommMonoid.sym Y (pres-* f))
fold-singleton : ∀ {ℓ o} {X : Setoid ℓ (o ⊍ ℓ)} {x : Term X} →
  x ≈ (fold {X = LM X} id₀ ε _•_ (term-lift (record {_⟨$⟩_ = inj; cong = embed}) x)) : commMonoid (ListMS X)
fold-singleton {X = X} {x = inj x} = embed (Setoid.refl X)
fold-singleton {x = ε} = ≈ₜ-refl
fold-singleton {ℓ} {o} {X = X} {x = x • x₁} = •-cong (fold-singleton {ℓ} {o} {x = x}) (fold-singleton {ℓ} {o} {x = x₁})


MultisetF : (ℓ o : Level) → Functor (Setoids ℓ o) (MonoidCat ℓ (o ⊍ ℓ))
MultisetF ℓ o = record
  {F₀ = λ S → commMonoid (ListMS S)
  ; F₁ = λ {X} {Y} f → let F = lift (ListCMHom X Y) f in record {Hom F}
  ; identity = term-id
  ; homomorphism = term-Hom
  ; F-resp-≡ = λ F≈G → term-resp-F F≈G
  }
MultisetLeft : (ℓ o : Level) → Adjunction (MultisetF ℓ (o ⊍ ℓ)) (Forget ℓ (o ⊍ ℓ))
MultisetLeft ℓ o = record
  {unit = record {η = λ X → record {_⟨$⟩_ = singleton (ListMS X)
    ; cong = λ {i} {j} i≈j → embed {x = i} {j} i≈j}
    ; commute = λ f → ≈ₜ-refl}
  ; counit = record
    {η = λ {X@ (MkCommMon A z _+_ _ _ _ _ _)} →
      MkHom (record {_⟨$⟩_ = fold id₀ z _+_
        ; cong = fold-resp-≈ X})
        (Setoid.refl A)
        (Setoid.refl A)}
    ; commute = fold-resp-lift
    }
  ; zig = fold-singleton {ℓ} {o}
  ; zag = λ {CM} → CommMonoid.refl CM
  }
  where
    open Multiset
    open CommMonoid
```

19 Structures.AbelianGroup

```
module Structures.AbelianGroup where
open import Level renaming (suc to lsuc; zero to lzero)
```

```
open import Categories.Category using (Category)
open import Categories.Functor   using (Functor)
open import Categories.Adjunction using (Adjunction)
open import Categories.Agda      using (Sets)
open import Function             using (const; id; _∘_; _$_)
open import Function2            using (_$ᵢ)
open import Relation.Unary       using (Pred; _∈_; _∪_; _∩_)

open import EqualityCombinators
open import DataProperties hiding (⊥; ⊥-elim)
open import Data.Empty

open import Algebra hiding (Monoid)
open import Algebra.Structures

open import Data.Nat     using (ℕ; suc) renaming (_+_ to _+ℕ_)
open import Data.Fin     using (Fin; inject+; raise)
open import Data.Integer using (ℤ; _+_; +_; -_)
open import Data.Integer.Properties using (commutativeRing)
```

The retract of an abelian group is an Abelian group; we only show this for the case of ℤ.

```
G1 : ∀ (c : Level) (A : Set c) → AbelianGroup c c
G1 c A = record
  {Carrier = A → ℤ
  ; _≈_    = _≐_
  ; _•_    = λ f g a → f a + g a
  ; ε      = λ _ → + 0
  ; _⁻¹ = λ f a → - f a
  ; isAbelianGroup = record
    {isGroup = record
      {isMonoid = record
        {isSemigroup = record
          {isEquivalence = ≐-isEquivalence
          ; assoc = λ f g h a → AG.+-assoc (f a) (g a) (h a)
          ; •-cong = λ x≈y u≈v a → AG.+-cong (x≈y a) (u≈v a)
          }
        ; identity = (λ h a → proj₁ AG.+-identity (h a)) , (λ h a → proj₂ AG.+-identity (h a))
        }
      ; inverse = (λ h a → proj₁ AG.-‿inverse (h a)) , (λ h a → proj₂ AG.-‿inverse (h a))
      ; ⁻¹-cong = λ i≈j a → ≡.cong (λ z → - z) (i≈j a)
      }
    ; comm = λ f g a → AG.+-comm (f a) (g a)
    }
  }
  where
    module AG = CommutativeRing commutativeRing
```

MA: One of our aims is to live in SET; yet the overall design of AbelianGroup, in the standard library, is via SETOID. Perhaps it would be prudent to make our own SET version? Otherwise, we run into a hybrid of situations such as those below regarding cong and expected derivable preservation properties. –cf the cong in the injective proof of G2 near the bottom.

```
record Hom {o ℓ} (X Y : AbelianGroup o ℓ) : Set (o ⊔ ℓ) where
  constructor MkHom
  open AbelianGroup X using () renaming (_•_ to _+₁_; ε to 0₁; _⁻¹ to -₁_; _≈_ to _≈₁_)
  open AbelianGroup Y using () renaming (_•_ to _+₂_; ε to 0₂; _⁻¹ to -₂_; _≈_ to _≈₂_)
  open AbelianGroup using (Carrier)
```

```
  field
    mor      : Carrier X → Carrier Y
    pres-+   : ∀ x y → mor (x +₁ y) ≡ mor x +₂ mor y
    pres-0   : mor 0 ₁ ≡ 0 ₂
    pres-inv : ∀ x → mor (-₁ x) ≡ -₂ (mor x)

  inv-char : {x y : Carrier Y} → x +₂ y ≈₂ 0 ₂ → y ≈₂ -₂ x
  inv-char {x} {y} x+y≈0 = begin⟨ (record {AbelianGroup Y}) ⟩
    y                 ≈˘⟨ proj₁ identity y              ⟩
    0 ₂ +₂ y          ≈˘⟨ •-cong (proj₁ inverse x) refl ⟩
    (-₂ x +₂ x) +₂ y ≈⟨ assoc _ _ _                     ⟩
      -₂ x +₂ (x +₂ y) ≈⟨ •-cong refl x+y≈0            ⟩
      -₂ x +₂ 0 ₂      ≈⟨ proj₂ identity _              ⟩
    -₂ x              ■
    where open import Relation.Binary.SetoidReasoning
      open AbelianGroup Y

  postulate expected : {x y : Carrier X} → x ≈₁ y → mor x ≈₂ mor y
  pres-inv-redundant : {x : Carrier X} → mor (-₁ x) ≈₂ -₂ (mor x)
  pres-inv-redundant {x} = inv-char (begin⟨ (record {AbelianGroup Y}) ⟩
    mor x +₂ mor (-₁ x) ≡⟨ ≡.sym (pres-+ _ _) ⟩
    mor (x +₁ -₁ x) ≈⟨ expected (proj₂ (AbelianGroup.inverse X) _) ⟩
    mor 0 ₁           ≡⟨ pres-0 ⟩
    0 ₂               ■)
    where open import Relation.Binary.SetoidReasoning
open Hom


AbelianGroupCat : ∀ o ℓ → Category (lsuc (o ⊔ ℓ)) (o ⊔ ℓ) o
AbelianGroupCat o ℓ = record
  {Obj   = AbelianGroup o ℓ
  ; _⇒_  = Hom
  ; _≡_   = λ f g → mor f ≐ mor g
  ; id    = MkHom id (λ _ → ≐-refl) ≡.refl ≐-refl
  ; _∘_  = λ F G → record
    {mor      =  mor F ∘ mor G
    ; pres-+  =  λ x y → ≡.cong (mor F) (pres-+ G x y) ⟨≡≡⟩ pres-+ F (mor G x) (mor G y)
    ; pres-0  =  ≡.cong (mor F) (pres-0 G) ⟨≡≡⟩ pres-0 F
    ; pres-inv =  λ x → ≡.cong (mor F) (pres-inv G x) ⟨≡≡⟩ pres-inv F (mor G x)
    }
  ; assoc     = ≐-refl
  ; identityˡ =    ≐-refl
  ; identityʳ =    ≐-refl
  ; equiv     =    record {IsEquivalence ≐-isEquivalence}
  ; ∘-resp-≡ = ∘-resp-≐
  }
    where open AbelianGroup
        open import Relation.Binary using (IsEquivalence)
Forget : (o ℓ : Level) → Functor (AbelianGroupCat o ℓ) (Sets o)
Forget o ℓ = record
  {F₀              = AbelianGroup.Carrier
  ; F₁             = Hom.mor
  ; identity       = ≡.refl
  ; homomorphism   = ≡.refl
  ; F-resp-≡      =  _$ᵢ
  }


open import Function.Equality using (_⟨$⟩_)
```

```
open import Function.Injection using (Injection; _↣_; module Injection)
open import Relation.Nullary using (¬_)


record DirectSum {o : Level} (A : Set o) : Set (lsuc o) where
  constructor FormalSum
  field
    f : A → ℤ
    B      : Pred A o   -- basis?
    finite : Σ ℕ (λ n → (Σ A B ↣ Fin n))
open DirectSum
```

⁉ • f is the injection of the "Alphabet" as "words" with possibly "negative multiplicity". • B is the subset of all words that contains only the reduced words. • finite is the proof that our construction has finite support.

```
private
    -- JC: why is this defined in Data.Fin.Properties but not exported?
  drop-suc : ∀ {o} {m n : Fin o} → Fin.suc m ≡ Fin.suc n → m ≡ n
  drop-suc ≡.refl = ≡.refl

  inject+-inject : ∀ {m n} → {i j : Fin m} → inject+ n i ≡ inject+ n j → i ≡ j
  inject+-inject {i = Fin.zero} {Fin.zero} ≡.refl = ≡.refl
  inject+-inject {i = Fin.zero} {Fin.suc _} ()
  inject+-inject {i = Fin.suc i} {Fin.zero} ()
  inject+-inject {i = Fin.suc i} {Fin.suc j} pf = ≡.cong Fin.suc (inject+-inject (drop-suc pf))

  raise-inject : ∀ {m n} → {i j : Fin m} → raise n i ≡ raise n j → i ≡ j
  raise-inject {n = ℕ.zero} pf = pf
  raise-inject {n = suc n} pf = raise-inject {n = n} (drop-suc pf)

  raise≢inject+ : (m n : ℕ) (i : Fin m) (j : Fin n) → ¬ (raise n i ≡ inject+ m j)
  raise≢inject+ m (suc n) i Fin.zero ()
  raise≢inject+ m _ i (Fin.suc j) eq = raise≢inject+ m _ i j (drop-suc eq)

  on-right₁ : {ℓ ℓ′ : Level} {A : Set ℓ} {B₁ B₂ : A → Set ℓ′} {a₁ a₂ : A} {b₁ : B₁ a₁} {b₂ : B₁ a₂}
    → (a₁ , b₁) ≡ (a₂ , b₂) → _≡_ {_} {Σ A (B₁ ∪ B₂)} (a₁ , inj₁ b₁) (a₂ , inj₁ b₂)
  on-right₁ ≡.refl = ≡.refl

  on-right₂ : {ℓ ℓ′ : Level} {A : Set ℓ} {B₁ B₂ : A → Set ℓ′} {a₁ a₂ : A} {b₁ : B₂ a₁} {b₂ : B₂ a₂}
    → (a₁ , b₁) ≡ (a₂ , b₂) → _≡_ {_} {Σ A (B₁ ∪ B₂)} (a₁ , inj₂ b₁) (a₂ , inj₂ b₂)
  on-right₂ ≡.refl = ≡.refl
```

The DirectSum datatype furnishes any type with the structure of an AbelianGroup. This is a step in constructing a Free functor.

```
G2 : ∀ (c : Level) (A : Set c) → AbelianGroup (lsuc c) c
G2 c A = record
  {Carrier = DirectSum A
  ; _≈_ = λ a₁ a₂ → f a₁ ≐ f a₂
  ; _•_ = λ {(FormalSum f₁ B₁ (n₁ , fin₁)) (FormalSum f₂ B₂ (n₂ , fin₂)) → record
    {f      = λ e → f₁ e + f₂ e
    ; B      = B₁ ∪ B₂
    ; finite = n₁ +ℕ n₂ , record
      {to          = record
        {_⟨$⟩_ = λ {(a , inj₁ b) → inject+ n₂ (to fin₁ ⟨$⟩ (a , b))
                  ; (a , inj₂ b) → raise n₁ (to fin₂ ⟨$⟩ (a , b))
                  }
        ; cong = λ {{i} {.i} ≡.refl → ≡.refl}
        }
      ; injective = λ {{(a₁ , inj₁ b₁)} {(a₂ , inj₁ b₂)} pf → on-right₁ (injective fin₁ (inject+-inject pf))
```

```
        ; {(a₁ , inj₂ b₁)} {(a₂ , inj₁ b₂)} pf → ⊥-elim (raise≢inject+ _ _ _ _ pf)
        ; {(a₁ , inj₁ b₁)} {(a₂ , inj₂ b₂)} pf → ⊥-elim (raise≢inject+ _ _ _ _ (≡.sym pf))
        ; {(a₁ , inj₂ b₁)} {(a₂ , inj₂ b₂)} pf → on-right₂ (injective fin₂ (raise-inject {n = n₁} pf))
        }
      }
    }}
  ; ε = record
      {f      =  λ _ → + 0
      ; B     =  λ _ → Lift ⊥
      ; finite =  0 , record
        {to = record
          {_⟨$⟩_ = λ {(_ , lift ())}
          ; cong = λ {{i} {.i} ≡.refl → ≡.refl}
          }
        ; injective = λ {{(_ , lift ())}}
        }
      }
  ; _⁻¹ = λ F → FormalSum (λ a → - f F a) (B F) (finite F)
  ; isAbelianGroup = record
    {isGroup = record
      {isMonoid = record
        {isSemigroup = record
          {isEquivalence = record {IsEquivalence ≐-isEquivalence}
          ; assoc = λ F G H a → AG.+-assoc (f F a) (f G a) (f H a)
          ; ∙-cong = λ x≈y u≈v a → AG.+-cong (x≈y a) (u≈v a)
          }
        ; identity = (λ F a → proj₁ AG.+-identity (f F a)) , (λ F a → proj₂ AG.+-identity (f F a))
        }
      ; inverse = (λ F a → proj₁ AG.-‿inverse (f F a)) , (λ F a → proj₂ AG.-‿inverse (f F a))
      ; ⁻¹-cong = λ i≈j a → ≡.cong (λ z → - z) (i≈j a)
      }
    ; comm = λ F G a → AG.+-comm (f F a) (f G a)
    }
  }
  where
    module AG = CommutativeRing commutativeRing
    open DirectSum
    open Injection
    open import Relation.Binary using (IsEquivalence)
```

20 Structures.Multiset

```
module Structures.Multiset where
open import Level renaming (zero to lzero; suc to lsuc; _⊔_ to _⊍_) hiding (lift)
open import Relation.Binary using (Setoid; Rel; IsEquivalence)
  -- open import Categories.Category using (Category)
open import Categories.Functor using (Functor)
open import Categories.Adjunction using (Adjunction)
open import Categories.Agda     using (Setoids)
open import Function.Equality using (Π; _⟶_; id; _∘_)
open import Data.List using (List; []; _++_; _::_; foldr) renaming (map to mapL)
open import Data.List.Properties using (map-++-commute; map-id; map-compose)
open import DataProperties hiding (⟨_,_⟩)
```

```
open import SetoidEquiv
open import ParComp
open import EqualityCombinators
open import Belongs
open import Structures.CommMonoid
```

20.1 CtrSetoid

As will be explained below, the kind of "container" used for building a Multiset needs to support a Setoid-polymorphic equivalence relation.

```
record IsCtrEquivalence {ℓ : Level} (o : Level) (Ctr : Set ℓ → Set ℓ)
    : Set (lsuc ℓ ⊍ lsuc o) where
  field
    equiv : (X : Setoid ℓ o) → Rel (Ctr (Setoid.Carrier X)) (o ⊍ ℓ)
    equivIsEquiv : (X : Setoid ℓ o) → IsEquivalence (equiv X)
```

20.2 Multiset

A "multiset on type X" is a structure on which one can define

- a *commutative monoid* structure,
- implement the concept of *singleton*
- implement the concept of *fold*; note that the name is inspired by its implementation in the main model. Its signature would have suggested "extract", but this would have been quite misleading.

```
record Multiset {ℓ o : Level} (X : Setoid ℓ o) : Set (lsuc ℓ ⊍ lsuc o) where
  open Setoid X renaming (Carrier to X₀)
  open IsCtrEquivalence
  open CommMonoid
  field
    Ctr : Set ℓ → Set ℓ
    Ctr-equiv : IsCtrEquivalence o Ctr
    Ctr-empty : (Y : Set ℓ) → Ctr Y
    Ctr-append : (Y : Set ℓ) → Ctr Y → Ctr Y → Ctr Y
  LIST-Ctr : Setoid ℓ (ℓ ⊍ o)
  LIST-Ctr = record
    {Carrier = Ctr X₀
    ; _≈_ = equiv Ctr-equiv X
    ; isEquivalence = equivIsEquiv Ctr-equiv X
    }
  empty = Ctr-empty X₀
  _+_ = Ctr-append X₀
  field
    MSisCommMonoid : IsCommutativeMonoid (equiv Ctr-equiv X) _+_ empty
  commMonoid : CommMonoid LIST-Ctr
  commMonoid = record
    {e = empty
    ; _*_ = _+_
    ; isCommMonoid = MSisCommMonoid
    }
  field
    singleton : X₀ → Ctr X₀
```

```
    cong-singleton : {i j : X₀} → (i ≈ j) → singleton i ≈ singleton j : commMonoid
    fold : {X : Setoid ℓ o} (CM : CommMonoid X) → let B = Setoid.Carrier X in Ctr B → B
    fold-cong : {YS : Setoid ℓ o} {CM : CommMonoid YS} →
      let Y = Setoid.Carrier YS in
      {i j : Ctr Y}
      → equiv Ctr-equiv YS i j
      → Setoid._≈_ YS (fold CM i) (fold CM j)
    fold-empty : {YS : Setoid ℓ o} {CM : CommMonoid YS} →
      let Y = Setoid.Carrier YS in
      Setoid._≈_ YS (fold CM (Ctr-empty Y)) (e CM)
    fold-+ : {YS : Setoid ℓ o} {CM : CommMonoid YS} →
      let Y = Setoid.Carrier YS in
      let _**_ = _*_ CM in
      {lx ly : Ctr Y} →
      Setoid._≈_ YS (fold CM (Ctr-append Y lx ly)) ((fold CM lx) ** (fold CM ly))
    fold-singleton : {CM : CommMonoid X} → (m : X₀) →
      m ≈ fold CM (singleton m)
```

A “multiset homomorphism” is a way to lift arbitrary (setoid) functions on the carriers to be homomorphisms on the underlying commutative monoid structure, as well as a few compatibility laws.

```
record MultisetHom {ℓ} {o} {X Y : Setoid ℓ (ℓ ⊍ o)} (A : Multiset X) (B : Multiset Y) : Set (lsuc ℓ ⊍ lsuc o) where
  open Multiset
  X₀ = Setoid.Carrier X
  field
    lift : (X ⟶ Y) → Hom (LIST-Ctr A , commMonoid A) (LIST-Ctr B , commMonoid B)
    singleton-commute : (f : X ⟶ Y) {x : X₀} → singleton B (f Π.⟨$⟩ x) ≈
      (Hom.mor (lift f) Π.⟨$⟩ singleton A x) : commMonoid B
    fold-commute : {W : CommMonoid X} {Z : CommMonoid Y} (f : Hom (X , W) (Y , Z))
      {lx : Ctr A X₀} →
      Setoid._≈_ Y (fold B Z (lift (Hom.mor f) Hom.⟨$⟩ lx))
        (Hom.mor f Π.⟨$⟩ (fold A W lx))
open MultisetHom
```

And now something somewhat different: to express that we have the right functoriality properties (and “zap”), we need to assume that we have *constructors* of Multiset and MultisetHom. With these in hand, we can then phrase what extra properties must hold. Because these properties hold at “different types” than the ones for the underlying ones, these cannot go into the above.

```
record FunctorialMSH {ℓ} {o} (MS : (X : Setoid ℓ (ℓ ⊍ o)) → Multiset X)
    (MSH : (X Y : Setoid ℓ (ℓ ⊍ o)) → MultisetHom {ℓ} {o} {X} {Y} (MS X) (MS Y))
    : Set (lsuc ℓ ⊍ lsuc o) where
  open Multiset using (Ctr; commMonoid; Ctr-equiv; fold; singleton; cong-singleton; LIST-Ctr)
  open Hom using (mor; _⟨$⟩_)
  open MultisetHom
  field
    id-pres : {X : Setoid ℓ (ℓ ⊍ o)} {x : Ctr (MS X) (Setoid.Carrier X)}
      → (lift (MSH X X) id) ⟨$⟩ x ≈ x : commMonoid (MS X)
    ∘-pres : {X Y Z : Setoid ℓ (ℓ ⊍ o)} {f : X ⟶ Y} {g : Y ⟶ Z}
      {x : Ctr (MS X) (Setoid.Carrier X)} →
      let gg = lift (MSH Y Z) g in
      let ff = lift (MSH X Y) f in
      mor (lift (MSH X Z) (g ∘ f)) Π.⟨$⟩ x ≈ gg ⟨$⟩ (ff ⟨$⟩ x) : commMonoid (MS Z)
    resp-≈ : {A B : Setoid ℓ (ℓ ⊍ o)} {F G : A ⟶ B}
      (F≈G : {x : Setoid.Carrier A} → (Setoid._≈_ B (F Π.⟨$⟩ x) (G Π.⟨$⟩ x))) →
```

```
    {x : Ctr (MS A) (Setoid.Carrier A)} →
    Hom.mor (lift (MSH A B) F) Π.⟨$⟩ x ≈ Hom.mor (lift (MSH A B) G) Π.⟨$⟩ x : commMonoid (MS B)
  fold-lift-singleton : {X : Setoid ℓ (ℓ ⊍ o)} →
    let ms = MS X in
    let Singleton = record {_⟨$⟩_ = singleton ms; cong = cong-singleton ms} in
    {l : Ctr ms (Setoid.Carrier X)} →
    IsCtrEquivalence.equiv (Ctr-equiv ms) X l
    (fold (MS (LIST-Ctr ms)) (commMonoid ms)
      (Hom.mor (lift (MSH X (LIST-Ctr ms)) Singleton) Π.⟨$⟩ l))
```

Given an implementation of a Multiset as well as of MultisetHom over that, build a Free Functor which is left adjoint to the forgetful functor.

```
module BuildLeftAdjoint (MS : ∀ {ℓ o} (X : Setoid ℓ (ℓ ⊍ o)) → Multiset X)
  (MSH : ∀ {ℓ o} (X Y : Setoid ℓ (ℓ ⊍ o)) → MultisetHom {ℓ} {o} (MS X) (MS {o = o} Y))
  (Func : ∀ {ℓ o} → FunctorialMSH {ℓ} {o} MS MSH) where
  open Multiset
  open MultisetHom
  open FunctorialMSH
  Free : (ℓO ℓ≡ : Level) → Functor (Setoids ℓO (ℓO ⊍ ℓ≡)) (MonoidCat ℓO (ℓO ⊍ ℓ≡))
  Free ℓO ℓ≡ = record
    {F₀ = λ S → LIST-Ctr (MS S) , commMonoid (MS S)
    ; F₁ = λ {X} {Y} f → record {Hom (lift {o = ℓ≡} (MSH X Y) f)}
    ; identity = id-pres Func
    ; homomorphism = ∘-pres Func
    ; F-resp-≡ = resp-≈ Func
    }
  LeftAdjoint : {ℓ o : Level} → Adjunction (Free ℓ o) (Forget ℓ (ℓ ⊍ o))
  LeftAdjoint = record
    {unit = record {η = λ X → record {_⟨$⟩_ = singleton (MS X)
      ; cong = cong-singleton (MS X)}
      ; commute = λ {X} {Y} → singleton-commute (MSH X Y)}
    ; counit = record
      {η = λ {(X , cm) → let M = MS X in
        MkHom (record {_⟨$⟩_ = fold M cm
          ; cong = fold-cong M})
          (fold-empty M {X} {cm}) (fold-+ M {X} {cm})}
      ; commute = λ {{X , _} {Y , _} f → fold-commute (MSH X Y) f}
      }
    ; zig = fold-lift-singleton Func
    ; zag = λ {{X , CM} {m} → fold-singleton (MS X) m}
    }
    where
      open Multiset
      open CommMonoid
```

20.3 An implementation of Multiset using lists with Bag equality

```
module ImplementationViaList {ℓ o : Level} (X : Setoid ℓ o) where
  open Setoid X hiding (refl) renaming (Carrier to X₀)
  open BagEq X using (≡→⇔)
  open ElemOfSing X
  open import Algebra using (Monoid)
```

```
open import Data.List using (monoid)
module ++ = Monoid (monoid (Setoid.Carrier X))
open Membership X using (elem-of)
open ConcatTo⊎S X using (⊎S≅++)
ListMS : Multiset X
ListMS = record
  {Ctr = List
  ; Ctr-equiv = record
    {equiv = λ Y → let open BagEq Y in _⇔_
    ; equivIsEquiv = λ _ → record {refl = ≅-refl; sym = ≅-sym; trans = ≅-trans}
    }
  ; Ctr-empty = λ _ → []
  ; Ctr-append = λ _ → _++_
  ; MSisCommMonoid = record
    {left-unit = λ _ → ≅-refl
    ; right-unit = λ xs → ≡→⇔ (proj₂ ++.identity xs)
    ; assoc    = λ xs ys zs → ≡→⇔ (++.assoc xs ys zs)
    ; comm     = λ xs ys →
      elem-of (xs + ys)           ≅˘⟨ ⊎S≅++ ⟩
      elem-of xs ⊎S elem-of ys  ≅⟨ ⊎S-comm _ _ ⟩
      elem-of ys ⊎S elem-of xs  ≅⟨ ⊎S≅++ ⟩
      elem-of (ys + xs) ■
    ; _⟨•⟩_ = λ {x} {y} {z} {w} x⇔y z⇔w →
        elem-of (x + z)           ≅˘⟨ ⊎S≅++ ⟩
        elem-of x ⊎S elem-of z  ≅⟨ x⇔y ⊎S₁ z⇔w ⟩
        elem-of y ⊎S elem-of w ≅⟨ ⊎S≅++ ⟩
        elem-of (y + w) ■
    }
  ; singleton = λ x → x :: []
  ; cong-singleton = singleton-≈
  ; fold = λ {(MkCommMon e _+_ _) → foldr _+_ e}
  ; fold-cong = λ {_} {CM} → fold-permute {CM = CM}
  ; fold-empty = λ {Y} → Setoid.refl Y
  ; fold-+ = λ {Y} {CM} {lx} {ly} → fold-CM-over-++ {Y} {CM} {lx} {ly}
  ; fold-singleton = λ {CM} m → ≈.sym CM (IsCommutativeMonoid.right-unit (isCommMonoid CM) m)
  }
  where
    open CommMonoid
    open IsCommutativeMonoid using (left-unit)
    fold-CM-over-++ : {Z : Setoid ℓ o} {cm : CommMonoid Z} {lx ly : List (Setoid.Carrier Z)} →
      let F = foldr (_*_ cm) (e cm) in
      F (lx + ly) ≈⌊ Z ⌋ (_*_ cm (F lx) (F ly))
    fold-CM-over-++ {Z} {MkCommMon e₁ _*₁_ isCM₁} {[]} = Setoid.sym Z (left-unit isCM₁ _)
    fold-CM-over-++ {Z} {MkCommMon e₁ _*₁_ isCM₁} {lx = x :: lx} {ly} =
      let F = foldr _*₁_ e₁ in begin⟨ Z ⟩
      x *₁ F (lx + ly)    ≈⟨ refl ⟨•⟩ fold-CM-over-++ {Z} {MkCommMon e₁ _*₁_ isCM₁} {lx} ⟩
      x *₁ (F lx *₁ F ly) ≈⟨ sym-z (assoc x (F lx) (F ly)) ⟩
        (x *₁ F lx) *₁ F ly □
      where
        open IsCommutativeMonoid isCM₁
        open import Relation.Binary.SetoidReasoning renaming (_■ to _□)
        open Setoid Z renaming (sym to sym-z)
    open Locations
    open Membership using (El)
    open ElemOf[]
    fold-permute : {Z : Setoid ℓ o} {CM : CommMonoid Z} {i j : List (Setoid.Carrier Z)} →
```

```
        let open BagEq Z in let open CommMonoid CM renaming (_*_ to _+_; e to e₁) in
        i ⇔ j → foldr _+_ e₁ i ≈⌊ Z ⌋ foldr _+_ e₁ j
      fold-permute {Z} {CM} {[]} {[]} i⇔j = Setoid.refl Z
      fold-permute {Z} {CM} {[]} {x :: j} i⇔j =
        ⊥-elim (elem-of-[] Z (_≅_.from i⇔j Π.⟨$⟩ El (here (Setoid.refl Z))))
      fold-permute {Z} {CM} {x :: i} {[]} i⇔j =
        ⊥-elim (elem-of-[] Z (_≅_.to i⇔j Π.⟨$⟩ El (here (Setoid.refl Z))))
      fold-permute {Z} {CM} {x :: i} {x₁ :: j} i⇔j = {!!}
ListCMHom : ∀ {ℓ o} (X Y : Setoid ℓ (ℓ ⊍ o))
  → MultisetHom {o = o} (ImplementationViaList.ListMS X) (ImplementationViaList.ListMS Y)
ListCMHom {ℓ} {o} X Y = record
  {lift = λ F → let g = Π._⟨$⟩_ F in record
    {mor = record
      {_⟨$⟩_ = mapL g
      ; cong = λ {xs} {ys} xs≈ys →
        elem-of (mapL g xs) ≅⟨ shift-map F xs ⟩
        shifted F xs        ≅⟨ shifted-cong F xs≈ys ⟩
        shifted F ys        ≅˘⟨ shift-map F ys ⟩
        elem-of (mapL g ys) ∎
      }
    ; pres-e =
          elem-of [] ≅˘⟨ ⊥⊥≅elem-of-[] ⟩
          ⊥⊥         ≅⟨ ⊥⊥≅elem-of-[] ⟩
          (elem-of e₁) ∎
        -- in the proof below, *₀ and *₁ are both ++
    ; pres-* = λ {x} {y} →
      elem-of (mapL g (x *₀ y)) ≅⟨ ≡→⇔ (map-++-commute g x y) ⟩
      elem-of (mapL g x *₁ mapL g y) ∎
    }
  ; singleton-commute = λ f {x} → ≅-refl
  ; fold-commute = f-comm
  }
    where
      open ImplementationViaList
      open CommMonoid (Multiset.commMonoid (ListMS X)) renaming (e to e₀; _*_ to _*₀_)
      open CommMonoid (Multiset.commMonoid (ListMS Y)) renaming (e to e₁; _*_ to _*₁_)
      open Membership Y using (elem-of)
      open BagEq Y using (≡→⇔)
      open ElemOfMap
      open ElemOf[] Y
      f-comm : {W : CommMonoid X} {Z : CommMonoid Y} (f : Hom (X , W) (Y , Z))
        {lx : List (Setoid.Carrier X)} →
        Setoid._≈_ Y (foldr (CommMonoid._*_ Z) (CommMonoid.e Z) (mapL (Π._⟨$⟩_ (Hom.mor f)) lx))
          (Hom.mor f Π.⟨$⟩ foldr (CommMonoid._*_ W) (CommMonoid.e W) lx)
      f-comm {MkCommMon e _*_ isCommMonoid₁} {MkCommMon e₂ _*₂_ isCM₂} f {[]} =
        Setoid.sym Y (Hom.pres-e f)
      f-comm {MkCommMon e _*_ isCommMonoid₁} {MkCommMon e₂ _*₂_ isCM₂} f {x :: lx} =
        let g = Π._⟨$⟩_ (Hom.mor f) in begin⟨ Y ⟩
          ((g x) *₂ (foldr _*₂_ e₂ (mapL g lx))) ≈⟨ refl ⟨•⟩ f-comm f {lx} ⟩
          ((g x) *₂ (g (foldr _*_ e lx)))        ≈⟨ sym (Hom.pres-* f) ⟩
          (g (x * foldr _*_ e lx)) □
        where
          open Setoid Y
          open import Relation.Binary.SetoidReasoning using (_≈⟨_⟩_; begin⟨_⟩_) renaming (_∎ to _□)
          open IsCommutativeMonoid isCM₂ using (_⟨•⟩_)
module BuildProperties where
```

```
open ImplementationViaList
functoriality : {ℓ o : Level} → FunctorialMSH {ℓ} {o} ListMS ListCMHom
functoriality {ℓ} {o} = record
  {id-pres = λ {X} {x} → BagEq.≡→⇔ X (map-id x)
  ; ∘-pres = λ {_} {_} {Z} {f} {g} {x} → BagEq.≡→⇔ Z (map-compose x)
  ; resp-≈ = λ {A} {B} {F} {G} F≈G {l} → respect-≈ {F = F} {G} F≈G l
  ; fold-lift-singleton = λ {X} {l} → BagEq.≡→⇔ X (concat-singleton l)
  }
  where
  open Membership
  open Locations using (here; there)
  open Setoid using (Carrier; trans; sym)
  open Multiset using (Ctr; commMonoid)
  respect-≈ : {A B : Setoid ℓ (o ⊍ ℓ)} {F G : A ⟶ B}
    (F≈G : {x : Carrier A} → F Π.⟨$⟩ x ≈⌊ B ⌋ G Π.⟨$⟩ x)
    (lst : Ctr (ListMS A) (Carrier A))
    → mapL (Π._⟨$⟩_ F) lst ≈ mapL (Π._⟨$⟩_ G) lst : commMonoid (ListMS B)
  respect-≈                    F≈G [] = ≅-refl
  respect-≈ {A} {B} {F} {G} F≈G (x :: lst) = record
    {to = record {_⟨$⟩_ = to-G; cong = cong-to-G}
    ; from = record {_⟨$⟩_ = from-G; cong = cong-from-G}
    ; inverse-of = record {left-inverse-of = left-inv; right-inverse-of = right-inv}}
      where
        open LocEquiv B
        f = mapL (Π._⟨$⟩_ F)
        g = mapL (Π._⟨$⟩_ G)
        to-G : {l : List (Carrier A)} → elements B (f l) → elements B (g l)
        to-G {[]} (El ())
        to-G {_ :: _} (El (here sm)) = El (here (trans B sm F≈G))
        to-G {_ :: _} (El (there belongs)) = lift-el B there (to-G (El belongs))
        cong-to-G : {l : List (Carrier A)} {i j : elements B (f l)} → belongs i ≋ belongs j
          → belongs (to-G i) ≋ belongs (to-G j)
        cong-to-G {[]} ()
        cong-to-G {_ :: _} (hereEq x≈z y≈z) = LocEquiv.hereEq (trans B x≈z F≈G) (trans B y≈z F≈G)
        cong-to-G {_ :: _} (thereEq i≋j) = LocEquiv.thereEq (cong-to-G i≋j)
        from-G : {l : List (Carrier A)} → elements B (g l) → elements B (f l)
        from-G {[]} (El ())
        from-G {_ :: _} (El (here sm)) = El (here (trans B sm (sym B F≈G)))
        from-G {_ :: xs} (El (there x₁)) = lift-el B there (from-G (El x₁))
        cong-from-G : {l : List (Carrier A)} {i j : elements B (g l)} → belongs i ≋ belongs j
          → belongs (from-G i) ≋ belongs (from-G j)
        cong-from-G {[]} ()
        cong-from-G {_ :: _} (hereEq x≈z y≈z) = hereEq (trans B x≈z (sym B F≈G)) (trans B y≈z (sym B F≈G))
        cong-from-G {_ :: _} (thereEq loc₁) = thereEq (cong-from-G loc₁)
        left-inv : {l : List (Carrier A)} (y : elements B (mapL (Π._⟨$⟩_ F) l))
          → belongs (from-G (to-G y)) ≋ belongs y
        left-inv {[]} (El ())
        left-inv {_ :: _} (El (here sm)) = hereEq (trans B (trans B sm F≈G) (sym B F≈G)) sm
        left-inv {_ :: _} (El (there belongs₁)) = thereEq (left-inv (El belongs₁))
        right-inv : {l : List (Carrier A)} (y : elements B (mapL (Π._⟨$⟩_ G) l))
          → belongs (to-G (from-G y)) ≋ belongs y
        right-inv {[]} (El ())
        right-inv {_ :: _} (El (here sm)) = hereEq (trans B (trans B sm (sym B F≈G)) F≈G) sm
        right-inv {_ :: _} (El (there belongs₁)) = thereEq (right-inv (El belongs₁))
  concat-singleton : {X : Set ℓ} (lst : List X)
```

```
    → lst ≡ foldr _++_ [] (mapL (λ x → x :: []) lst)
  concat-singleton [] = ≡.refl
  concat-singleton (x :: lst) = ≡.cong (λ z → x :: z) (concat-singleton lst)
```

Last but not least, build the left adjoint:

```
module FreeCommMonoid = BuildLeftAdjoint ImplementationViaList.ListMS ListCMHom
  BuildProperties.functoriality
```

Part V

Setoids

```
module SetoidEquiv where
open import Level renaming (_⊔_ to _⊍_)
open import Relation.Binary using (Setoid)
open import EqualityCombinators using (_≡_; module ≡)


open import Function using (flip)
open import Function.Inverse public using () renaming
  (Inverse to _≅_
  ; id    to ≅-refl
  ; sym to ≅-sym
  )
≅-trans : {a b c ℓa ℓb ℓc : Level} {A : Setoid a ℓa} {B : Setoid b ℓb} {C : Setoid c ℓc}
  → A ≅ B → B ≅ C → A ≅ C
≅-trans = flip Function.Inverse._∘_
infix 3 _■
infixr 2 _≅⟨_⟩_ _≅˘⟨_⟩_
_≅⟨_⟩_ : {x y z ℓx ℓy ℓz : Level} (X : Setoid x ℓx) {Y : Setoid y ℓy} {Z : Setoid z ℓz}
  → X ≅ Y → Y ≅ Z → X ≅ Z
X ≅⟨ X≅Y ⟩ Y≅Z = ≅-trans X≅Y Y≅Z
_≅˘⟨_⟩_ : {x y z ℓx ℓy ℓz : Level} (X : Setoid x ℓx) {Y : Setoid y ℓy} {Z : Setoid z ℓz}
  → Y ≅ X → Y ≅ Z → X ≅ Z
X ≅˘⟨ Y≅X ⟩ Y≅Z = ≅-trans (≅-sym Y≅X) Y≅Z
_■ : {x ℓx : Level} (X : Setoid x ℓx) → X ≅ X
X ■ = ≅-refl
  -- ≅-reflexive
≡→≅ : {a ℓa : Level} {A B : Setoid a ℓa} → A ≡ B → A ≅ B
≡→≅ ≡.refl = ≅-refl


module SetoidOfIsos where
open import Level renaming (_⊔_ to _⊍_)
open import Relation.Binary using (Setoid)
open import Function.Equality using (Π)
open import SetoidEquiv


record _≈_ {a b ℓa ℓb} {A : Setoid a ℓa} {B : Setoid b ℓb} (eq₁ eq₂ : A ≅ B) : Set (a ⊍ b ⊍ ℓa ⊍ ℓb) where
  constructor eq
```

```
  open _≅_
  open Setoid A using () renaming (_≈_ to _≈₁_)
  open Setoid B using () renaming (_≈_ to _≈₂_)
  open Π
  field
    to≈ : ∀ x → to eq₁ ⟨$⟩ x ≈₂ to eq₂ ⟨$⟩ x
    from≈ : ∀ x → from eq₁ ⟨$⟩ x ≈₁ from eq₂ ⟨$⟩ x
module _ {a b ℓa ℓb} {A : Setoid a ℓa} {B : Setoid b ℓb} where
  id≋ : {x : A ≅ B} → x ≋ x
  id≋ = eq (λ _ → Setoid.refl B) (λ _ → Setoid.refl A)
  sym≋ : {i j : A ≅ B} → i ≋ j → j ≋ i
  sym≋ (eq to≈ from≈) = eq (λ x → Setoid.sym B (to≈ x)) (λ x → Setoid.sym A (from≈ x))
  trans≋ : {i j k : A ≅ B} → i ≋ j → j ≋ k → i ≋ k
  trans≋ (eq to≈₀ from≈₀) (eq to≈₁ from≈₁) = eq (λ x → Setoid.trans B (to≈₀ x) (to≈₁ x)) (λ x → Setoid.trans A (from≈₀ x) (from≈₁ x))
_≅S_ : ∀ {a b ℓa ℓb} (A : Setoid a ℓa) (B : Setoid b ℓb) → Setoid (ℓb ⊍ (ℓa ⊍ (b ⊍ a))) (ℓb ⊍ (ℓa ⊍ (b ⊍ a)))
_≅S_ A B = record
  {Carrier = A ≅ B
  ; _≈_ = _≋_
  ; isEquivalence = record {refl = id≋; sym = sym≋; trans = trans≋}}
```

21 SetoidSetoid

```
module SetoidSetoid where
open import Level renaming (zero to lzero; suc to lsuc; _⊔_ to _⊍_) hiding (lift)
open import Relation.Binary         using (Setoid)
open import Function.Equivalence using (Equivalence; id; _∘_; sym)
open import Function                using (flip)
open import DataProperties using (⊤; tt)
open import SetoidEquiv
```

Setoid of proofs ProofSetoid (up to Equivalence), and Setoid of equality proofs in a given setoid.

```
ProofSetoid : (ℓP ℓp : Level) → Setoid (lsuc ℓP ⊍ lsuc ℓp) (ℓP ⊍ ℓp)
ProofSetoid ℓP ℓp = record
  {Carrier        = Setoid ℓP ℓp
  ; _≈_           = Equivalence
  ; isEquivalence = record {refl = id; sym = sym; trans = flip _∘_}
  }
```

Given two elements of a given Setoid A, define a Setoid of equivalences of those elements. We consider all such equivalences to be equivalent. In other words, for e_1 e_2 : Setoid.Carrier A, then $e_1 \approx_s e_2$, as a type, is contractible.

```
_≈S_ : {ℓs ℓS ℓp : Level} {S : Setoid ℓS ℓs} → (e₁ e₂ : Setoid.Carrier S) → Setoid ℓs ℓp
_≈S_ {S = S} e₁ e₂ = let open Setoid S in record
  {Carrier        = e₁ ≈ e₂
  ; _≈_           = λ _ _ → ⊤
  ; isEquivalence = record {refl = tt; sym = λ _ → tt; trans = λ _ _ → tt}
  }
```

21.1 Unions of SetoidFamily

We need a way to put two SetoidFamily "side by side" – a form of parellel composition. To achieve this requires a certain amount of infrastructure: parallel composition of relations, and both disjoint sum and cartesian product of Setoids. So the next couple of sections proceed with that infrastruction, before diving in to the crux of the matter.

```
module SetoidFamilyUnion where

open import Level
open import Relation.Binary    using (Setoid; REL; Rel)
open import Function           using (flip) renaming (id to id₀; _∘_ to _⊚_)
open import Function.Equality using (Π; _⟨$⟩_; cong; id; _⟶_; _∘_)
open import Function.Inverse   using () renaming (_InverseOf_ to Inv)
open import Relation.Binary.Product.Pointwise using (_×-setoid_)

open import Categories.Category using (Category)
open import Categories.Object.Coproduct

open import DataProperties
open import SetoidEquiv
open import ISEquiv
open import ParComp

open import TypeEquiv using (swap₊; swap⋆)
```

21.2 Disjoint parallel composition

The motivation for parallel composition is to lift this to SetoidFamily. But there are two rather different cases. First, a rather straightforward situation when the underlying Setoid are different, there is little choice but to take the union of the Setoids as the Carrier, and everything else follows straightforwardly.

21.2.1 Basic definitions

For some odd reason, the levels of the families must be the same. Even using direct matching (instead of [_,_]).

```
infix 3 _⊎⊎_

_⊎⊎_ : {ℓS ℓs ℓT ℓt ℓA ℓa : Level} {S : Setoid ℓS ℓs} {T : Setoid ℓT ℓt}
  → SetoidFamily S ℓA ℓa → SetoidFamily T ℓA ℓa → SetoidFamily (S ⊎S T) ℓA ℓa
X ⊎⊎ Y = record
  {index = [ A.index , B.index ]
  ; reindex =
    λ {{inj₁ s₁} {inj₁ s₂} (left s₁≈s₂) → record
        {_⟨$⟩_ = _⟨$⟩_ (A.reindex s₁≈s₂)
        ; cong = Π.cong (A.reindex s₁≈s₂)
        }
      ; {inj₂ t₁} {inj₂ t₂} (right t₁≈t₂) → record
        {_⟨$⟩_ = _⟨$⟩_ (B.reindex t₁≈t₂)
        ; cong = Π.cong (B.reindex t₁≈t₂)
        }
      }
  ; id-coh = λ {{inj₁ x} → A.id-coh; {inj₂ y} → B.id-coh}
  ; sym-iso = λ {{inj₁ x} (left r₁) → A.sym-iso r₁; {inj₂ y} (right r₂) → B.sym-iso r₂}
  ; trans-coh = λ {{inj₁ x} (left r₁) (left r₂) → A.trans-coh r₁ r₂
    ; {inj₂ y₂} (right r₂) (right r₃) → B.trans-coh r₂ r₃}
  }
    where
```

```
    module A = SetoidFamily X
    module B = SetoidFamily Y
```

21.2.2 ⊎⊎-comm

```
⊎⊎-comm : {ℓS ℓs ℓT ℓt ℓA ℓa : Level} {S : Setoid ℓS ℓs} {T : Setoid ℓT ℓt}
  {A₁ : SetoidFamily S ℓA ℓa} {A₂ : SetoidFamily T ℓA ℓa}
  → (A₁ ⊎⊎ A₂) ♯ (A₂ ⊎⊎ A₁)
⊎⊎-comm {S = S} {T} {A} {B} = record
  {to = FArr swap⊎ [ (λ _ → id) , (λ _ → id) ]
    (λ {{inj₁ x} {inj₁ y} (left r₁) → Setoid.refl (index A y)
      ; {inj₂ y} {inj₂ z} (right r₂) → Setoid.refl (index B z)}})
  ; from = FArr swap⊎ (λ {(inj₁ x) → id
    ; (inj₂ y) → id})
    (λ {{x = inj₁ x₁} {By = By} (left r₁) → Setoid.refl (index B x₁)
      ; {x = inj₂ x₂} {By = By} (right r₂) → Setoid.refl (index A x₂)}})
  ; left-inv = record
    {ext = λ {(inj₁ x) → left (Setoid.refl T); (inj₂ y) → right (Setoid.refl S)}
    ; transport-ext-coh = λ {(inj₁ x) Bx → Setoid.trans (index B x) (id-coh B) (id-coh B)
      ; (inj₂ y) Ay → Setoid.trans (index A y) (id-coh A) (id-coh A)}
    }
  ; right-inv = record
    {ext = λ {(inj₁ x) → left (Setoid.refl S); (inj₂ y) → right (Setoid.refl T)}
    ; transport-ext-coh = λ {(inj₁ x) Ax → Setoid.trans (index A x) (id-coh A) (id-coh A)
      ; (inj₂ y) By → Setoid.trans (index B y) (id-coh B) (id-coh B)}
    }
  }
  where
    open SetoidFamily
    swap⊎ : ∀ {ℓA ℓa ℓB ℓb} {A : Setoid ℓA ℓa} {B : Setoid ℓB ℓb} → A ⊎S B ⟶ B ⊎S A
    swap⊎ = record
      {_⟨$⟩_ = [ inj₂ , inj₁ ]
      ; cong = λ {(left r₁) → right r₁; (right r₂) → left r₂}
      }
```

21.3 Common-base composition

The second situation is when it is known that the two underlying Setoid are the same (which is actually the case we care more about), in which case things are rather more complex.

```
_⊔⊔_ : {ℓS ℓs ℓA₁ ℓa₁ ℓA₂ ℓa₂ : Level} {S : Setoid ℓS ℓs}
  → SetoidFamily S ℓA₁ ℓa₁ → SetoidFamily S ℓA₂ ℓa₂ → SetoidFamily S (ℓA₁ ⊔ ℓA₂) (ℓa₁ ⊔ ℓa₂)
X ⊔⊔ Y = record
  {index = λ s → A.index s ⊎S B.index s
  ; reindex = λ x≈y → record
    {_⟨$⟩_ = λ {(inj₁ x) → inj₁ (A.reindex x≈y ⟨$⟩ x)
      ; (inj₂ y) → inj₂ (B.reindex x≈y ⟨$⟩ y)}
    ; cong = λ {(left r₁) → left (cong (A.reindex x≈y) r₁)
      ; (right r₂) → right (cong (B.reindex x≈y) r₂)}}
  ; id-coh = λ {{_} {inj₁ x} → left A.id-coh; {_} {inj₂ y} → right B.id-coh}
  ; sym-iso = λ x≈y → record
    {left-inverse-of =
      λ {(inj₁ x) → left (Inv.left-inverse-of (A.sym-iso x≈y) x)
```

```
        ; (inj₂ y) → right (Inv.left-inverse-of (B.sym-iso x≈y) y)}
    ; right-inverse-of =
      λ {(inj₁ x) → left (Inv.right-inverse-of (A.sym-iso x≈y) x)
        ; (inj₂ y) → right (Inv.right-inverse-of (B.sym-iso x≈y) y)}}
  ; trans-coh =
    λ {{b = inj₁ x₁} p q → left (A.trans-coh p q)
      ; {b = inj₂ y₁} p q → right (B.trans-coh p q)}
  }
    where
      module A = SetoidFamily X
      module B = SetoidFamily Y
```

And it is commutative too:

```
⊔⊔-comm : {ℓS ℓs ℓA ℓa ℓB ℓb : Level} {S : Setoid ℓS ℓs}
  {A₁ : SetoidFamily S ℓA ℓa} {A₂ : SetoidFamily S ℓB ℓb}
  → (A₁ ⊔⊔ A₂) ♯ (A₂ ⊔⊔ A₁)
⊔⊔-comm {S = S} {A} {B} = record
  {to = FArr id
    (λ s → record
      {_⟨$⟩_ = swap₊
      ; cong = λ {(left r₁) → right r₁; (right r₂) → left r₂}})
    (λ {{y} {x} {inj₁ x₁} p → right (refl (index A _))
      ; {y} {x} {inj₂ y₁} p → left (refl (index B _))})
  ; from = FArr id
    (λ s → record
      {_⟨$⟩_ = swap₊
      ; cong = λ {(left r₁) → right r₁; (right r₂) → left r₂}})
    (λ {{By = inj₁ x₁} p → right (refl (index B _))
      ; {By = inj₂ y₁} p → left (refl (index A _))})
  ; left-inv = record
    {ext = λ _ → refl S
    ; transport-ext-coh = λ {x (inj₁ x₁) → left (trans (index B x) (id-coh B) (id-coh B))
      ; x (inj₂ y) → right (trans (index A x) (id-coh A) (id-coh A))}}
  ; right-inv = record
    {ext = λ _ → refl S
    ; transport-ext-coh = λ {x (inj₁ x₁) → left (trans (index A x) (id-coh A) (id-coh A))
      ; x (inj₂ y) → right (trans (index B x) (id-coh B) (id-coh B))}}
  }
  where open SetoidFamily; open Setoid
```

21.4 ⊎⊎₁ - parallel composition of equivalences

```
_⊎⊎₁_ : {ℓS ℓs ℓT ℓt ℓU ℓu ℓV ℓv ℓA ℓa ℓC ℓc : Level}
  {S : Setoid ℓS ℓs} {T : Setoid ℓT ℓt} {U : Setoid ℓU ℓu} {V : Setoid ℓV ℓv}
  {A : SetoidFamily S ℓA ℓa} {B : SetoidFamily U ℓA ℓa}
  {C : SetoidFamily T ℓC ℓc} {D : SetoidFamily V ℓC ℓc}
  → A ♯ C → B ♯ D → (A ⊎⊎ B) ♯ (C ⊎⊎ D)
_⊎⊎₁_ {A = A} {B} {C} {D} A♯C B♯D = record
  {to = FArr (record
    {_⟨$⟩_ = [ (λ s → inj₁ (A→C.map ⟨$⟩ s)) , (λ u → inj₂ (B→D.map ⟨$⟩ u)) ]
    ; cong = λ {(left r₁) → left (cong A→C.map r₁)
      ; (right r₂) → right (cong B→D.map r₂)}})
    [ A→C.transport , B→D.transport ]
    (λ {{By = By} (left r₁) → A→C.transport-coh {By = By} r₁
```

```
      ; {By = By} (right r₂) → B→D.transport-coh {By = By} r₂})
  ; from = FArr (record
    {_⟨$⟩_ = _⟨$⟩_ C→A.map ⊎₁ _⟨$⟩_ D→B.map
    ; cong = λ {(left r₁) → left (cong C→A.map r₁)
      ; (right r₂) → right (cong D→B.map r₂)}})
    [ C→A.transport , D→B.transport ]
    (λ {{By = By} (left r₁) → C→A.transport-coh {By = By} r₁
      ; {By = By} (right r₂) → D→B.transport-coh {By = By} r₂})
  ; left-inv = record
    {ext = λ {(inj₁ t) → left (_≈≈_.ext (left-inv A♯C) t)
      ; (inj₂ v) → right (_≈≈_.ext (left-inv B♯D) v)}
    ; transport-ext-coh = λ {(inj₁ x) Bx → _≈≈_.transport-ext-coh (left-inv A♯C) x Bx
      ; (inj₂ y) Bx → _≈≈_.transport-ext-coh (left-inv B♯D) y Bx}}
  ; right-inv = record
    {ext = [ (λ t → left (_≈≈_.ext (right-inv A♯C) t)) ,
      (λ v → right (_≈≈_.ext (right-inv B♯D) v)) ]
    ; transport-ext-coh = λ {(inj₁ x) Bx → _≈≈_.transport-ext-coh (right-inv A♯C) x Bx
      ; (inj₂ y) By → _≈≈_.transport-ext-coh (right-inv B♯D) y By}}
  }
  where
    open _♯_
    open SetoidFamily
    module A→C = _⇛_ (to A♯C)
    module B→D = _⇛_ (to B♯D)
    module C→A = _⇛_ (from A♯C)
    module D→B = _⇛_ (from B♯D)
```

We can make a Category out of a SetoidFamily over a single Setoid. FSSF = Fixed Setoid SetoidFamily. We also fix it so that _⇛_ only contains id-like things.

Begin inactive material

```
FSSF-Cat : {ℓS ℓs ℓA ℓa : Level} (S : Setoid ℓS ℓs) → Category _ _ _
FSSF-Cat {_} {_} {ℓA} {ℓa} S = record
  {Obj = SetoidFamily S ℓA ℓa
  ; _⇒_ = λ B B' → Σ (B ⇛ B') (λ arr → ∀ s → _⇛_.map arr ⟨$⟩ s ≈ s)
  ; _≡_ = λ a₁ a₂ → proj₁ a₁ ≈≈ proj₁ a₂
  ; id = id⇛ , λ _ → refl
  ; _∘_ = λ {(B⇛C , refl₁) (A⇛B , refl₂) → A⇛B ∘⇛ B⇛C , (λ s → trans (refl₁ (_⇛_.map A⇛B ⟨$⟩ s)) (refl₂ s))}
  ; assoc = {!!}   -- λ {{f = f} {g} {h} → assocˡ f g h}
  ; identityˡ = {!!}   -- λ {{f = f} → unitʳ f} – flipped, because ∘⇛ is.
  ; identityʳ = {!!}   -- λ {{f = f} → unitˡ f}
  ; equiv = {!!}   -- record {refl = λ {f} → ≈≈-refl f; sym = ≈≈-sym; trans = _⟨≈≈⟩_}
  ; ∘-resp-≡ = {!!}   -- λ {A} {B} {C} {f} {h} {g} {i} f≈h g≈i → ∘⇛-cong {S = S} {S} {S} {A} {B} {C} {g} {f} {i} {h} g≈i f≈h
  }
  where open Setoid S
```

_⊔⊔_ is? a coproduct for FSSF-Cat.

```
⊔⊔-is-coproduct : {ℓS ℓs ℓA ℓa ℓB ℓb : Level} {S : Setoid ℓS ℓs}
  (A B : SetoidFamily S ℓA ℓa) → Coproduct (FSSF-Cat S) A B
⊔⊔-is-coproduct {S = S} A B = record
  {A+B = A ⊔⊔ B
  ; i₁ = record
    {map = id
    ; transport = λ s → record {_⟨$⟩_ = inj₁; cong = left}
    ; transport-coh = λ {_} {x} _ → left (refl (index A x))
```

```
    }
  ; i₂ = record
    {map = id
    ; transport = λ s → record {_⟨$⟩_ = inj₂; cong = right}
    ; transport-coh = λ {_} {x} _ → right (refl (index B x))
    }
  ; [_,_] = λ {C} A⇛C B⇛C → let
    C⇛B : C ⇛ B   -- putative inverses toA⇛CandB⇛C|
    C⇛B = {!!}
    C⇛A : C ⇛ A
    C⇛A = {!!}
    in record
    {map = map A⇛C
    ; transport = λ sA → let   -- sA is thought of as an index for A.
      sB : Carrier S
      sB = map C⇛B ⟨$⟩ (map A⇛C ⟨$⟩ sA)
      in record
      {_⟨$⟩_ = λ {(inj₁ x) → transport A⇛C sA ⟨$⟩ x
                 ; (inj₂ y) → ?   -- {!transport B⇛C sB ⟨$⟩ y!}
                 }
      ; cong = λ    {(left r₁) → cong (transport A⇛C sA) r₁
                 ; (right r₂) → ?   -- {! cong (transport {!!} sA) r₂ !}
                 }
      }
    ; transport-coh = λ {{By = inj₁ x₁} → {!!}; {By = inj₂ y₁} → {!!}}
    }
  ; commute₁ = record {ext = {!!}; transport-ext-coh = {!!}}
  ; commute₂ = record {ext = {!!}; transport-ext-coh = {!!}}
  ; universal = λ x x₁ → record {ext = {!!}; transport-ext-coh = {!!}}
  }
  where
    open Setoid; open SetoidFamily; open _⇛_
```

However, to make _⊔⊔₁_ "work", the underlying maps in A ♯ C and B ♯ D must be coherent in some way.

```
_⊔⊔₁_ : {ℓS ℓs ℓT ℓt ℓA ℓa ℓC ℓc : Level}
  {S : Setoid ℓS ℓs} {T : Setoid ℓT ℓt}
  {A : SetoidFamily S ℓA ℓa} {B : SetoidFamily S ℓA ℓa}
  {C : SetoidFamily T ℓC ℓc} {D : SetoidFamily T ℓC ℓc}
  → A ♯ C → B ♯ D → (A ⊔⊔ B) ♯ (C ⊔⊔ D)
_⊔⊔₁_ {S = S} {T} {A} {B} {C} {D} A♯C B♯D = record
  {to = FArr A→C.map
    (λ x → record
      {_⟨$⟩_ = λ {(inj₁ Ax) → inj₁ (A→C.transport x ⟨$⟩ Ax)
        ; (inj₂ Bx) → inj₂ (
            reindex D (Setoid.sym T (_≈≈_.ext (left-inv B♯D) (A→C.map ⟨$⟩ x))) ∘ (B→D.transport (D→B.map ⟨$⟩ {!A→C.map ⟨$⟩ x
            -- {!B→D.transport ? ∘ (D→B.transport (A→C.map ⟨$⟩ x)) !}
            )}
      ; cong = {!!}})
    {!!}
  ; from = FArr {!!} {!!} {!!}
  ; left-inv = {!!}
  ; right-inv = {!!}
  }
  where
    open _♯_
    open SetoidFamily
```

```
    module A→C = _⇛_ (to A♯C)
    module B→D = _⇛_ (to B♯D)
    module C→A = _⇛_ (from A♯C)
    module D→B = _⇛_ (from B♯D)
```

End inactive material

We can do product too.

```
_××_ : {ℓS ℓs ℓA₁ ℓa₁ ℓA₂ ℓa₂ : Level} {S : Setoid ℓS ℓs}
  → SetoidFamily S ℓA₁ ℓa₁ → SetoidFamily S ℓA₂ ℓa₂ → SetoidFamily (S ×S S) _ _
X ×× Y = record
  {index = λ s → A.index (proj₁ s) ×S B.index (proj₂ s)
  ; reindex = λ {(x≈y₁ , x≈y₂) → record
    {_⟨$⟩_ = (λ y → A.reindex x≈y₁ ⟨$⟩ y) ×₁ (λ y → B.reindex x≈y₂ ⟨$⟩ y)
    ; cong = λ {(r₁ , r₂) → (Π.cong (A.reindex x≈y₁) r₁) , (Π.cong (B.reindex x≈y₂) r₂)}
    }}
  ; id-coh = A.id-coh , B.id-coh
  ; sym-iso = λ {(x≈y₁ , x≈y₂) → record
    {left-inverse-of = λ {(a , b) → (Inv.left-inverse-of (A.sym-iso x≈y₁) a) ,
                                    (Inv.left-inverse-of (B.sym-iso x≈y₂) b)}
    ; right-inverse-of = λ {(a , b) → (Inv.right-inverse-of (A.sym-iso x≈y₁) a) ,
                                      (Inv.right-inverse-of (B.sym-iso x≈y₂) b)}
    }}
  ; trans-coh = λ {(a≈b₁ , a≈b₂) (b≈c₁ , b≈c₂) → A.trans-coh a≈b₁ b≈c₁ ,
                        B.trans-coh a≈b₂ b≈c₂}
  }
    where
      module A = SetoidFamily X
      module B = SetoidFamily Y
```

And it is commutative too:

```
××-comm : {ℓS ℓs ℓA ℓa ℓB ℓb : Level} {S : Setoid ℓS ℓs}
  {A₁ : SetoidFamily S ℓA ℓa} {A₂ : SetoidFamily S ℓB ℓb}
  → (A₁ ×× A₂) ♯ (A₂ ×× A₁)
××-comm {S = S} {A} {B} = record
  {to = FArr
    (record {_⟨$⟩_ = swap*; cong = swap*})
    (λ _ → record {_⟨$⟩_ = swap*; cong = swap*})
    (λ _ → refl (index B _) , refl (index A _))
  ; from = FArr
    (record {_⟨$⟩_ = swap*; cong = swap*})
    (λ _ → record {_⟨$⟩_ = swap*; cong = swap*})
    (λ _ → refl (index A _) , refl (index B _))
  ; left-inv = record
    {ext = λ _ → refl S , refl S
    ; transport-ext-coh = λ _ _ →
      trans (index B _) (id-coh B) (id-coh B) ,
      trans (index A _) (id-coh A) (id-coh A)}
  ; right-inv = record
    {ext = λ _ → refl S , refl S
    ; transport-ext-coh = λ _ _ →
      (trans (index A _) (id-coh A) (id-coh A)) ,
      (trans (index B _) (id-coh B) (id-coh B))}
  }
  where open SetoidFamily; open Setoid
```

Part VI

Equiv

22 Equiv

```
{-# OPTIONS --without-K #-}
module Equiv where
open import Level using (_⊔_)
open import Function using (_∘_; id)
open import Data.Sum renaming (map to _⊎→_)
open import Data.Product using (Σ; _×_; _,_; proj₁; proj₂) renaming (map to _×→_)
open import Relation.Binary using (IsEquivalence)
open import Relation.Binary.PropositionalEquality
  using (_≡_; refl; sym; trans; cong; cong₂; module ≡-Reasoning)
infix 4 _≐_
infix 3 _≃_
infixr 5 _●_
infix 8 _⊎≃_
infixr 7 _×≃_
  --
  -- Extensional equivalence of (unary) functions
_≐_ : ∀ {ℓ ℓ'} → {A : Set ℓ} {B : Set ℓ'} → (f g : A → B) → Set (ℓ ⊔ ℓ')
_≐_ {A = A} f g = (x : A) → f x ≡ g x
≐-refl : ∀ {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} {f : A → B} → (f ≐ f)
≐-refl _ = refl
≐-sym : ∀ {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} {f g : A → B} → (f ≐ g) → (g ≐ f)
≐-sym H x = sym (H x)
≐-trans : ∀ {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} {f g h : A → B} → (f ≐ g) → (g ≐ h) → (f ≐ h)
≐-trans H G x = trans (H x) (G x)
∘-resp-≐ : ∀ {ℓA ℓB ℓC} {A : Set ℓA} {B : Set ℓB} {C : Set ℓC} {f h : B → C} {g i : A → B} →
  (f ≐ h) → (g ≐ i) → f ∘ g ≐ h ∘ i
∘-resp-≐ {f = f} {i = i} f≐h g≐i x = trans (cong f (g≐i x)) (f≐h (i x))
≐-isEquivalence : ∀ {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} → IsEquivalence (_≐_ {ℓ} {ℓ'} {A} {B})
≐-isEquivalence = record {refl = ≐-refl; sym = ≐-sym; trans = ≐-trans}
  -- generally useful
cong∘l : ∀ {ℓ ℓ' ℓ''} {A : Set ℓ} {B : Set ℓ'} {C : Set ℓ''}
  {g i : A → B} → (f : B → C) →
  (g ≐ i) → (f ∘ g) ≐ (f ∘ i)
cong∘l f g~i x = cong f (g~i x)
cong∘r : ∀ {ℓ ℓ' ℓ''} {A : Set ℓ} {B : Set ℓ'} {C : Set ℓ''}
  {f h : B → C} → (g : A → B) →
  (f ≐ h) → (f ∘ g) ≐ (h ∘ g)
cong∘r g f~h x = f~h (g x)
  --
  -- Quasi-equivalences a la HoTT:
record isqinv {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} (f : A → B) : Set (ℓ ⊔ ℓ') where
  constructor qinv
  field
    g : B → A
    α : (f ∘ g) ≐ id
```

```
    β : (g ∘ f) ≐ id
  -- We explicitly choose quasi-equivalences, even though these
  -- these are not a proposition. This is fine for us, as we're
  -- explicitly looking at equivalences-of-equivalences, and we
  -- so we prefer a symmetric definition over a truncated one.
  --
  -- Equivalences between sets A and B: a function f and a quasi-inverse for f.
_≃_ : ∀ {ℓ ℓ'} → Set ℓ → Set ℓ' → Set (ℓ ⊔ ℓ')
A ≃ B = Σ (A → B) isqinv
id≃ : ∀ {ℓ} {A : Set ℓ} → A ≃ A
id≃ = (id , qinv id (λ _ → refl) (λ _ → refl))
sym≃ : ∀ {ℓ ℓ′} {A : Set ℓ} {B : Set ℓ′} → (A ≃ B) → B ≃ A
sym≃ (A→B , equiv) = e.g , qinv A→B e.β e.α
  where module e = isqinv equiv
abstract
  trans≃ :    ∀ {ℓ ℓ′ ℓ″} {A : Set ℓ} {B : Set ℓ′} {C : Set ℓ″} → A ≃ B → B ≃ C → A ≃ C
  trans≃ {A = A} {B} {C} (f , qinv f⁻¹ fα fβ) (g , qinv g⁻¹ gα gβ) =
    (g ∘ f) , (qinv (f⁻¹ ∘ g⁻¹) (λ x → trans (cong g (fα (g⁻¹ x))) (gα x))
      (λ x → trans (cong f⁻¹ (gβ (f x))) (fβ x)))
    -- more convenient infix version, flipped
  _•_ : ∀ {ℓ ℓ′ ℓ″} {A : Set ℓ} {B : Set ℓ′} {C : Set ℓ″} → B ≃ C → A ≃ B → A ≃ C
  a • b = trans≃ b a
    -- since we're abstract, these all us to do restricted expansion
  β₁ : ∀ {ℓ ℓ′ ℓ″} {A : Set ℓ} {B : Set ℓ′} {C : Set ℓ″} {f : B ≃ C} {g : A ≃ B} →
    proj₁ (f • g) ≐ (proj₁ f ∘ proj₁ g)
  β₁ x = refl
  β₂ : ∀ {ℓ ℓ′ ℓ″} {A : Set ℓ} {B : Set ℓ′} {C : Set ℓ″} {f : B ≃ C} {g : A ≃ B} →
    isqinv.g (proj₂ (f • g)) ≐ (isqinv.g (proj₂ g) ∘ (isqinv.g (proj₂ f)))
  β₂  x = refl
  -- convenient infix version
infixr 5 _⟨≃≃⟩_
_⟨≃≃⟩_ = trans≃
≃IsEquiv : {ℓ : Level.Level} → IsEquivalence {Level.suc ℓ} {ℓ} {Set ℓ} _≃_
≃IsEquiv = record {refl = id≃; sym = sym≃; trans = trans≃}
open import Relation.Binary using (Setoid)
≃-setoid : {ℓ : Level.Level} → Setoid (Level.suc ℓ) ℓ
≃-setoid {ℓ} = record {Carrier = Set ℓ; _≈_ = _≃_; isEquivalence = ≃IsEquiv}
  -- useful throughout below as an abbreviation
gg : ∀ {ℓ ℓ′} {A : Set ℓ} {B : Set ℓ′} → (A ≃ B) → (B → A)
gg z = isqinv.g (proj₂ z)
  -- equivalences are injective
inj≃ : ∀ {ℓ ℓ'} {A : Set ℓ} {B : Set ℓ'} → (eq : A ≃ B) → (x y : A) → (proj₁ eq x ≡ proj₁ eq y → x ≡ y)
inj≃ (f , qinv g α β) x y p = trans
  (sym (β x)) (trans
  (cong g p) (
  β y))
  -- equivalence is a congruence for plus/times
abstract
  private
    _⊎≐_ : ∀ {ℓA ℓB ℓC ℓD} {A : Set ℓA} {B : Set ℓB} {C : Set ℓC} {D : Set ℓD}
      {f : A → C} {finv : C → A} {g : B → D} {ginv : D → B} →
      (α : f ∘ finv ≐ id) → (β : g ∘ ginv ≐ id) →
```

```
      (f ⊎→ g) ∘ (finv ⊎→ ginv) ≐ id {A = C ⊎ D}
    _⊎≐_ α β (inj₁ x) = cong inj₁ (α x)
    _⊎≐_ α β (inj₂ y) = cong inj₂ (β y)
  _⊎≃_ : ∀ {ℓA ℓB ℓC ℓD} {A : Set ℓA} {B : Set ℓB} {C : Set ℓC} {D : Set ℓD}
    → A ≃ C → B ≃ D → (A ⊎ B) ≃ (C ⊎ D)
  (fp , eqp) ⊎≃ (fq , eqq) =
    Data.Sum.map fp fq ,
    qinv (P.g ⊎→ Q.g) (P.α ⊎≐ Q.α) (P.β ⊎≐ Q.β)
    where module P = isqinv eqp
          module Q = isqinv eqq
  β⊎₁ : ∀ {ℓA ℓB ℓC ℓD} {A : Set ℓA} {B : Set ℓB} {C : Set ℓC} {D : Set ℓD}
    → {f : A ≃ C} → {g : B ≃ D} → proj₁ (f ⊎≃ g) ≐ Data.Sum.map (proj₁ f) (proj₁ g)
  β⊎₁ _ = refl
  β⊎₂ : ∀ {ℓA ℓB ℓC ℓD} {A : Set ℓA} {B : Set ℓB} {C : Set ℓC} {D : Set ℓD}
    → {f : A ≃ C} → {g : B ≃ D} → gg (f ⊎≃ g) ≐ Data.Sum.map (gg f) (gg g)
  β⊎₂ _ = refl
  -- ⊗
abstract
  private
    _×≐_ : ∀ {ℓA ℓB ℓC ℓD} {A : Set ℓA} {B : Set ℓB} {C : Set ℓC} {D : Set ℓD}
      {f : A → C} {finv : C → A} {g : B → D} {ginv : D → B} →
      (α : f ∘ finv ≐ id) → (β : g ∘ ginv ≐ id) →
      (f ×→ g) ∘ (finv ×→ ginv) ≐ id {A = C × D}
    _×≐_ α β (x , y) = cong₂ _,_ (α x) (β y)
  _×≃_ : ∀ {ℓA ℓB ℓC ℓD} {A : Set ℓA} {B : Set ℓB} {C : Set ℓC} {D : Set ℓD}
    → A ≃ C → B ≃ D → (A × B) ≃ (C × D)
  (fp , eqp) ×≃ (fq , eqq) =
    Data.Product.map fp fq ,
    qinv
      (P.g ×→ Q.g)
      (_×≐_ {f = fp} {g = fq} P.α Q.α)
      (_×≐_ {f = P.g} {g = Q.g} P.β Q.β)
    where module P = isqinv eqp
          module Q = isqinv eqq
  β×₁ : ∀ {ℓA ℓB ℓC ℓD} {A : Set ℓA} {B : Set ℓB} {C : Set ℓC} {D : Set ℓD}
    → {f : A ≃ C} → {g : B ≃ D} → proj₁ (f ×≃ g) ≐ Data.Product.map (proj₁ f) (proj₁ g)
  β×₁ _ = refl
  β×₂ : ∀ {ℓA ℓB ℓC ℓD} {A : Set ℓA} {B : Set ℓB} {C : Set ℓC} {D : Set ℓD}
    → {f : A ≃ C} → {g : B ≃ D} → gg (f ×≃ g) ≐ Data.Product.map (gg f) (gg g)
  β×₂ _ = refl
```

23 Indexed Setoid Equivalence

```
module ISEquiv where
open import Level using (Level; suc; _⊔_)
open import Relation.Binary using (Setoid)
open import Function.Inverse using (_InverseOf_) renaming (Inverse to _≅_; id to ≅-refl)
open import Function.Equality using (_⟨$⟩_; _⟶_; Π; id; _∘_)
```

A SetoidFamily (over a Setoid S), is a family of Setoids indexed by the carrier of S, along with a way to "reindex" between equivalent members of S. reindex works as expected with respect to the the equivalences of S.

```
record SetoidFamily {ℓS ℓs : Level} (S : Setoid ℓS ℓs) (ℓA ℓa : Level) : Set (ℓS ⊔ ℓs ⊔ suc (ℓA ⊔ ℓa)) where
  open Setoid using () renaming (Carrier to |_|)
  open Setoid S using (_≈_; refl; sym; trans)
  field
    index : | S | → Setoid ℓA ℓa
    reindex : {x y : | S |} → x ≈ y → index x ⟶ index y
    id-coh : {a : | S |} {b : | index a |} → Setoid._≈_ (index a) (reindex refl ⟨$⟩ b) b
    sym-iso : {x y : | S |} → (p : x ≈ y) → reindex (sym p) InverseOf reindex p
    trans-coh : {x y z : | S |} {b : | index x |} → (p : x ≈ y) → (q : y ≈ z) →
      Setoid._≈_ (index z) (reindex (trans p q) ⟨$⟩ b)
        (reindex q ∘ reindex p ⟨$⟩ b)
```

A map _⇛_ of SetoidFamily is a map (aka _⟶_) of the underlying setoids, and transport, a method of mapping from index B x to the setoid obtained by shifting from one Setoid to another, i.e. index B' (map ⟨$⟩ x). Lastly, transport and reindex obey a commuting law.

```
record _⇛_ {ℓS ℓs ℓA ℓa ℓS' ℓs' ℓA' ℓa' : Level} {S : Setoid ℓS ℓs} {S' : Setoid ℓS' ℓs'}
(B : SetoidFamily S ℓA ℓa) (B' : SetoidFamily S' ℓA' ℓa') :
  Set (ℓS ⊔ ℓA ⊔ ℓS' ⊔ ℓA' ⊔ ℓa' ⊔ ℓs ⊔ ℓs' ⊔ ℓa) where
constructor FArr
open SetoidFamily
open Setoid using () renaming (Carrier to |_|)
open Setoid S using (_≈_)
field
  map : S ⟶ S'
  transport : (x : | S |) → index B x ⟶ index B' (map ⟨$⟩ x)
  transport-coh : {y x : | S |} {By : | index B y |} → (p : y ≈ x) →
    Setoid._≈_ (index B' (map ⟨$⟩ x))
      (transport x ⟨$⟩ (reindex B p ⟨$⟩ By))
      (reindex B' (Π.cong map p) ⟨$⟩ (transport y ⟨$⟩ By))
```

We say that two maps F and G are equivalent (written F ≈≈ G) if there is an (extensional) equivalence between the underlying Setoid maps, and a certain coherence law.

```
infix 3 _≈≈_
infixr 3 _⟨≈≈⟩_
record _≈≈_ {ℓS ℓs ℓA ℓa ℓS' ℓs' ℓA' ℓa' : Level} {S : Setoid ℓS ℓs} {S' : Setoid ℓS' ℓs'}
  {B : SetoidFamily S ℓA ℓa} {B' : SetoidFamily S' ℓA' ℓa'}
  (F : B ⇛ B') (G : B ⇛ B') : Set (ℓA ⊔ ℓS ⊔ ℓs' ⊔ ℓa') where
    open Setoid using () renaming (Carrier to |_|)
    open Setoid S using () renaming (_≈_ to _≈₁_)
    open Setoid S' using () renaming (_≈_ to _≈₂_)
    open SetoidFamily
    open _⇛_
    field
      ext : (x : | S |) → map G ⟨$⟩ x ≈₂ map F ⟨$⟩ x
      transport-ext-coh : (x : | S |) (Bx : | index B x |) →
        Setoid._≈_ (index B' (map F ⟨$⟩ x))
          (reindex B' (ext x) ⟨$⟩ (transport G x ⟨$⟩ Bx))
          (transport F x ⟨$⟩ Bx)
```

_≈≈_ is an equivalence relation.

```
≈≈-refl : {ℓS ℓs ℓT ℓt ℓA ℓa ℓB ℓb : Level} {S : Setoid ℓS ℓs} {T : Setoid ℓT ℓt}
  {A : SetoidFamily S ℓA ℓa} {B : SetoidFamily T ℓB ℓb}
  (F : A ⇛ B) → F ≈≈ F
```

```
≈≈-refl {T = T} {B = B} F = record
  {ext = λ _ → refl; transport-ext-coh = λ x Bx → id-coh {map F ⟨$⟩ x} {transport F x ⟨$⟩ Bx}}
  where open Setoid T; open SetoidFamily B; open _⇛_
≈≈-sym : {ℓS ℓs ℓA ℓa ℓS' ℓs' ℓA' ℓa' : Level} {S : Setoid ℓS ℓs} {S' : Setoid ℓS' ℓs'}
  {B : SetoidFamily S ℓA ℓa} {B' : SetoidFamily S' ℓA' ℓa'}
  {F : B ⇛ B'} {G : B ⇛ B'} → F ≈≈ G → G ≈≈ F
≈≈-sym {S = S} {S'} {B} {B'} {F} {G} record {ext = ext; transport-ext-coh = tec} = record
  {ext = λ x → sym (ext x)
  ; transport-ext-coh = λ x Bx → Setoid.trans (index (map G ⟨$⟩ x))
    (Setoid.sym (index (map G ⟨$⟩ x)) (Π.cong (reindex (sym (ext x))) (tec x Bx)))
    ((left-inverse-of (sym-iso (ext x)) (transport G x ⟨$⟩ Bx)))}
  where
    open SetoidFamily B'
    open _InverseOf_
    open Setoid S'
    open _⇛_
_⟨≈≈⟩_ : {ℓS ℓs ℓA ℓa ℓS' ℓs' ℓA' ℓa' : Level}
  {S : Setoid ℓS ℓs} {S' : Setoid ℓS' ℓs'}
  {B : SetoidFamily S ℓA ℓa} {B' : SetoidFamily S' ℓA' ℓa'}
  {F : B ⇛ B'} {G : B ⇛ B'} {H : B ⇛ B'} → F ≈≈ G → G ≈≈ H → F ≈≈ H
_⟨≈≈⟩_ {S' = S'} {B} {B'} {F} {G} {H} F≈G G≈H = record
  {ext = λ x → trans (G=H.ext x) (F=G.ext x)
  ; transport-ext-coh = λ x Bx →
    let open Setoid (index B' (_⇛_.map F ⟨$⟩ x)) renaming (trans to _⟨≈⟩_) in
    (SetoidFamily.trans-coh B' (G=H.ext x) (F=G.ext x) ⟨≈⟩
    (Π.cong (reindex B' (F=G.ext x)) (G=H.transport-ext-coh x Bx))) ⟨≈⟩
    (F=G.transport-ext-coh x Bx)
  }
  where
    open Setoid S'
    open SetoidFamily
    module F=G = _≈≈_ F≈G
    module G=H = _≈≈_ G≈H
```

If ⇛ is going to be a proper notion of mapping, it should at least have an identity map as well as composition. [We might expect more, that it can all be packaged as a Category. It can, but we don't need it, so we do just the parts that are needed.

```
id⇛ : {ℓS ℓs ℓA ℓa : Level} {S : Setoid ℓS ℓs}
{B : SetoidFamily S ℓA ℓa} → B ⇛ B
id⇛ {S = S} {B} =
  FArr id (λ _ → reindex refl)
      (λ {y} {x} {By} y≈x → Setoid.trans (index x)
        id-coh
        (Π.cong (reindex y≈x) (Setoid.sym (index y) (id-coh {y} {By}))))
    where
      open SetoidFamily B
      open Setoid S
infixr 9 _∘⇛_
_∘⇛_ : {ℓS ℓs ℓT ℓt ℓU ℓu ℓA ℓa ℓB ℓb ℓC ℓc : Level}
{S : Setoid ℓS ℓs} {T : Setoid ℓT ℓt} {U : Setoid ℓU ℓu}
{A : SetoidFamily S ℓA ℓa} {B : SetoidFamily T ℓB ℓb} {C : SetoidFamily U ℓC ℓc} →
(A ⇛ B) → (B ⇛ C) → (A ⇛ C)
_∘⇛_ {A = A} {B} {C} A⇛B B⇛C = FArr (G.map ∘ F.map) (λ x → G.transport (F.map ⟨$⟩ x) ∘ F.transport x)
  (λ {y} {x} {By} y≈x →
  let open Setoid (index C (G.map ∘ F.map ⟨$⟩ x)) renaming (trans to _⟨≈⟩_) in
```

```
  Π.cong (G.transport (F.map ⟨$⟩ x)) (F.transport-coh {By = By} y≈x) ⟨≈⟩
  G.transport-coh (Π.cong F.map y≈x))
  where
    module F = _⇛_ A⇛B
    module G = _⇛_ B⇛C
    open SetoidFamily
```

Lastly, we need to know when two SetoidFamily are equivalent. In fact, we'll use a quasi-equivalence (we have no need for it to be a proposition). So we'll need two maps back and forth, and show that they compose to the identity, up to equivalence of maps.

```
infix 3 _♯_
record _♯_ {ℓS ℓs ℓA ℓa ℓS' ℓs' ℓA' ℓa' : Level} {S : Setoid ℓS ℓs} {S' : Setoid ℓS' ℓs'}
(From : SetoidFamily S ℓA ℓa) (To : SetoidFamily S' ℓA' ℓa')
: Set (ℓS ⊔ ℓA ⊔ ℓS' ⊔ ℓs ⊔ ℓa ⊔ ℓA' ⊔ ℓs' ⊔ ℓa') where
field
  to        : From ⇛ To
  from      : To ⇛ From
  left-inv  : from ∘⇛ to ≈≈ id⇛ {B = To}
  right-inv : to ∘⇛ from ≈≈ id⇛ {B = From}
```

We need to show that _♯_ is also an equivalence relation too. This relies on some properties of ∘⇛ and id⇛, so we prove these first. We could prove less general versions of left-unital and right-unital, but these are easy enough.

We'll also need that ∘⇛ is associative and a congruence. For associativity, giving the arguments helps inference; not sure how crucial this is, but as it is not too painful, let's see.

```
unitˡ : {ℓS ℓs ℓA ℓa ℓS' ℓs' ℓA' ℓa' : Level} {S : Setoid ℓS ℓs} {S' : Setoid ℓS' ℓs'}
{B : SetoidFamily S ℓA ℓa} {B' : SetoidFamily S' ℓA' ℓa'} (F : B ⇛ B') →
id⇛ ∘⇛ F ≈≈ F
unitˡ {S = S} {S'} {B} {B'} F = record
  {ext = λ _ → Setoid.refl S'
  ; transport-ext-coh = λ x Bx →
    let T = index B' (_⇛_.map F ⟨$⟩ x) in
    let open Setoid T renaming (refl to reflT; sym to symT; trans to _⟨≈⟩_) in
    id-coh B' ⟨≈⟩ symT (Π.cong (_⇛_.transport F x) (id-coh B))}
  where open SetoidFamily
unitʳ : {ℓS ℓs ℓA ℓa ℓS' ℓs' ℓA' ℓa' : Level} {S : Setoid ℓS ℓs} {S' : Setoid ℓS' ℓs'}
{B : SetoidFamily S ℓA ℓa} {B' : SetoidFamily S' ℓA' ℓa'} (F : B ⇛ B') →
F ∘⇛ id⇛ ≈≈ F
unitʳ {S = S} {S'} {B} {B'} F = record
  {ext = λ _ → Setoid.refl S'
  ; transport-ext-coh = λ x Bx →
    let T = index B' (_⇛_.map F ⟨$⟩ x) in
    let open Setoid T renaming (trans to _⟨≈⟩_) in
    id-coh B' ⟨≈⟩ sym (id-coh B')}
  where open SetoidFamily
assocˡ : {ℓS ℓs ℓT ℓt ℓU ℓu ℓA ℓa ℓB ℓb ℓC ℓc ℓV ℓv ℓD ℓd : Level}
{S : Setoid ℓS ℓs} {T : Setoid ℓT ℓt} {U : Setoid ℓU ℓu} {V : Setoid ℓV ℓv}
{A : SetoidFamily S ℓA ℓa} {B : SetoidFamily T ℓB ℓb} {C : SetoidFamily U ℓC ℓc} {D : SetoidFamily V ℓD ℓd}
(F : A ⇛ B) (G : B ⇛ C) (H : C ⇛ D) → F ∘⇛ (G ∘⇛ H) ≈≈ (F ∘⇛ G) ∘⇛ H
assocˡ {V = V} {_} {_} {_} {D} F G H = record
  {ext = λ _ → Setoid.refl V; transport-ext-coh = λ _ _ → SetoidFamily.id-coh D}
assocʳ : {ℓS ℓs ℓT ℓt ℓU ℓu ℓA ℓa ℓB ℓb ℓC ℓc ℓV ℓv ℓD ℓd : Level}
```

```
{S : Setoid ℓS ℓs} {T : Setoid ℓT ℓt} {U : Setoid ℓU ℓu} {V : Setoid ℓV ℓv}
{A : SetoidFamily S ℓA ℓa} {B : SetoidFamily T ℓB ℓb} {C : SetoidFamily U ℓC ℓc} {D : SetoidFamily V ℓD ℓd}
(F : A ⇛ B) (G : B ⇛ C) (H : C ⇛ D) → (F ∘⇛ G) ∘⇛ H ≈≈ F ∘⇛ (G ∘⇛ H)
assocʳ F G H = ≈≈-sym (assocˡ F G H)

∘⇛-cong : {ℓS ℓs ℓT ℓt ℓU ℓu ℓA ℓa ℓB ℓb ℓC ℓc : Level}
{S : Setoid ℓS ℓs} {T : Setoid ℓT ℓt} {U : Setoid ℓU ℓu}
{A : SetoidFamily S ℓA ℓa} {B : SetoidFamily T ℓB ℓb} {C : SetoidFamily U ℓC ℓc}
  {F : A ⇛ B} {G : B ⇛ C} {H : A ⇛ B} {I : B ⇛ C}
  → F ≈≈ H → G ≈≈ I → F ∘⇛ G ≈≈ H ∘⇛ I
∘⇛-cong {U = U} {A} {B} {C} {F} {G} {H} {I} F≈H G≈I = record
  {ext =
    let open Setoid U renaming (trans to _⟨≈⟩_) in
    λ x → G=I.ext (map H ⟨$⟩ x) ⟨≈⟩ Π.cong (map G) (F=H.ext x)
  ; transport-ext-coh = λ x Bx →
    let V = index (map (F ∘⇛ G) ⟨$⟩ x) in let open Setoid V renaming (trans to _⟨≈⟩_) in
    trans-coh (G=I.ext (map H ⟨$⟩ x)) (Π.cong (map G) (F=H.ext x)) ⟨≈⟩
      (Π.cong (reindex (Π.cong (map G) (F=H.ext x))) (G=I.transport-ext-coh (map H ⟨$⟩ x) (transport H x ⟨$⟩ Bx)) ⟨≈⟩
      (sym (transport-coh G (F=H.ext x)) ⟨≈⟩
      Π.cong (transport G (map F ⟨$⟩ x)) (F=H.transport-ext-coh x Bx)))
  }
  where
    module F=H = _≈≈_ F≈H; module G=I = _≈≈_ G≈I
    open SetoidFamily C; open _⇛_
```

(SetoidFamily.trans-coh B' (G=H.ext x) (F=G.ext x) ⟨≈⟩ (Π.cong (reindex B' (F=G.ext x)) (G=H.transport-ext-coh x Bx))) ⟨≈⟩ (F=G.transport-ext-coh x Bx)

And now we are in a good position to show that ♯ is an equivalence relation.

```
♯-refl : {ℓS ℓs ℓA ℓa : Level} {S : Setoid ℓS ℓs}
{B : SetoidFamily S ℓA ℓa} → B ♯ B
♯-refl = record {to = id⇛; from = id⇛; left-inv = unitˡ id⇛; right-inv = unitʳ id⇛}

♯-sym : {ℓS ℓs ℓA ℓa ℓS' ℓs' ℓA' ℓa' : Level} {S : Setoid ℓS ℓs} {S' : Setoid ℓS' ℓs'}
{B : SetoidFamily S ℓA ℓa} {B' : SetoidFamily S' ℓA' ℓa'}
→ B ♯ B' → B' ♯ B
♯-sym B♯B' = record {to = eq.from; from = eq.to; left-inv = eq.right-inv; right-inv = eq.left-inv}
  where module eq = _♯_ B♯B'

♯-trans : {ℓS ℓs ℓA ℓa ℓT ℓt ℓB ℓb ℓU ℓu ℓC ℓc : Level}
{S : Setoid ℓS ℓs} {T : Setoid ℓT ℓt} {U : Setoid ℓU ℓu}
{A : SetoidFamily S ℓA ℓa} {B : SetoidFamily T ℓB ℓb} {C : SetoidFamily U ℓC ℓc}
→ A ♯ B → B ♯ C → A ♯ C
♯-trans A♯B B♯C = record
  {to = AB.to ∘⇛ BC.to
  ; from = BC.from ∘⇛ AB.from
  ; left-inv =
    assocˡ (BC.from ∘⇛ AB.from) AB.to BC.to ⟨≈≈⟩
    (∘⇛-cong (assocʳ BC.from AB.from AB.to ⟨≈≈⟩
      ∘⇛-cong (≈≈-refl BC.from) AB.left-inv ⟨≈≈⟩
      unitʳ BC.from) (≈≈-refl BC.to) ⟨≈≈⟩
    BC.left-inv)
  ; right-inv =
    assocˡ (AB.to ∘⇛ BC.to) BC.from AB.from ⟨≈≈⟩
    ∘⇛-cong (assocʳ AB.to BC.to BC.from ⟨≈≈⟩
      ∘⇛-cong (≈≈-refl _) BC.right-inv ⟨≈≈⟩
      unitʳ AB.to) (≈≈-refl AB.from) ⟨≈≈⟩
    AB.right-inv}
  where module AB = _♯_ A♯B; module BC = _♯_ B♯C
```

As with Setoid-reasoning, we introduce what looks like a seemingly unnecessary type is used to make it possible to infer arguments even if the underlying equality evaluates.

```
infixr 2 _♯⟨_⟩_ _♯˘⟨_⟩_
infix 4 _Is♯To_
infix 1 begin_
infix 3 _□
data _Is♯To_ {ℓS ℓs ℓA ℓa ℓS' ℓs' ℓA' ℓa' : Level} {S : Setoid ℓS ℓs} {S' : Setoid ℓS' ℓs'}
(From : SetoidFamily S ℓA ℓa) (To : SetoidFamily S' ℓA' ℓa')
: Set (ℓS ⊔ ℓA ⊔ ℓS' ⊔ ℓs ⊔ ℓa ⊔ ℓA' ⊔ ℓs' ⊔ ℓa') where
  relTo : (x♯y : From ♯ To) → From Is♯To To
begin_ : {ℓS ℓs ℓA ℓa ℓS' ℓs' ℓA' ℓa' : Level} {S : Setoid ℓS ℓs} {S' : Setoid ℓS' ℓs'}
{From : SetoidFamily S ℓA ℓa} {To : SetoidFamily S' ℓA' ℓa'} → From Is♯To To → From ♯ To
begin relTo x♯y = x♯y
_♯⟨_⟩_ : {ℓS ℓs ℓA ℓa ℓT ℓt ℓB ℓb ℓU ℓu ℓC ℓc : Level}
{S : Setoid ℓS ℓs} {T : Setoid ℓT ℓt} {U : Setoid ℓU ℓu}
(A : SetoidFamily S ℓA ℓa) {B : SetoidFamily T ℓB ℓb} {C : SetoidFamily U ℓC ℓc}
  → A ♯ B → B Is♯To C → A Is♯To C
A ♯⟨ A♯B ⟩ (relTo B♯C) = relTo (♯-trans A♯B B♯C)
_♯˘⟨_⟩_ : {ℓS ℓs ℓA ℓa ℓT ℓt ℓB ℓb ℓU ℓu ℓC ℓc : Level}
{S : Setoid ℓS ℓs} {T : Setoid ℓT ℓt} {U : Setoid ℓU ℓu}
(A : SetoidFamily S ℓA ℓa) {B : SetoidFamily T ℓB ℓb} {C : SetoidFamily U ℓC ℓc}
  → B ♯ A → B Is♯To C → A Is♯To C
A ♯˘⟨ B♯A ⟩ (relTo B♯C) = relTo (♯-trans (♯-sym B♯A) B♯C)
_□ : {ℓS ℓs ℓA ℓa : Level} {S : Setoid ℓS ℓs}
(B : SetoidFamily S ℓA ℓa) → B Is♯To B
B □ = relTo (♯-refl {B = B})
```

24 TypeEquiv

```
{-# OPTIONS --without-K #-}
module TypeEquiv where
open import Level using (Level; zero; suc)
open import DataProperties
open import Algebra using (CommutativeSemiring)
open import Algebra.Structures
  using (IsSemigroup; IsCommutativeMonoid; IsCommutativeSemiring)
open import Function renaming (_∘_ to _∘_)
open import Relation.Binary.PropositionalEquality using (refl)
open import Equiv
  using (_≐_; ≐-refl; _≃_; id≃; sym≃; ≃IsEquiv; qinv; _⊎≃_; _×≃_)
  --
  -- Type Equivalences
  -- for each type combinator, define two functions that are inverses, and
  -- establish an equivalence. These are all in the 'semantic space' with
  -- respect to Pi combinators.
  -- swap₊
swap₊ : ∀ {ℓ₁ ℓ₂} {A : Set ℓ₁} {B : Set ℓ₂} → A ⊎ B → B ⊎ A
swap₊ (inj₁ a) = inj₂ a
swap₊ (inj₂ b) = inj₁ b
```

```
abstract
  swapswap₊ : ∀ {ℓ₁ ℓ₂} {A : Set ℓ₁} {B : Set ℓ₂} → swap₊ ∘ swap₊ {A = A} {B} ≐ id
  swapswap₊ (inj₁ a) = refl
  swapswap₊ (inj₂ b) = refl

swap₊equiv : ∀ {ℓ₁ ℓ₂} {A : Set ℓ₁} {B : Set ℓ₂} → (A ⊎ B) ≃ (B ⊎ A)
swap₊equiv = (swap₊ , qinv swap₊ swapswap₊ swapswap₊)

  -- unite₊ and uniti₊

unite₊ : {ℓ′ ℓ : Level} {A : Set ℓ} → ⊥ {ℓ′} ⊎ A → A
unite₊ (inj₁ ())
unite₊ (inj₂ y) = y

uniti₊ : {ℓ′ ℓ : Level} {A : Set ℓ} → A → ⊥ {ℓ′} ⊎ A
uniti₊ a = inj₂ a

abstract
  uniti₊∘unite₊ : {ℓ ℓ′ : Level} {A : Set ℓ} → uniti₊ ∘ unite₊ ≐ id {A = ⊥ {ℓ′} ⊎ A}
  uniti₊∘unite₊ (inj₁ ())
  uniti₊∘unite₊ (inj₂ y) = refl

    -- this is so easy, Agda can figure it out by itself (see below)

  unite₊∘uniti₊ : {ℓ ℓ′ : Level} {A : Set ℓ} → unite₊ {ℓ′} ∘ uniti₊ ≐ id {A = A}
  unite₊∘uniti₊ _ = refl

unite₊equiv : {ℓ ℓ′ : Level} {A : Set ℓ} → (⊥ {ℓ′} ⊎ A) ≃ A
unite₊equiv {ℓ} {ℓ′} = (unite₊ , qinv uniti₊ (unite₊∘uniti₊ {ℓ} {ℓ′}) uniti₊∘unite₊)

uniti₊equiv : {ℓ ℓ′ : Level} {A : Set ℓ} → A ≃ (⊥ {ℓ′} ⊎ A)
uniti₊equiv = sym≃ unite₊equiv

  -- unite₊′ and uniti₊′

unite₊′ : {ℓ′ ℓ : Level} {A : Set ℓ} → A ⊎ ⊥ {ℓ′} → A
unite₊′ (inj₁ x) = x
unite₊′ (inj₂ ())

uniti₊′ : {ℓ′ ℓ : Level} {A : Set ℓ} → A → A ⊎ ⊥ {ℓ′}
uniti₊′ a = inj₁ a

abstract
  uniti₊′∘unite₊′ : ∀ {ℓ ℓ′} {A : Set ℓ} → uniti₊′ ∘ unite₊′ ≐ id {A = A ⊎ ⊥ {ℓ′}}
  uniti₊′∘unite₊′ (inj₁ _) = refl
  uniti₊′∘unite₊′ (inj₂ ())

    -- this is so easy, Agda can figure it out by itself (see below)

  unite₊′∘uniti₊′ : ∀ {ℓ ℓ′} {A : Set ℓ} → unite₊′ {ℓ′} ∘ uniti₊′ ≐ id {A = A}
  unite₊′∘uniti₊′ _ = refl

unite₊′equiv : ∀ {ℓ′ ℓ} {A : Set ℓ} → (A ⊎ ⊥ {ℓ′}) ≃ A
unite₊′equiv = (unite₊′ , qinv uniti₊′ ≐-refl uniti₊′∘unite₊′)

uniti₊′equiv : ∀ {ℓ′ ℓ} {A : Set ℓ} → A ≃ (A ⊎ ⊥ {ℓ′})
uniti₊′equiv = sym≃ unite₊′equiv

  -- unite* and uniti*

unite* : {ℓ′ ℓ : Level} {A : Set ℓ} → ⊤ {ℓ′} × A → A
unite* (tt , x) = x

uniti* : {ℓ′ ℓ : Level} {A : Set ℓ} → A → ⊤ {ℓ′} × A
uniti* x = tt , x

abstract
  uniti*∘unite* : ∀ {ℓ ℓ′} {A : Set ℓ} → uniti* ∘ unite* ≐ id {A = ⊤ {ℓ′} × A}
  uniti*∘unite* (tt , x) = refl

unite*equiv : ∀ {ℓ ℓ′} {A : Set ℓ} → (⊤ {ℓ′} × A) ≃ A
unite*equiv = unite* , qinv uniti* ≐-refl uniti*∘unite*
```

```
uniti*equiv : ∀ {ℓ ℓ′} {A : Set ℓ} → A ≃ (⊤ {ℓ′} × A)
uniti*equiv = sym≃ unite*equiv

  -- unite*′ and uniti*′

unite*′ : ∀ {ℓ ℓ′} {A : Set ℓ} → A × ⊤ {ℓ′} → A
unite*′ (x , tt) = x

uniti*′ : ∀ {ℓ ℓ′} {A : Set ℓ} → A → A × ⊤ {ℓ′}
uniti*′ x = x , tt

abstract
  uniti*′∘unite*′ : ∀ {ℓ ℓ′} {A : Set ℓ} → uniti*′ ∘ unite*′ ≐ id {A = A × ⊤ {ℓ′}}
  uniti*′∘unite*′ (x , tt) = refl

unite*′equiv : ∀ {ℓ ℓ′} {A : Set ℓ} → (A × ⊤ {ℓ′}) ≃ A
unite*′equiv = unite*′ , qinv uniti*′ ≐-refl uniti*′∘unite*′

uniti*′equiv : ∀ {ℓ ℓ′} {A : Set ℓ} → A ≃ (A × ⊤ {ℓ′})
uniti*′equiv = sym≃ unite*′equiv

  -- swap*

swap* : ∀ {ℓ ℓ′} {A : Set ℓ} {B : Set ℓ′} → A × B → B × A
swap* (a , b) = (b , a)

abstract
  swapswap* : {A B : Set} → swap* ∘ swap* ≐ id {A = A × B}
  swapswap* (a , b) = refl

swap*equiv : {A B : Set} → (A × B) ≃ (B × A)
swap*equiv = swap* , qinv swap* swapswap* swapswap*

  -- assocl₊ and assocr₊

assocl₊ : ∀ {ℓ₁ ℓ₂ ℓ₃} {A : Set ℓ₁} {B : Set ℓ₂} {C : Set ℓ₃} →
  (A ⊎ (B ⊎ C)) → ((A ⊎ B) ⊎ C)
assocl₊ (inj₁ a) = inj₁ (inj₁ a)
assocl₊ (inj₂ (inj₁ b)) = inj₁ (inj₂ b)
assocl₊ (inj₂ (inj₂ c)) = inj₂ c

assocr₊ : ∀ {ℓ₁ ℓ₂ ℓ₃} {A : Set ℓ₁} {B : Set ℓ₂} {C : Set ℓ₃} →
  ((A ⊎ B) ⊎ C) → (A ⊎ (B ⊎ C))
assocr₊ (inj₁ (inj₁ a)) = inj₁ a
assocr₊ (inj₁ (inj₂ b)) = inj₂ (inj₁ b)
assocr₊ (inj₂ c) = inj₂ (inj₂ c)

abstract
  assocl₊∘assocr₊ : ∀ {ℓ₁ ℓ₂ ℓ₃} {A : Set ℓ₁} {B : Set ℓ₂} {C : Set ℓ₃} →
    assocl₊ ∘ assocr₊ ≐ id {A = ((A ⊎ B) ⊎ C)}
  assocl₊∘assocr₊ (inj₁ (inj₁ a)) = refl
  assocl₊∘assocr₊ (inj₁ (inj₂ b)) = refl
  assocl₊∘assocr₊ (inj₂ c) = refl

  assocr₊∘assocl₊ : ∀ {ℓ₁ ℓ₂ ℓ₃} {A : Set ℓ₁} {B : Set ℓ₂} {C : Set ℓ₃} →
    assocr₊ ∘ assocl₊ ≐ id {A = (A ⊎ (B ⊎ C))}
  assocr₊∘assocl₊ (inj₁ a) = refl
  assocr₊∘assocl₊ (inj₂ (inj₁ b)) = refl
  assocr₊∘assocl₊ (inj₂ (inj₂ c)) = refl

assocr₊equiv : ∀ {ℓ₁ ℓ₂ ℓ₃} {A : Set ℓ₁} {B : Set ℓ₂} {C : Set ℓ₃} →
  ((A ⊎ B) ⊎ C) ≃ (A ⊎ (B ⊎ C))
assocr₊equiv =
  assocr₊ , qinv assocl₊ assocr₊∘assocl₊ assocl₊∘assocr₊

assocl₊equiv : ∀ {ℓ₁ ℓ₂ ℓ₃} {A : Set ℓ₁} {B : Set ℓ₂} {C : Set ℓ₃} →
  (A ⊎ (B ⊎ C)) ≃ ((A ⊎ B) ⊎ C)
assocl₊equiv = sym≃ assocr₊equiv
```

```
  -- assocl* and assocr*
assocl* : {A B C : Set} → (A × (B × C)) → ((A × B) × C)
assocl* (a , (b , c)) = ((a , b) , c)
assocr* : {A B C : Set} → ((A × B) × C) → (A × (B × C))
assocr* ((a , b) , c) = (a , (b , c))
abstract
  assocl*∘assocr* : {A B C : Set} → assocl* ∘ assocr* ≐ id {A = ((A × B) × C)}
  assocl*∘assocr* = ≐-refl
  assocr*∘assocl* : {A B C : Set} → assocr* ∘ assocl* ≐ id {A = (A × (B × C))}
  assocr*∘assocl* = ≐-refl
assocl*equiv : {A B C : Set} → (A × (B × C)) ≃ ((A × B) × C)
assocl*equiv =
  assocl* , qinv assocr* assocl*∘assocr* assocr*∘assocl*
assocr*equiv : {A B C : Set} → ((A × B) × C) ≃ (A × (B × C))
assocr*equiv = sym≃ assocl*equiv
  -- distz and factorz, on left
distz : ∀ {ℓ ℓ′} {A : Set ℓ} → (⊥ × A) → ⊥ {ℓ′}
distz = proj₁
factorz : ∀ {ℓ ℓ′} {A : Set ℓ} → ⊥ {ℓ′} → (⊥ {ℓ′} × A)
factorz ()
abstract
  distz∘factorz : ∀ {ℓ ℓ′} {A : Set ℓ} → distz ∘ factorz {ℓ} {ℓ′} {A} ≐ id
  distz∘factorz ()
  factorz∘distz : ∀ {ℓ ℓ′} {A : Set ℓ} → factorz {ℓ} {ℓ′} {A} ∘ distz ≐ id
  factorz∘distz (() , proj₂)
distzequiv : ∀ {ℓ ℓ′} {A : Set ℓ} → (⊥ × A) ≃ ⊥ {ℓ′}
distzequiv {A = A} =
  distz , qinv factorz (distz∘factorz {_} {_} {A}) factorz∘distz
factorzequiv : ∀ {ℓ ℓ′} {A : Set ℓ} → ⊥ {ℓ′} ≃ (⊥ × A)
factorzequiv {A = A} = sym≃ distzequiv
  -- distz and factorz, on right
distzr : {ℓ′ ℓ : Level} {A : Set ℓ} → (A × ⊥) → ⊥ {ℓ′}
distzr = proj₂
factorzr : {ℓ′ ℓ : Level} {A : Set ℓ} → ⊥ {ℓ′} → (A × ⊥ {ℓ′})
factorzr ()
abstract
  distzr∘factorzr : {ℓ ℓ′ : Level} {A : Set ℓ} → distzr ∘ factorzr {ℓ′} {ℓ} {A} ≐ id
  distzr∘factorzr ()
  factorzr∘distzr : {ℓ ℓ′ : Level} {A : Set ℓ} → factorzr {ℓ′} {ℓ} {A} ∘ distzr ≐ id
  factorzr∘distzr (_ , ())
distzrequiv : {ℓ ℓ′ : Level} {A : Set ℓ} → (A × ⊥) ≃ ⊥ {ℓ′}
distzrequiv {_} {_} {A} =
  distzr , qinv factorzr (distzr∘factorzr {_} {_} {A}) factorzr∘distzr
factorzrequiv : ∀ {ℓ ℓ′} {A : Set ℓ} → ⊥ {ℓ′} ≃ (A × ⊥)
factorzrequiv {A} = sym≃ distzrequiv
  -- dist and factor, on right
dist : {A B C : Set} → ((A ⊎ B) × C) → (A × C) ⊎ (B × C)
dist (inj₁ x , c) = inj₁ (x , c)
dist (inj₂ y , c) = inj₂ (y , c)
factor : {A B C : Set} → (A × C) ⊎ (B × C) → ((A ⊎ B) × C)
```

```
factor (inj₁ (a , c)) = inj₁ a , c
factor (inj₂ (b , c)) = inj₂ b , c
abstract
  dist∘factor : {A B C : Set} → dist {A} {B} {C} ∘ factor ≐ id
  dist∘factor (inj₁ x) = refl
  dist∘factor (inj₂ y) = refl
  factor∘dist : {A B C : Set} → factor {A} {B} {C} ∘ dist ≐ id
  factor∘dist (inj₁ x , c) = refl
  factor∘dist (inj₂ y , c) = refl
distequiv : {A B C : Set} → ((A ⊎ B) × C) ≃ ((A × C) ⊎ (B × C))
distequiv = dist , qinv factor dist∘factor factor∘dist
factorequiv : {A B C : Set} → ((A × C) ⊎ (B × C)) ≃ ((A ⊎ B) × C)
factorequiv = sym≃ distequiv
  -- dist and factor, on left
distl : {A B C : Set} → A × (B ⊎ C) → (A × B) ⊎ (A × C)
distl (x , inj₁ x₁) = inj₁ (x , x₁)
distl (x , inj₂ y) = inj₂ (x , y)
factorl : {A B C : Set} → (A × B) ⊎ (A × C) → A × (B ⊎ C)
factorl (inj₁ (x , y)) = x , inj₁ y
factorl (inj₂ (x , y)) = x , inj₂ y
abstract
  distl∘factorl : {A B C : Set} → distl {A} {B} {C} ∘ factorl ≐ id
  distl∘factorl (inj₁ (x , y)) = refl
  distl∘factorl (inj₂ (x , y)) = refl
  factorl∘distl : {A B C : Set} → factorl {A} {B} {C} ∘ distl ≐ id
  factorl∘distl (a , inj₁ x) = refl
  factorl∘distl (a , inj₂ y) = refl
distlequiv : {A B C : Set} → (A × (B ⊎ C)) ≃ ((A × B) ⊎ (A × C))
distlequiv = distl , qinv factorl distl∘factorl factorl∘distl
factorlequiv : {A B C : Set} → ((A × B) ⊎ (A × C)) ≃ (A × (B ⊎ C))
factorlequiv = sym≃ distlequiv
  --
  -- Commutative semiring structure
typesPlusIsSG : IsSemigroup {Level.suc Level.zero} {Level.zero} {Set} _≃_ _⊎_
typesPlusIsSG = record {
  isEquivalence = ≃IsEquiv;
  assoc = λ t₁ t₂ t₃ → assocr₊equiv {_} {_} {_} {t₁} {t₂} {t₃};
  •-cong = _⊎≃_
  }
typesTimesIsSG : IsSemigroup {Level.suc Level.zero} {Level.zero} {Set} _≃_ _×_
typesTimesIsSG = record {
  isEquivalence = ≃IsEquiv;
  assoc = λ t₁ t₂ t₃ → assocr*equiv {t₁} {t₂} {t₃};
  •-cong = _×≃_
  }
typesPlusIsCM : IsCommutativeMonoid _≃_ _⊎_ ⊥
typesPlusIsCM = record {
  isSemigroup = typesPlusIsSG;
  identityˡ = λ t → unite₊equiv {_} {_} {t};
  comm = λ t₁ t₂ → swap₊equiv {_} {_} {t₁} {t₂}
  }
typesTimesIsCM : IsCommutativeMonoid _≃_ _×_ ⊤
```

```
typesTimesIsCM = record {
  isSemigroup = typesTimesIsSG;
  identityˡ = λ t → unite*equiv {_} {_} {t};
  comm = λ t₁ t₂ → swap*equiv {t₁} {t₂}
  }
typesIsCSR : IsCommutativeSemiring _≃_ _⊎_ _×_ ⊥ ⊤
typesIsCSR = record {
  +-isCommutativeMonoid = typesPlusIsCM;
  *-isCommutativeMonoid = typesTimesIsCM;
  distribʳ = λ t₁ t₂ t₃ → distequiv {t₂} {t₃} {t₁};
  zeroˡ = λ t → distzequiv {_} {_} {t}
  }
typesCSR : CommutativeSemiring (Level.suc Level.zero) Level.zero
typesCSR = record {
  Carrier = Set;
  _≈_ = _≃_;
  _+_ = _⊎_;
  _*_ = _×_;
  0# = ⊥;
  1# = ⊤;
  isCommutativeSemiring = typesIsCSR
  }
```

Part VII

Misc

25 Function2

```
module Function2 where
  -- should be defined in Function ?
infix 4 _$ᵢ
_$ᵢ : ∀ {a b} {A : Set a} {B : A → Set b} →
  ((x : A) → B x) → {x : A} → B x
_$ᵢ f {x} = f x
```

26 Parallel Composition

We need a way to put two SetoidFamily "side by side" – a form of parellel composition. To achieve this requires a certain amount of infrastructure: parallel composition of relations, and both disjoint sum and cartesian product of Setoids. So the next couple of sections proceed with that infrastruction, before diving in to the crux of the matter.

```
module ParComp where
open import Level
open import Relation.Binary   using (Setoid; REL; Rel)
open import Function          using (flip) renaming (id to id₀; _∘_ to _⊚_)
open import Function.Equality using (Π; _⟨$⟩_; cong; id; _⟶_; _∘_)
```

```
open import Function.Inverse   using () renaming (_InverseOf_ to Inv)
open import Relation.Binary.Product.Pointwise using (_×-setoid_)
open import Categories.Category using (Category)
open import Categories.Object.Coproduct
open import DataProperties
open import SetoidEquiv
open import TypeEquiv using (swap₊; swap⋆)
```

26.1 Parallel Composition of relations

Parallel composition of heterogeneous relations.

Note that this is a specialized version of the standard library's _⊎-Rel_ (see Relation.Binary.Sum); this one gets rid of the bothersome $_1\sim_2$ term.

```
data _∥_ {a₁ b₁ c₁ a₂ b₂ c₂ : Level}
  {A₁ : Set a₁} {B₁ : Set b₁} {A₂ : Set a₂} {B₂ : Set b₂}
  (R₁ : REL A₁ B₁ c₁) (R₂ : REL A₂ B₂ c₂)
  : REL (A₁ ⊎ A₂) (B₁ ⊎ B₂) (c₁ ⊔ c₂) where
  left : {x : A₁} {y : B₁} (r₁ : R₁ x y) → (R₁ ∥ R₂) (inj₁ x) (inj₁ y)
  right : {x : A₂} {y : B₂} (r₂ : R₂ x y) → (R₁ ∥ R₂) (inj₂ x) (inj₂ y)
elim : {a₁ b₁ a₂ b₂ c₁ c₂ d : Level}
  {A₁ : Set a₁} {B₁ : Set b₁} {A₂ : Set a₂} {B₂ : Set b₂}
  {R₁ : REL A₁ B₁ c₁} {R₂ : REL A₂ B₂ c₂}
  (P : {a : A₁ ⊎ A₂} {b : B₁ ⊎ B₂} (pf : (R₁ ∥ R₂) a b) → Set d)
  (F : {a : A₁} {b : B₁} → (f : R₁ a b) → P (left f))
  (G : {a : A₂} {b : B₂} → (g : R₂ a b) → P (right g))
  {a : A₁ ⊎ A₂} {b : B₁ ⊎ B₂} → (x : (R₁ ∥ R₂) a b) → P x
elim P F G (left r₁) = F r₁
elim P F G (right r₂) = G r₂
  -- If the argument relations are symmetric then so is their parallel composition.
∥-sym : {a a′ c c′ : Level} {A : Set a} {R₁ : Rel A c}
  {A′ : Set a′} {R₂ : Rel A′ c′}
  (sym₁ : {x y : A} → R₁ x y → R₁ y x) (sym₂ : {x y : A′} → R₂ x y → R₂ y x)
  {x y : A ⊎ A′}
  → (R₁ ∥ R₂) x y → (R₁ ∥ R₂) y x
∥-sym {R₁ = R₁} {R₂ = R₂} sym₁ sym₂ pf =
  elim (λ {a b} (x : (R₁ ∥ R₂) a b) → (R₁ ∥ R₂) b a) (left ⊚ sym₁) (right ⊚ sym₂) pf
∥-trans : {a a′ ℓ ℓ′ : Level} (A : Setoid a ℓ) (A′ : Setoid a′ ℓ′)
  {x y z : Setoid.Carrier A ⊎ Setoid.Carrier A′} →
  let R₁ = Setoid._≈_ A in let R₂ = Setoid._≈_ A′ in
  (R₁ ∥ R₂) x y → (R₁ ∥ R₂) y z → (R₁ ∥ R₂) x z
∥-trans A A′ {inj₁ x} (left r₁) (left r₂) = left (Setoid.trans A r₁ r₂)
∥-trans A A′ {inj₂ y₂} (right r₂) (right r₃) = right (Setoid.trans A′ r₂ r₃)
```

26.2 Union and product of Setoid

```
module _ {ℓA₁ ℓa₁ ℓA₂ ℓa₂ : Level} (S₁ : Setoid ℓA₁ ℓa₁) (S₂ : Setoid ℓA₂ ℓa₂) where
  infix 3 _⊎S_ _×S_
  open Setoid S₁ renaming (Carrier to s₁; _≈_ to _≈₁_; refl to refl₁; sym to sym₁)
  open Setoid S₂ renaming (Carrier to s₂; _≈_ to _≈₂_; refl to refl₂; sym to sym₂)
  _⊎S_ : Setoid (ℓA₁ ⊔ ℓA₂) (ℓa₁ ⊔ ℓa₂)
```

```
_⊎S_ = record
  {Carrier = s₁ ⊎ s₂
  ; _≈_ = _≈₁_ ∥ _≈₂_
  ; isEquivalence = record
    {refl = λ {{inj₁ x} → left refl₁; {inj₂ y} → right refl₂}
    ; sym = ∥-sym sym₁ sym₂
    ; trans = ∥-trans S₁ S₂
    }
  }
_×S_ : Setoid (ℓA₁ ⊔ ℓA₂) (ℓa₁ ⊔ ℓa₂)
_×S_ = S₁ ×-setoid S₂
```

26.3 Union of Setoid is commutative

```
module _ {ℓA₁ ℓa₁ ℓA₂ ℓa₂ : Level} (S₁ : Setoid ℓA₁ ℓa₁) (S₂ : Setoid ℓA₂ ℓa₂) where
  ⊎S-comm : (S₁ ⊎S S₂) ≅ (S₂ ⊎S S₁)
  ⊎S-comm = record
    {to = record {_⟨$⟩_ = swap₊; cong = λ {(left r₁) → right r₁; (right r₂) → left r₂}}
    ; from = record {_⟨$⟩_ = swap₊; cong = λ {(left r₁) → right r₁; (right r₂) → left r₂}}
    ; inverse-of = record
      {left-inverse-of = λ {(inj₁ x) → left (refl S₁); (inj₂ y) → right (refl S₂)}
      ; right-inverse-of = λ {(inj₁ x) → left (refl S₂); (inj₂ y) → right (refl S₁)}}
    }
    where open Setoid
```

26.4 _⊎S_ is a congruence

```
module _ {ℓA₁ ℓa₁ ℓA₂ ℓa₂ ℓA₃ ℓa₃ ℓA₄ ℓa₄ : Level}
  {S₁ : Setoid ℓA₁ ℓa₁} {S₂ : Setoid ℓA₂ ℓa₂} {S₃ : Setoid ℓA₃ ℓa₃} {S₄ : Setoid ℓA₄ ℓa₄} where
  _⊎S₁_ : S₁ ≅ S₃ → S₂ ≅ S₄ → (S₁ ⊎S S₂) ≅ (S₃ ⊎S S₄)
  1≅3 ⊎S₁ 2≅4 = record
    {to = record
      {_⟨$⟩_ = λ {(inj₁ x) → inj₁ (to 1≅3 ⟨$⟩ x); (inj₂ y) → inj₂ (to 2≅4 ⟨$⟩ y)}
      ; cong = λ {(left r₁) → left (cong (to 1≅3) r₁); (right r₂) → right (cong (to 2≅4) r₂)}}
    ; from = record
      {_⟨$⟩_ = λ {(inj₁ x) → inj₁ (from 1≅3 ⟨$⟩ x); (inj₂ y) → inj₂ (from 2≅4 ⟨$⟩ y)}
      ; cong = λ {(left r₁) → left (cong (from 1≅3) r₁); (right r₂) → right (cong (from 2≅4) r₂)}}
    ; inverse-of = record
      {left-inverse-of = λ {(inj₁ x) → left (left-inverse-of 1≅3 x)
        ; (inj₂ y) → right (left-inverse-of 2≅4 y)}
      ; right-inverse-of = λ {(inj₁ x) → left (right-inverse-of 1≅3 x)
        ; (inj₂ y) → right (right-inverse-of 2≅4 y)}}
    }
    where open _≅_
```

27 Belongs

Rather than over-generalize to a type of locations for an arbitrary predicate, stick to simply working with locations, and making them into a type.

```
module Belongs where
open import Level renaming (zero to lzero; suc to lsuc) hiding (lift)
open import Relation.Binary using (Setoid; IsEquivalence; Rel;
  Reflexive; Symmetric; Transitive)
open import Function.Equality using (Π; _⟶_; id; _∘_; _⟨$⟩_; cong)
open import Function          using (_$_; flip) renaming (id to id₀; _∘_ to _⊚_)
open import Data.List using (List; []; _++_; _::_; map; reverse)
open import Data.Nat using (ℕ; zero; suc)
open import EqualityCombinators
open import DataProperties
open import SetoidEquiv
open import ParComp
open import Structures.CommMonoid
open import TypeEquiv using (swap₊)
```

The goal of this section is to capture a notion that we have an element x belonging to a list xs. We want to know *which* x ∈ xs is the witness, as there could be many x's in xs. Furthermore, we are in the Setoid setting, thus we do not care about x itself, any y such that x ≈ y will do, as long as it is in the "right" location.

And then we want to capture the idea of when two such are equivalent – when is it that Belongs xs is just as good as Belongs ys?

For the purposes of CommMonoid, all we need is some notion of Bag Equivalence. We will aim for that, without generalizing too much.

27.1 Location

Setoid-based variant of Any, but without the extra property. Nevertheless, much inspiration came from reading Data.List.Any and Data.List.Any.Properties.

First, a notion of Location in a list, but suited for our purposes.

```
module Locations {ℓS ℓs : Level} (S : Setoid ℓS ℓs) where
  open Setoid S
  infix 4 _∈₀_
  data _∈₀_ : Carrier → List Carrier → Set (ℓS ⊔ ℓs) where
    here  : {x a : Carrier} {xs : List Carrier} (sm : a ≈ x)      → a ∈₀ (x :: xs)
    there : {x a : Carrier} {xs : List Carrier} (pxs : a ∈₀ xs) → a ∈₀ (x :: xs)
  open _∈₀_ public
```

One instinct is go go with natural numbers directly; while this has the "right" computational content, that is harder for deduction. Nevertheless, the 'location' function is straightforward:

```
toℕ : {x : Carrier} {xs : List Carrier} → x ∈₀ xs → ℕ
toℕ (here _)  = 0
toℕ (there pf) = suc (toℕ pf)
```

We need to know when two locations are the same.

```
module LocEquiv {ℓS ℓs} (S : Setoid ℓS ℓs) where
  open Setoid            S
  open Locations         S
  open SetoidCombinators S
  infix 3 _≋_
```

```
data _≋_ : {y y' : Carrier} {ys : List Carrier} (loc : y ∈₀ ys) (loc' : y' ∈₀ ys) → Set (ℓS ⊔ ℓs) where
  hereEq : {xs : List Carrier} {x y z : Carrier} (x≈z : x ≈ z) (y≈z : y ≈ z)
    → here {x = z} {x} {xs} x≈z ≋ here {x = z} {y} {xs} y≈z
  thereEq : {xs : List Carrier} {x x' z : Carrier} {loc : x ∈₀ xs} {loc' : x' ∈₀ xs}
    → loc ≋ loc' → there {x = z} loc ≋ there {x = z} loc'
open _≋_ public
```

These are seen to be another form of natural numbers as well.

It is on purpose that _≋_ preserves positions. Suppose that we take the setoid of the Latin alphabet, with _≈_ identifying upper and lower case. There should be 3 elements of _≋_ for a :: A :: a :: [], not 6. When we get to defining BagEq, there will be 6 different ways in which that list, as a Bag, is equivalent to itself.

Furthermore, it is important to notice that we have an injectivity property: $\mathsf{x} \in_0 \mathsf{xs} \approx \mathsf{y} \in_0 \mathsf{xs}$ implies $\mathsf{x} \approx \mathsf{y}$.

```
≋→≈ : {x y : Carrier} {xs : List Carrier} (x∈xs : x ∈₀ xs) (y∈xs : y ∈₀ xs)
  → x∈xs ≋ y∈xs → x ≈ y
≋→≈ (here x≈z) ∘ (here y≈z) (hereEq .x≈z y≈z)                  = x≈z ⟨≈≈˘⟩ y≈z
≋→≈ (there x∈xs) ∘ (there _) (thereEq {loc' = loc'} x∈xs≋loc') = ≋→≈ x∈xs loc' x∈xs≋loc'
```

27.2 Substitution

Given $\mathsf{x} \approx \mathsf{y}$, we have a substitution-like operator that maps from $\mathsf{x} \in_0 \mathsf{xs}$ to $\mathsf{y} \in_0 \mathsf{xs}$. Here, choose the HoTT-inspired name, ap-$\in_0$. We will see later that these are the essential ingredients for showing that ♯ (at $\in_0$) is reflexive.

```
module Substitution {ℓS ℓs : Level} (S : Setoid ℓS ℓs) where
  open Setoid S
  open Locations S
  open LocEquiv S
  open SetoidCombinators S
  ap-∈₀ : {x y : Carrier} {xs : List Carrier} → x ≈ y → x ∈₀ xs → y ∈₀ xs
  ap-∈₀ x≈y (here a≈x) = here (x≈y ⟨≈˘≈⟩ a≈x)
  ap-∈₀ x≈y (there x∈xs) = there (ap-∈₀ x≈y x∈xs)
  ap-∈₀-eq : {x y : Carrier} {xs : List Carrier} → (p : x ≈ y) → (x∈xs : x ∈₀ xs) → x∈xs ≋ ap-∈₀ p x∈xs
  ap-∈₀-eq p (here sm) = hereEq sm (p ⟨≈˘≈⟩ sm)
  ap-∈₀-eq p (there x∈xs) = thereEq (ap-∈₀-eq p x∈xs)
  ap-∈₀-refl : {x : Carrier} {xs : List Carrier} → (x∈xs : x ∈₀ xs) → ap-∈₀ refl x∈xs ≋ x∈xs
  ap-∈₀-refl (Locations.here sm) = hereEq (refl ⟨≈˘≈⟩ sm) sm
  ap-∈₀-refl (Locations.there xx) = thereEq (ap-∈₀-refl xx)
  ap-∈₀-cong : {x y : Carrier} {xs : List Carrier} (x≈y : x ≈ y)
    {i j : x ∈₀ xs} → i ≋ j → ap-∈₀ x≈y i ≋ ap-∈₀ x≈y j
  ap-∈₀-cong x≈y (hereEq x≈z y≈z) = hereEq (x≈y ⟨≈˘≈⟩ x≈z) (x≈y ⟨≈˘≈⟩ y≈z)
  ap-∈₀-cong x≈y (thereEq i≋j) = thereEq (ap-∈₀-cong x≈y i≋j)
  ap-∈₀-linv : {x y : Carrier} {xs : List Carrier} (x≈y : x ≈ y)
    (x∈xs : x ∈₀ xs) → ap-∈₀ (sym x≈y) (ap-∈₀ x≈y x∈xs) ≋ x∈xs
  ap-∈₀-linv x≈y (here sm) = hereEq ((sym x≈y) ⟨≈˘≈⟩ (x≈y ⟨≈˘≈⟩ sm)) sm
  ap-∈₀-linv x≈y (there x∈xs) = thereEq (ap-∈₀-linv x≈y x∈xs)
  ap-∈₀-rinv : {x y : Carrier} {ys : List Carrier} (x≈y : x ≈ y)
    (y∈ys : y ∈₀ ys) → ap-∈₀ x≈y (ap-∈₀ (sym x≈y) y∈ys) ≋ y∈ys
  ap-∈₀-rinv x≈y (here sm) = hereEq (x≈y ⟨≈˘≈⟩ (sym x≈y ⟨≈˘≈⟩ sm)) sm
  ap-∈₀-rinv x≈y (there y∈ys) = thereEq (ap-∈₀-rinv x≈y y∈ys)
  ap-∈₀-trans : {x y z : Carrier} {xs : List Carrier} {x∈xs : x ∈₀ xs}
    (x≈y : x ≈ y) (y≈z : y ≈ z) → ap-∈₀ (trans x≈y y≈z) x∈xs ≋ ap-∈₀ y≈z (ap-∈₀ x≈y x∈xs)
```

```
ap-∈₀-trans {x∈xs = here sm} x≈y y≈z = hereEq (trans x≈y y≈z ⟨≈˘≈⟩ sm) (y≈z ⟨≈˘≈⟩ (x≈y ⟨≈˘≈⟩ sm))
ap-∈₀-trans {x∈xs = there x∈xs} x≈y y≈z = thereEq (ap-∈₀-trans x≈y y≈z)
```

27.3 Membership module

We now have all the ingredients to show that locations (_∈₀_) form a Setoid.

```
module Membership {ℓS ℓs} (S : Setoid ℓS ℓs) where
  open Setoid S
  open Locations S
  open LocEquiv S
  open Substitution S
  ≋-refl : {x : Carrier} {xs : List Carrier} {p : x ∈₀ xs} → p ≋ p
  ≋-refl {p = here a≈x} = hereEq a≈x a≈x
  ≋-refl {p = there p}   = thereEq ≋-refl
  ≋-sym : {l : List Carrier} {x y : Carrier} {x∈l : x ∈₀ l} {y∈l : y ∈₀ l} → x∈l ≋ y∈l → y∈l ≋ x∈l
  ≋-sym (hereEq x≈z y≈z) = hereEq _ _
  ≋-sym (thereEq pf) = thereEq (≋-sym pf)
  ≋-trans : {l : List Carrier} {x y z : Carrier} {x∈l : x ∈₀ l} {y∈l : y ∈₀ l} {z∈l : z ∈₀ l}
    → x∈l ≋ y∈l → y∈l ≋ z∈l → x∈l ≋ z∈l
  ≋-trans (hereEq x≈z y≈z) (hereEq .y≈z y≈z₁) = hereEq x≈z y≈z₁
  ≋-trans (thereEq pp) (thereEq qq) = thereEq (≋-trans pp qq)
  ≡→≋ : {x : Carrier} {xs : List Carrier} {p q : x ∈₀ xs} → p ≡ q → p ≋ q
  ≡→≋ ≡.refl = ≋-refl
```

The type elements l is just ∃ Carrier (λ witness → witness ∈₀ l), but it is more convenient to have a dedicated name (and notation). For now, no dedicated name will be given to the equality.

```
record elements (l : List Carrier) : Set (ℓS ⊔ ℓs) where
  constructor El
  field
    {witness} : Carrier
    belongs : witness ∈₀ l
open elements public
lift-el : {l₁ l₂ : List Carrier} (f : ∀ {w} → (w ∈₀ l₁ → w ∈₀ l₂))
  → elements l₁ → elements l₂
lift-el f (El l) = El (f l)
_⟷_ : {l : List Carrier} → Rel (elements l) (ℓs ⊔ ℓS)
(El b₁) ⟷ (El b₂) = b₁ ≋ b₂
elem-of : List Carrier → Setoid (ℓs ⊔ ℓS) (ℓs ⊔ ℓS)
elem-of l = record
  {Carrier = elements l
  ; _≈_ = _⟷_
  ; isEquivalence = record {refl = ≋-refl; sym = ≋-sym; trans = ≋-trans}
  }
```

27.4 BagEq

Fundamental definition: two Bags, represented as List Carrier are equivalent if and only if there exists a permutation between their Setoid of positions, and this is independent of the representative. The best way to succinctly express this is via _⇔_.

It is very important to note that _⇔_ isn't reflective 'for free', i.e. the proof does not involve just id.

```
module BagEq {ℓS ℓs} (S : Setoid ℓS ℓs) where
  open Setoid S
  open Locations S
  open LocEquiv S
  open Membership S
  open Substitution S

  infix 3 _⇔_

  _⇔_ : (xs ys : List Carrier) → Set (ℓS ⊔ ℓs)
  xs ⇔ ys = elem-of xs ≅ elem-of ys

  ≡→⇔ : {a b : List Carrier} → a ≡ b → a ⇔ b
  ≡→⇔ ≡.refl = ≅-refl
```

27.5 ++≉⊎S : ⋯ → (elem-of xs ⊎S elem-of ys) ≅ elem-of (xs + ys)

```
module ConcatTo⊎S {ℓS ℓs : Level} (S : Setoid ℓS ℓs) where
  open Setoid S renaming (Carrier to A)
  open SetoidCombinators S
  open LocEquiv S
  open Locations S
  open Membership S
  open Substitution S

  ⊎S≅++ : {xs ys : List A} → (elem-of xs ⊎S elem-of ys) ≅ (elem-of (xs + ys))
  ⊎S≅++ {xs} {ys} = record
    {to = record {_⟨$⟩_ = ⊎→++; cong = ⊎→++-cong}
    ; from = record {_⟨$⟩_ = ++→⊎ xs; cong = ++→⊎-cong xs}
    ; inverse-of = record
      {left-inverse-of = lefty
      ; right-inverse-of = righty {xs}}
    }
    where
      ⊎ˡ : ∀ {zs ws} {a : A} → a ∈₀ zs → a ∈₀ zs + ws
      ⊎ˡ (here sm) = here sm
      ⊎ˡ (there pf) = there (⊎ˡ pf)

      ⊎ʳ : (zs : List A) {ws : List A} {a : A} → a ∈₀ ws → a ∈₀ zs + ws
      ⊎ʳ []       p = p
      ⊎ʳ (x :: l) p = there (⊎ʳ l p)

      ⊎→++ : ∀ {zs ws} → elements zs ⊎ elements ws → elements (zs + ws)
      ⊎→++ (inj₁ (El w∈zs)) = El (⊎ˡ w∈zs)
      ⊎→++ {zs} (inj₂ (El w∈ws)) = El (⊎ʳ zs w∈ws)

      ⊎ˡ-cong : {zs ws : List A} {x y : elements zs} → x ⟷ y
        → ⊎ˡ {zs} {ws} (belongs x) ≋ ⊎ˡ (belongs y)
      ⊎ˡ-cong (hereEq x≈z y≈z) = hereEq x≈z y≈z
      ⊎ˡ-cong (thereEq x≋y) = thereEq (⊎ˡ-cong x≋y)

      ⊎ʳ-cong : (zs : List A) {ws : List A} {x y : elements ws} → x ⟷ y
        → ⊎ʳ zs (belongs x) ≋ ⊎ʳ zs (belongs y)
      ⊎ʳ-cong [] pf = pf
      ⊎ʳ-cong (x :: l) pf = thereEq (⊎ʳ-cong l pf)

      ⊎→++-cong : {zs ws : List A} {i j : elements zs ⊎ elements ws}
        → ((λ w₁ w₂ → belongs w₁ ≋ belongs w₂) ∥ (λ w₁ w₂ → belongs w₁ ≋ belongs w₂)) i j
        → belongs (⊎→++ i) ≋ belongs (⊎→++ j)
```

```
⊎→++-cong (left    ∈x≋∈y)     = ⊎ˡ-cong ∈x≋∈y
⊎→++-cong {zs} (right ∈x≋∈y) = ⊎ʳ-cong zs ∈x≋∈y

∽∥∽-cong : {xs ys us vs : List A}
  (F : elements xs → elements us)
  (F-cong : {p : elements xs} {q : elements xs} → p ⟷ q → F p ⟷ F q)
  (G : elements ys → elements vs)
  (G-cong : {p : elements ys} {q : elements ys} → p ⟷ q → G p ⟷ G q)
  → {pf : elements xs ⊎ elements ys} {pf' : elements xs ⊎ elements ys}
  → (_⟷_ ∥ _⟷_) pf pf' → (_⟷_ ∥ _⟷_) ((F ⊎₁ G) pf) ((F ⊎₁ G) pf')
∽∥∽-cong F F-cong G G-cong (left x~₁y) = left (F-cong x~₁y)
∽∥∽-cong F F-cong G G-cong (right x~₂y) = right (G-cong x~₂y)

++→⊎ : ∀ xs {ys} → elements (xs + ys) → elements xs ⊎ elements ys
++→⊎ [] p = inj₂ p
++→⊎ (x :: l) (El (here p)) = inj₁ (El (here p))
++→⊎ (x :: l) (El (there p)) = (lift-el there ⊎₁ id₀) (++→⊎ l (El p))

++→⊎-cong : (zs : List A) {ws : List A}
  {i j : elements (zs + ws)} → i ⟷ j
  → (_⟷_ ∥ _⟷_) (++→⊎ zs i) (++→⊎ zs j)
++→⊎-cong [] i≋j = right i≋j
++→⊎-cong (_ :: xs) (hereEq _ _) = left (hereEq _ _)
++→⊎-cong (_ :: xs) (thereEq pf) = ∽∥∽-cong (lift-el there) thereEq id₀ id₀ (++→⊎-cong xs pf)

righty : {xs ys : List A} (x : elements (xs + ys)) → ⊎→++ (++→⊎ xs x) ⟷ x
righty {[]} x = ≋-refl
righty {x :: xs₁} (El (here sm)) = hereEq sm sm
righty {_ :: xs₁} (El (there x)) with ++→⊎ xs₁ (El x) | righty {xs₁} (El x)
... | inj₁ x₁∈xs₁ | ans = thereEq ans
... | inj₂ x₁∈ys | ans = thereEq ans

lefty : {xs ys : List A} (x : elements xs ⊎ elements ys) →
  (_⟷_ ∥ _⟷_) (++→⊎ xs (⊎→++ x)) x
lefty {[]} (inj₁ (El ()))
lefty {[]} (inj₂ y) = right ≋-refl
lefty {_ :: _} (inj₁ (El (here sm))) = left (hereEq sm sm)
lefty {_ :: xs₁} {ys} (inj₁ (El (there x))) with ++→⊎ xs₁ {ys} (El (⊎ˡ x))
  | lefty {ys = ys} (inj₁ (El x))
... | inj₁ res | left ans = left (thereEq ans)
... | inj₂ res | ()
lefty {x₂ :: xs₂} {ys} (inj₂ (El (here sm))) with ++→⊎ xs₂ (El (⊎ʳ xs₂ {ys} (here sm)))
  | lefty {xs₂} {ys} (inj₂ (El (here sm)))
... | inj₁ res | ()
... | inj₂ res | right ans = right ans
lefty {x₂ :: xs₂} {ys} (inj₂ (El (there x))) with ++→⊎ xs₂ (El (⊎ʳ xs₂ {ys} (there x)))
  | lefty {xs₂} {ys} (inj₂ (El (there x)))
... | inj₁ res | ()
... | inj₂ res | right ans = right ans
```

27.6 Bottom as a Setoid

```
⊥⊥ : ∀ {ℓl ℓi} → Setoid ℓl ℓi
⊥⊥ = record
  {Carrier = ⊥
  ; _≈_ = λ _ _ → ⊤
  ; isEquivalence = record {refl = tt; sym = λ _ → tt; trans = λ _ _ → tt}
  }
```

```
module ElemOf[] {ℓS ℓs : Level} (S : Setoid ℓS ℓs) where
  open Membership S

  elem-of-[] : Setoid.Carrier (elem-of []) → ⊥ {ℓS}
  elem-of-[] (El ())

  ⊥⊥≅elem-of-[] : ⊥⊥ {ℓS} {ℓs} ≅ (elem-of [])
  ⊥⊥≅elem-of-[] = record
    {to = record {_⟨$⟩_ = λ {()}; cong = λ {{()}}}
    ; from = record {_⟨$⟩_ = elem-of-[]; cong = λ {{El ()}}}
    ; inverse-of = record {left-inverse-of = λ {()}; right-inverse-of = λ {(El ())}}}
```

27.7 elem-of map properties

```
module ElemOfMap {ℓS ℓs : Level} {S T : Setoid ℓS ℓs} where
  open Setoid hiding (_≈_)
  open BagEq S
  open Membership T using (lift-el; elem-of; ≋-refl; ≋-sym; ≋-trans)
  open Membership S using (El; belongs; elements; _←→_) renaming (elem-of to elem-ofₛ)
  open _≅_
  open LocEquiv T using (_≋_)
  open LocEquiv S renaming (_≋_ to _≋ₛ_)
  open Locations T using (_∈₀_)
  open Locations S renaming (here to hereₛ; there to thereₛ) hiding (_∈₀_)

  copy-func : {l : List (Carrier S)} (F : S ⟶ T) → (e : elements l) → (F ⟨$⟩ Membership.witness e ∈₀ map (_⟨$⟩_ F) l)
  copy-func F (El (hereₛ sm)) = Locations.here (cong F sm)
  copy-func F (El (thereₛ belongs₁)) = Locations.there (copy-func F (El belongs₁))

  record shifted-elements (F : S ⟶ T) (l : List (Carrier S)) : Set (ℓS ⊔ ℓs) where
    constructor SE
    open Setoid T using (_≈_)
    field
      elem : Membership.elements S l
      {shift-witness} : Carrier T
      sw-good : shift-witness ≈ F ⟨$⟩ Membership.witness elem

  open shifted-elements

  copy-func-cong : {l : List (Carrier S)} (F : S ⟶ T) {i j : shifted-elements F l}
    → Membership.belongs (elem i) ≋ₛ Membership.belongs (elem j)
    → copy-func F (elem i) ≋ copy-func F (elem j)
  copy-func-cong F (LocEquiv.hereEq x≈z y≈z) = LocEquiv.hereEq (cong F x≈z) (cong F y≈z)
  copy-func-cong {_ :: _} F {SE (El (Locations.there el₁)) g₁} {SE (El (Locations.there el₂)) g₂}
    (LocEquiv.thereEq eq) = LocEquiv.thereEq (copy-func-cong F {SE (El el₁) g₁} {SE (El el₂) g₂} eq)

  copy-unfunc : {l : List (Carrier S)} (F : S ⟶ T) {w : Carrier T} → w ∈₀ map (_⟨$⟩_ F) l → shifted-elements F l
  copy-unfunc {[]} F {w} ()
  copy-unfunc {x :: l} F {w} (Locations.here sm) = record
    {elem = Membership.El {witness = x} (Locations.here (refl S))
    ; sw-good = sm}
  copy-unfunc {x :: l} F {w} (Locations.there kk) =
    let se = copy-unfunc {l} F {w} kk in
    record {elem = Membership.El (Locations.there (belongs (elem se))); sw-good = sw-good se}

  copy-unfunc-cong : {l : List (Carrier S)} (F : S ⟶ T) {w₁ w₂ : Carrier T}
    → {b₁ : w₁ ∈₀ map (_⟨$⟩_ F) l} → {b₂ : w₂ ∈₀ map (_⟨$⟩_ F) l} → b₁ ≋ b₂
    → belongs (elem (copy-unfunc F b₁)) ≋ₛ belongs (elem (copy-unfunc F b₂))
  copy-unfunc-cong {[]} F ()
  copy-unfunc-cong {x :: l} F (LocEquiv.hereEq x≈z y≈z) = LocEquiv.hereEq (refl S) (refl S)
  copy-unfunc-cong {x :: l} F (LocEquiv.thereEq b₁≋b₂) = LocEquiv.thereEq (copy-unfunc-cong {l} F b₁≋b₂)
```

```
left-inv : {l : List (Carrier S)} {F : S ⟶ T} (x : Membership.elements T (map (_⟨$⟩_ F) l)) →
  copy-func F (elem (copy-unfunc F (Membership.belongs x))) ≋ Membership.belongs x
left-inv {[]} (Membership.El ())
left-inv {x :: l} {F} (Membership.El (Locations.here sm)) = LocEquiv.hereEq (cong F (refl S)) sm
left-inv {x :: l} {F} (Membership.El (Locations.there belongs₁)) = LocEquiv.thereEq (left-inv (Membership.El belongs₁))
right-inv : {l : List (Carrier S)} {F : S ⟶ T} (se : shifted-elements F l)
  → Membership.belongs (elem (copy-unfunc F (copy-func F (elem se)))) ≋ₛ Membership.belongs (elem se)
right-inv {[]} {F} (SE (Membership.El ()) _)
right-inv {x :: l} {F} (SE (Membership.El (Locations.here sm)) sw-good₁) = LocEquiv.hereEq (refl S) sm
right-inv {x :: l} {F} (SE (Membership.El (Locations.there belongs₁)) sw-good₁) = thereEq (right-inv (SE (El belongs₁) sw-good₁))
shifted : (S ⟶ T) → List (Carrier S) → Setoid (ℓS ⊔ ℓs) _
shifted F l = record
  {Carrier = shifted-elements F l
  ; _≈_ = λ a b → elem a ⟷ elem b
  ; isEquivalence = record
    {refl = Membership.≋-refl S
    ; sym = Membership.≋-sym S
    ; trans = Membership.≋-trans S
    }
  }
shift-map : (F : S ⟶ T) (l : List (Carrier S)) → elem-of (map (_⟨$⟩_ F) l) ≅ shifted F l
shift-map F l = record
  {to = record
    {_⟨$⟩_ = λ {(El belongs₁) → copy-unfunc F belongs₁}
    ; cong = copy-unfunc-cong F}
  ; from = record
    {_⟨$⟩_ = λ {x → Membership.El (copy-func F (elem x))}
    ; cong = λ {i} {j} i≈j → copy-func-cong F {i} {j} i≈j}   -- need to eta expand
  ; inverse-of = record
    {left-inverse-of = left-inv
    ; right-inverse-of = right-inv}
  }
shifted-cong : (F : S ⟶ T) {xs ys : List (Carrier S)} (xs≈ys : xs ⇔ ys) → shifted F xs ≅ shifted F ys
shifted-cong F xs≈ys = record
  {to = record
    {_⟨$⟩_ = λ sh → record
      {elem = Membership.El (belongs (to xs≈ys ⟨$⟩ (elem sh)))
      ; shift-witness = F ⟨$⟩ Membership.witness (to xs≈ys ⟨$⟩ elem sh)
      ; sw-good = refl T
      }
    ; cong = cong (to xs≈ys)}
  ; from = record
    {_⟨$⟩_ = λ sh → record
      {elem = Membership.El (belongs (from xs≈ys ⟨$⟩ elem sh))
      ; sw-good = refl T
      }
    ; cong = cong (from xs≈ys)}
  ; inverse-of = record
    {left-inverse-of = λ sh → left-inverse-of xs≈ys (elem sh)
    ; right-inverse-of = λ sh → right-inverse-of xs≈ys (elem sh)
    }
  }
```

27.8 Properties of singleton lists

```
module ElemOfSing {ℓS ℓs} (X : Setoid ℓS ℓs) where
  open Setoid X renaming (Carrier to X₀)
  open BagEq X
  open Membership X
  open Locations X
  open LocEquiv X
  open SetoidCombinators X
  singleton-≈ : {i j : X₀} (i≈j : i ≈ j) → (i :: []) ⇔ (j :: [])
  singleton-≈ {i} {j} i≈j = record
    {to = record {_⟨$⟩_ = ∈a→∈b i≈j; cong = cong-to i≈j}
    ; from = record {_⟨$⟩_ = ∈a→∈b (sym i≈j); cong = cong-to (sym i≈j)}
    ; inverse-of = record
      {left-inverse-of = inv i≈j (sym i≈j)
      ; right-inverse-of = inv (sym i≈j) i≈j}
    }
    where
      ∈a→∈b : {a b : X₀} → a ≈ b → elements (a :: []) → elements (b :: [])
      ∈a→∈b a≈b (Membership.El (Locations.here sm)) = El (here (sm ⟨≈≈⟩ a≈b))
      ∈a→∈b _ (Membership.El (Locations.there ()))
      cong-to : {a b : X₀} → (a≈b : a ≈ b) → {∈a₁ ∈a₂ : elements (a :: [])}
        → belongs ∈a₁ ≋ belongs ∈a₂ → belongs (∈a→∈b a≈b ∈a₁) ≋ belongs (∈a→∈b a≈b ∈a₂)
      cong-to a≈b (LocEquiv.hereEq x≈z y≈z) = LocEquiv.hereEq (x≈z ⟨≈≈⟩ a≈b) (y≈z ⟨≈≈⟩ a≈b)
      cong-to _ (LocEquiv.thereEq ())
      inv : {a b : X₀} (a≈b : a ≈ b) (b≈a : b ≈ a) (x : elements (a :: [])) →
        belongs (∈a→∈b b≈a (∈a→∈b a≈b x)) ≋ belongs x
      inv a≈b b≈a (El (here sm)) = LocEquiv.hereEq ((sm ⟨≈≈⟩ a≈b) ⟨≈≈⟩ b≈a) sm
      inv a≈b b≈a (El (there ()))
```

27.9 Properties of fold over list

```
module ElemOfFold {ℓS ℓs} (X : Setoid ℓS ℓs) where
  open Setoid X renaming (Carrier to X₀)
  open BagEq X
  open Membership X
  open Locations X
  open LocEquiv X
  open import Data.List
  open CommMonoid
  open ElemOf[] X
  open _≅_
  fold-cong : {CM : CommMonoid X} {i j : List X₀} → i ⇔ j
    → foldr (_*_ CM) (e CM) i ≈ foldr (_*_ CM) (e CM) j
  fold-cong {CM} {[]} {[]} i⇔j = refl
  fold-cong {CM} {[]} {x :: j} i⇔j = ⊥-elim (elem-of-[] (from i⇔j ⟨$⟩ (El (here refl))))
  fold-cong {CM} {x :: i} {[]} i⇔j = ⊥-elim (elem-of-[] (to i⇔j ⟨$⟩ (El (here refl))))
  fold-cong {CM} {x :: i} {y :: j} i⇔j with (to i⇔j ⟨$⟩ (El (here refl)))
  ... | El pos = {!!}
```

28 Some

```
module Some where
open import Level renaming (zero to lzero; suc to lsuc) hiding (lift)
open import Relation.Binary using (Setoid; IsEquivalence; Rel;
  Reflexive; Symmetric; Transitive)
open import Function.Equality using (Π; _⟶_; id; _∘_; _⟨$⟩_; cong)
open import Function using (_$_) renaming (id to id₀; _∘_ to _⊚_)
open import Function.Equivalence using (Equivalence)
open import Data.List using (List; []; _++_; _::_; map)
open import Data.Nat using (ℕ; zero; suc)
open import EqualityCombinators
open import DataProperties
open import SetoidEquiv
open import ParComp
open import TypeEquiv using (swap₊)
open import SetoidSetoid
```

The goal of this section is to capture a notion that we have a proof of a property P of an element x belonging to a list xs. But we don't want just any proof, but we want to know *which* x ∈ xs is the witness. However, we are in the Setoid setting, and in a setting where multiplicity matters (i.e. we may have x occuring twice in xs, yielding two different proofs that P holds). And we do not care very much about the exact x, any y such that x ≈ y will do, as long as it is in the "right" location.

And then we want to capture the idea of when two such are equivalent – when is it that Some P xs is just as good as Some P ys? In fact, we'll generalize this some more to Some Q ys.

For the purposes of CommMonoid however, all we really need is some notion of Bag Equivalence. However, many of the properties we need to establish are simpler if we generalize to the situation described above.

28.1 Some_0

Setoid-based variant of Any.

Quite a bit of this is directly inspired by Data.List.Any and Data.List.Any.Properties.

$\boxed{\textbf{[WK:}}$ *A ⟶ SSetoid _ _ is a pretty strong assumption. Logical equivalence does not ask for the two morphisms back and forth to be inverse.* $\boxed{]}$ $\boxed{\textbf{[JC:}}$ *This is pretty much directly influenced by Nisse's paper: logical equivalence only gives Set, not Multiset, at least if used for the equivalence of over List. To get Multiset, we need to preserve full equivalence, i.e. capture permutations. My reason to use A ⟶ SSetoid _ _ is to mesh well with the rest. It is not cast in stone and can potentially be weakened.* $\boxed{]}$

```
module Locations {ℓS ℓs ℓp : Level} (S : Setoid ℓS ℓs) (P₀ : Setoid.Carrier S → Set ℓp) where
  open Setoid S renaming (Carrier to A)
  data Some₀ : List A → Set ((ℓS ⊔ ℓs) ⊔ ℓp) where
    here : {x a : A} {xs : List A} (sm : a ≈ x) (px : P₀ a) → Some₀ (x :: xs)
    there : {x : A} {xs : List A} (pxs : Some₀ xs) → Some₀ (x :: xs)
```

Inhabitants of Some_0 really are just locations: $\mathsf{Some}_0\ \mathsf{P}\ \mathsf{xs} \cong \Sigma\ \mathsf{i} : \mathsf{Fin}\ (\mathsf{length}\ \mathsf{xs}) \bullet \mathsf{P}\ (\mathsf{x}\ !\ \mathsf{i})$. Thus one possibility is to go with natural numbers directly, but that seems awkward. Nevertheless, the 'location' function is straightforward:

```
toℕ : {xs : List A} → Some₀ xs → ℕ
toℕ (here _ _) = 0
toℕ (there pf) = suc (toℕ pf)
```

We need to know when two locations are the same. We need to be proving the same property P_0, but we can have different (but equivalent) witnesses.

```
module _ {ℓS ℓs ℓP} {S : Setoid ℓS ℓs} {P₀ : Setoid.Carrier S → Set ℓP} where
  open Setoid S renaming (Carrier to A)
  open Locations
  infix 3 _≋_
  data _≋_ : {ys : List A} (pf pf' : Some₀ S P₀ ys) → Set (ℓS ⊔ ℓs) where
    hereEq : {xs : List A} {x y z : A} (px : P₀ x) (qy : P₀ y)
      → (x≈z : x ≈ z) → (y≈z : y ≈ z)
      → _≋_ (here {x = z} {x} {xs} x≈z px) (here {x = z} {y} {xs} y≈z qy)
    thereEq : {xs : List A} {x : A} {pxs : Some₀ S P₀ xs} {qxs : Some₀ S P₀ xs}
      → _≋_ pxs qxs → _≋_ (there {x = x} pxs) (there {x = x} qxs)
```

Notice that these are another form of "natural numbers" whose elements are of the form $\mathsf{thereEq}^{\mathsf{n}}$ (hereEq Px Qx _ _) for some $\mathsf{n} : \mathbb{N}$.

It is on purpose that _≋_ preserves positions. Suppose that we take the setoid of the Latin alphabet, with _≈_ identifying upper and lower case. There should be 3 elements of _≋_ for a :: A :: a :: [], not 6. When we get to defining BagEq, there will be 6 different ways in which that list, as a Bag, is equivalent to itself.

```
  ≋-refl : {xs : List A} {p : Some₀ S P₀ xs} → p ≋ p
  ≋-refl {p = here a≈x px} = hereEq px px a≈x a≈x
  ≋-refl {p = there p} = thereEq ≋-refl

  ≋-sym : {xs : List A} {p : Some₀ S P₀ xs} {q : Some₀ S P₀ xs} → p ≋ q → q ≋ p
  ≋-sym (hereEq a≈x b≈x px py) = hereEq b≈x a≈x py px
  ≋-sym (thereEq eq) = thereEq (≋-sym eq)

  ≋-trans : {xs : List A} {p q r : Some₀ S P₀ xs}
    → p ≋ q → q ≋ r → p ≋ r
  ≋-trans (hereEq pa qb a≈x b≈x) (hereEq pc qd c≈y d≈y) = hereEq pa qd _ _
  ≋-trans (thereEq e) (thereEq f) = thereEq (≋-trans e f)

  ≡→≋ : {xs : List A} {p q : Some₀ S P₀ xs} → p ≡ q → p ≋ q
  ≡→≋ ≡.refl = ≋-refl
module _ {ℓS ℓs ℓP} {S : Setoid ℓS ℓs} (P₀ : Setoid.Carrier S → Set ℓP) where
  open Setoid S
  open Locations

  Some : List Carrier → Setoid ((ℓS ⊔ ℓs) ⊔ ℓP) (ℓS ⊔ ℓs)
  Some xs = record
    {Carrier       = Some₀ S P₀ xs
    ; _≈_          = _≋_
    ; isEquivalence = record {refl = ≋-refl; sym = ≋-sym; trans = ≋-trans}
    }

  ≡→Some : {xs ys : List (Setoid.Carrier S)} → xs ≡ ys → Some xs ≅ Some ys
  ≡→Some ≡.refl = ≅-refl
```

28.2 Membership module

First, define a few convenient combinators for equational reasoning in Setoid.

```
module Membership {ℓS ℓs : Level} (S : Setoid ℓS ℓs) where
  open Locations

  open SetoidCombinators S public
  open Setoid S
```

setoid≈ x is actually a mapping from S to SSetoid _ _; it maps elements y of Carrier S to the setoid of "x $\approx_s$ y".

```
  -- the levels might be off
setoid≈ : Carrier → (S ⟶ ProofSetoid ℓs ℓs)
setoid≈ x = record
  {_⟨$⟩_ = λ s → _≈S_ {S = S} x s
  ; cong = λ i≈j → record
    {to   = record {_⟨$⟩_ = λ x≈i → x≈i ⟨≈≈⟩ i≈j; cong = λ _ → tt}
    ; from = record {_⟨$⟩_ = λ x≈i → x≈i ⟨≈≈˘⟩ i≈j; cong = λ _ → tt}}}
infix 4 _∈₀_ _∈_
_∈_ : Carrier → List Carrier → Setoid (ℓS ⊔ ℓs) (ℓS ⊔ ℓs)
x ∈ xs = Some {S = S} (_≈_ x) xs
_∈₀_ : Carrier → List Carrier → Set (ℓS ⊔ ℓs)
x ∈₀ xs = Setoid.Carrier (x ∈ xs)
∈₀-subst₁ : {x y : Carrier} {xs : List Carrier} → x ≈ y → x ∈₀ xs → y ∈₀ xs
∈₀-subst₁ {x} {y} {∘ (_ :: _)} x≈y (here a≈x px) = here a≈x (sym x≈y ⟨≈≈⟩ px)
∈₀-subst₁ {x} {y} {∘ (_ :: _)} x≈y (there x∈xs) = there (∈₀-subst₁ x≈y x∈xs)
∈₀-subst₁-cong : {x y : Carrier} {xs : List Carrier} (x≈y : x ≈ y)
                 {i j : x ∈₀ xs} → i ≋ j → ∈₀-subst₁ x≈y i ≋ ∈₀-subst₁ x≈y j
∈₀-subst₁-cong x≈y (hereEq px qy x≈z y≈z) = hereEq (sym x≈y ⟨≈≈⟩ px) (sym x≈y ⟨≈≈⟩ qy) x≈z y≈z
∈₀-subst₁-cong x≈y (thereEq i≋j) = thereEq (∈₀-subst₁-cong x≈y i≋j)
∈₀-subst₁-equiv : {x y : Carrier} {xs : List Carrier} → x ≈ y → (x ∈ xs) ≅ (y ∈ xs)
∈₀-subst₁-equiv {x} {y} {xs} x≈y = record
  {to = record {_⟨$⟩_ = ∈₀-subst₁ x≈y; cong = ∈₀-subst₁-cong x≈y}
  ; from = record {_⟨$⟩_ = ∈₀-subst₁ (sym x≈y); cong = ∈₀-subst₁-cong′}
  ; inverse-of = record {left-inverse-of = left-inv; right-inverse-of = right-inv}}
  where
    ∈₀-subst₁-cong′ : ∀ {ys} {i j : y ∈₀ ys} → i ≋ j → ∈₀-subst₁ (sym x≈y) i ≋ ∈₀-subst₁ (sym x≈y) j
    ∈₀-subst₁-cong′ (hereEq px qy x≈z y≈z) = hereEq (sym (sym x≈y) ⟨≈≈⟩ px) (sym (sym x≈y) ⟨≈≈⟩ qy) x≈z y≈z
    ∈₀-subst₁-cong′ (thereEq i≋j) = thereEq (∈₀-subst₁-cong′ i≋j)
    left-inv : ∀ {ys} (x∈ys : x ∈₀ ys) → ∈₀-subst₁ (sym x≈y) (∈₀-subst₁ x≈y x∈ys) ≋ x∈ys
    left-inv (here sm px) = hereEq (sym (sym x≈y) ⟨≈≈⟩ (sym x≈y ⟨≈≈⟩ px)) px sm sm
    left-inv (there x∈ys) = thereEq (left-inv x∈ys)
    right-inv : ∀ {ys} (y∈ys : y ∈₀ ys) → ∈₀-subst₁ x≈y (∈₀-subst₁ (sym x≈y) y∈ys) ≋ y∈ys
    right-inv (here sm px) = hereEq (sym x≈y ⟨≈≈⟩ (sym (sym x≈y) ⟨≈≈⟩ px)) px sm sm
    right-inv (there y∈ys) = thereEq (right-inv y∈ys)
infix 3 _≋₀_
data _≋₀_ : {ys : List Carrier} {y y′ : Carrier} → y ∈₀ ys → y′ ∈₀ ys → Set (ℓS ⊔ ℓs) where
  hereEq : {xs : List Carrier} {x y y′ z z′ : Carrier}
    → (y≈x : y ≈ x) (z≈y : z ≈ y) (y′≈x : y′ ≈ x) (z′≈y′ : z′ ≈ y′)
    → _≋₀_ (here {x = x} {y} {xs} y≈x z≈y) (here {x = x} {y′} {xs} y′≈x z′≈y′)
  thereEq : {xs : List Carrier} {x y y′ : Carrier} {y∈xs : y ∈₀ xs} {y′∈xs : y′ ∈₀ xs}
          → y∈xs ≋₀ y′∈xs → _≋₀_ (there {x = x} y∈xs) (there {x = x} y′∈xs)
≋→≋₀ : {ys : List Carrier} {y : Carrier} {pf pf' : y ∈₀ ys}
        → pf ≋ pf' → pf ≋₀ pf'
≋→≋₀ (hereEq _ _ _ _) = hereEq _ _ _ _
≋→≋₀ (thereEq eq) = thereEq (≋→≋₀ eq)
≋₀-refl : {xs : List Carrier} {x : Carrier} {p : x ∈₀ xs} → p ≋₀ p
≋₀-refl {p = here _ _} = hereEq _ _ _ _
≋₀-refl {p = there p} = thereEq ≋₀-refl
≋₀-sym : {xs : List Carrier} {x y : Carrier} {p : x ∈₀ xs} {q : y ∈₀ xs} → p ≋₀ q → q ≋₀ p
≋₀-sym (hereEq a≈x b≈x px py) = hereEq px py a≈x b≈x
≋₀-sym (thereEq eq) = thereEq (≋₀-sym eq)
```

```
≋₀-trans : {xs : List Carrier} {x y z : Carrier} {p : x ∈₀ xs} {q : y ∈₀ xs} {r : z ∈₀ xs}
  → p ≋₀ q → q ≋₀ r → p ≋₀ r
≋₀-trans (hereEq pa qb a≈x b≈x) (hereEq pc qd c≈y d≈y) = hereEq _ _ _ _
≋₀-trans (thereEq e) (thereEq f) = thereEq (≋₀-trans e f)

record BagEq (xs ys : List Carrier) : Set (ℓS ⊔ ℓs) where
  constructor BE
  field
    permut : {x : Carrier} → (x ∈ xs) ≅ (x ∈ ys)
    repr-indep-to : {x x' : Carrier} {x∈xs : x ∈₀ xs} {x'∈xs : x' ∈₀ xs} (x≈x' : x ≈ x') →
                    (x∈xs ≋₀ x'∈xs) → _≅_.to (permut {x}) ⟨$⟩ x∈xs ≋₀ _≅_.to (permut {x'}) ⟨$⟩ x'∈xs
    repr-indep-fr : {y y' : Carrier} {y∈ys : y ∈₀ ys} {y'∈ys : y' ∈₀ ys} (y≈y' : y ≈ y') →
                    (y∈ys ≋₀ y'∈ys) → _≅_.from (permut {y}) ⟨$⟩ y∈ys ≋₀ _≅_.from (permut {y'}) ⟨$⟩ y'∈ys

open BagEq

BE-refl : {xs : List Carrier} → BagEq xs xs
BE-refl = BE ≅-refl (λ _ pf → pf) (λ _ pf → pf)

BE-sym : {xs ys : List Carrier} → BagEq xs ys → BagEq ys xs
BE-sym (BE p ind-to ind-fr) = BE (≅-sym p) ind-fr ind-to

BE-trans : {xs ys zs : List Carrier} → BagEq xs ys → BagEq ys zs → BagEq xs zs
BE-trans (BE p₀ to₀ fr₀) (BE p₁ to₁ fr₁) =
  BE (≅-trans p₀ p₁) (λ x≈x' pf → to₁ x≈x' (to₀ x≈x' pf)) (λ y≈y' pf → fr₀ y≈y' (fr₁ y≈y' pf))

∈₀-Subst₂ : {x : Carrier} {xs ys : List Carrier} → BagEq xs ys → x ∈ xs ⟶ x ∈ ys
∈₀-Subst₂ {x} xs≅ys = _≅_.to (permut xs≅ys {x})

∈₀-subst₂ : {x : Carrier} {xs ys : List Carrier} → BagEq xs ys → x ∈₀ xs → x ∈₀ ys
∈₀-subst₂ xs≅ys x∈xs = ∈₀-Subst₂ xs≅ys ⟨$⟩ x∈xs

∈₀-subst₂-cong : {x : Carrier} {xs ys : List Carrier} (xs≅ys : BagEq xs ys)
               → {p q : x ∈₀ xs}
               → p ≋ q
               → ∈₀-subst₂ xs≅ys p ≋ ∈₀-subst₂ xs≅ys q
∈₀-subst₂-cong xs≅ys = cong (∈₀-Subst₂ xs≅ys)

transport : {ℓQ ℓq : Level} → (Q : S ⟶ ProofSetoid ℓQ ℓq) →
  let Q₀ = λ e → Setoid.Carrier (Q ⟨$⟩ e) in
  {a x : Carrier} (p : Q₀ a) (a≈x : a ≈ x) → Q₀ x
transport Q p a≈x = Equivalence.to (Π.cong Q a≈x) ⟨$⟩ p

∈₀-subst₁-elim : {x : Carrier} {xs : List Carrier} (x∈xs : x ∈₀ xs) →
  ∈₀-subst₁ refl x∈xs ≋ x∈xs
∈₀-subst₁-elim (here sm px) = hereEq (refl ⟨≈˘≈⟩ px) px sm sm
∈₀-subst₁-elim (there x∈xs) = thereEq (∈₀-subst₁-elim x∈xs)

  -- note how the back-and-forth is clearly apparent below
∈₀-subst₁-sym : {a b : Carrier} {xs : List Carrier} {a≈b : a ≈ b}
  {a∈xs : a ∈₀ xs} {b∈xs : b ∈₀ xs} → ∈₀-subst₁ a≈b a∈xs ≋ b∈xs →
  ∈₀-subst₁ (sym a≈b) b∈xs ≋ a∈xs
∈₀-subst₁-sym {a≈b = a≈b} {here sm px} {here sm₁ px₁} (hereEq _ .px₁ .sm .sm₁) = hereEq (sym (sym a≈b) ⟨≈≈⟩ px₁) px sm₁ sm
∈₀-subst₁-sym {a∈xs = there a∈xs} {here sm px} ()
∈₀-subst₁-sym {a∈xs = here sm px} {there b∈xs} ()
∈₀-subst₁-sym {a∈xs = there a∈xs} {there b∈xs} (thereEq pf) = thereEq (∈₀-subst₁-sym pf)

∈₀-subst₁-trans : {a b c : Carrier} {xs : List Carrier} {a≈b : a ≈ b}
  {b≈c : b ≈ c} {a∈xs : a ∈₀ xs} {b∈xs : b ∈₀ xs} {c∈xs : c ∈₀ xs} →
  ∈₀-subst₁ a≈b a∈xs ≋ b∈xs → ∈₀-subst₁ b≈c b∈xs ≋ c∈xs →
  ∈₀-subst₁ (a≈b ⟨≈≈⟩ b≈c) a∈xs ≋ c∈xs
∈₀-subst₁-trans {a≈b = a≈b} {b≈c} {here sm px} {∘ (here y≈z qy)} {∘ (here z≈w qz)} (hereEq ._ qy .sm y≈z) (hereEq ._ qz foo z≈w)
∈₀-subst₁-trans {a≈b = a≈b} {b≈c} {there a∈xs} {there b∈xs} {∘ (there _)} (thereEq pp) (thereEq qq) = thereEq (∈₀-subst₁-trans pp
```

28.3 ++≅ : ⋯ → (Some P xs ⊎⊎ Some P ys) ≅ Some P (xs + ys)

```
module _ {ℓS ℓs ℓP : Level} {A : Setoid ℓS ℓs} {P₀ : Setoid.Carrier A → Set ℓP} where
  ++≅ : {xs ys : List (Setoid.Carrier A)} → (Some P₀ xs ⊎⊎ Some P₀ ys) ≅ Some P₀ (xs + ys)
  ++≅ {xs} {ys} = record
    {to = record {_⟨$⟩_ = ⊎→++; cong = ⊎→++-cong}
    ; from = record {_⟨$⟩_ = ++→⊎ xs; cong = new-cong xs}
    ; inverse-of = record
      {left-inverse-of = lefty xs
      ; right-inverse-of = righty xs
      }
    }
    where
      open Setoid A
      open Locations
      _∽_ = _≋_; ∽-refl = ≋-refl {S = A} {P₀}
        -- "ealier"
      ⊎→ˡ : ∀ {ws zs} → Some₀ A P₀ ws → Some₀ A P₀ (ws + zs)
      ⊎→ˡ (here p a≈x) = here p a≈x
      ⊎→ˡ (there p) = there (⊎→ˡ p)
      yo : {xs : List Carrier} {x y : Some₀ A P₀ xs} → x ∽ y → ⊎→ˡ x ∽ ⊎→ˡ y
      yo (hereEq px py _ _) = hereEq px py _ _
      yo (thereEq pf) = thereEq (yo pf)
        -- "later"
      ⊎→ʳ : ∀ xs {ys} → Some₀ A P₀ ys → Some₀ A P₀ (xs + ys)
      ⊎→ʳ [] p = p
      ⊎→ʳ (x :: xs) p = there (⊎→ʳ xs p)
      oy : (xs : List Carrier) {x y : Some₀ A P₀ ys} → x ∽ y → ⊎→ʳ xs x ∽ ⊎→ʳ xs y
      oy [] pf = pf
      oy (x :: xs) pf = thereEq (oy xs pf)
        -- Some₀ is ++→⊎-homomorphic, in the second argument.
      ⊎→++ : ∀ {zs ws} → (Some₀ A P₀ zs ⊎ Some₀ A P₀ ws) → Some₀ A P₀ (zs + ws)
      ⊎→++ (inj₁ x) = ⊎→ˡ x
      ⊎→++ {zs} (inj₂ y) = ⊎→ʳ zs y
      ++→⊎ : ∀ xs {ys} → Some₀ A P₀ (xs + ys) → Some₀ A P₀ xs ⊎ Some₀ A P₀ ys
      ++→⊎ []        p         = inj₂ p
      ++→⊎ (x :: l) (here p _) = inj₁ (here p _)
      ++→⊎ (x :: l) (there p)  = (there ⊎₁ id₀) (++→⊎ l p)
        -- all of the following may need to change
      ⊎→++-cong : {a b : Some₀ A P₀ xs ⊎ Some₀ A P₀ ys} → (_∽_ ∥ _∽_) a b → ⊎→++ a ∽ ⊎→++ b
      ⊎→++-cong (left x₁~x₂)  = yo x₁~x₂
      ⊎→++-cong (right y₁~y₂) = oy xs y₁~y₂
      ∽∥∽-cong : {xs ys us vs : List Carrier}
                 (F : Some₀ A P₀ xs → Some₀ A P₀ us)
                 (F-cong : {p q : Some₀ A P₀ xs} → p ∽ q → F p ∽ F q)
                 (G : Some₀ A P₀ ys → Some₀ A P₀ vs)
                 (G-cong : {p q : Some₀ A P₀ ys} → p ∽ q → G p ∽ G q)
                 → {pf pf' : Some₀ A P₀ xs ⊎ Some₀ A P₀ ys}
                 → (_∽_ ∥ _∽_) pf pf' → (_∽_ ∥ _∽_) ((F ⊎₁ G) pf) ((F ⊎₁ G) pf')
      ∽∥∽-cong F F-cong G G-cong (left x~₁y) = left (F-cong x~₁y)
      ∽∥∽-cong F F-cong G G-cong (right x~₂y) = right (G-cong x~₂y)
      new-cong : (xs : List Carrier) {i j : Some₀ A P₀ (xs + ys)} → i ∽ j → (_∽_ ∥ _∽_) (++→⊎ xs i) (++→⊎ xs j)
      new-cong [] pf = right pf
```

```
new-cong (x :: xs) (hereEq px py _ _) = left (hereEq px py _ _)
new-cong (x :: xs) (thereEq pf) = ∽‖∽-cong there thereEq id₀ id₀ (new-cong xs pf)
lefty : (xs {ys} : List Carrier) (p : Some₀ A P₀ xs ⊎ Some₀ A P₀ ys) → (_∽_ ‖ _∽_) (++→⊎ xs (⊎→++ p)) p
lefty [] (inj₁ ())
lefty [] (inj₂ p) = right ≋-refl
lefty (x :: xs) (inj₁ (here px _)) = left ∽-refl
lefty (x :: xs) {ys} (inj₁ (there p)) with ++→⊎ xs {ys} (⊎→++ (inj₁ p)) | lefty xs {ys} (inj₁ p)
... | inj₁ _ | (left x~₁y) = left (thereEq x~₁y)
... | inj₂ _ | ()
lefty (z :: zs) {ws} (inj₂ p) with ++→⊎ zs {ws} (⊎→++ {zs} (inj₂ p)) | lefty zs (inj₂ p)
... | inj₁ x | ()
... | inj₂ y | (right x~₂y) = right x~₂y
righty : (zs {ws} : List Carrier) (p : Some₀ A P₀ (zs + ws)) → (⊎→++ (++→⊎ zs p)) ∽ p
righty [] {ws} p = ∽-refl
righty (x :: zs) {ws} (here px _) = ∽-refl
righty (x :: zs) {ws} (there p) with ++→⊎ zs p | righty zs p
... | inj₁ _  | res = thereEq res
... | inj₂ _  | res = thereEq res
```

28.4 Bottom as a setoid

```
⊥⊥ : ∀ {ℓS ℓs} → Setoid ℓS ℓs
⊥⊥ = record
  {Carrier = ⊥
  ; _≈_ = λ _ _ → ⊤
  ; isEquivalence = record {refl = tt; sym = λ _ → tt; trans = λ _ _ → tt}
  }
```

```
module _ {ℓS ℓs ℓP ℓp : Level} {S : Setoid ℓS ℓs} {P : S ⟶ ProofSetoid ℓP ℓp} where
  ⊥≅Some[] : ⊥⊥ {(ℓS ⊔ ℓs) ⊔ ℓP} {(ℓS ⊔ ℓs) ⊔ ℓp} ≅ Some {S = S} (λ e → Setoid.Carrier (P ⟨$⟩ e)) []
  ⊥≅Some[] = record
    {to         = record {_⟨$⟩_ = λ {()}; cong = λ {{()}}}
    ; from       = record {_⟨$⟩_ = λ {()}; cong = λ {{()}}}
    ; inverse-of = record {left-inverse-of = λ _ → tt; right-inverse-of = λ {()}}
    }
```

28.5 map≅ : ···→ Some (P ∘ f) xs ≅ Some P (map (_⟨$⟩_ f) xs)

```
map≅ : {ℓS ℓs ℓP ℓp : Level} {A B : Setoid ℓS ℓs} {P : B ⟶ ProofSetoid ℓP ℓp} →
  let P₀ = λ e → Setoid.Carrier (P ⟨$⟩ e) in
  {f : A ⟶ B} {xs : List (Setoid.Carrier A)} →
  Some {S = A} (P₀ ⊚ (_⟨$⟩_ f)) xs ≅ Some {S = B} P₀ (map (_⟨$⟩_ f) xs)
map≅ {A = A} {B} {P} {f} = record
  {to = record {_⟨$⟩_ = map⁺; cong = map⁺-cong}
  ; from = record {_⟨$⟩_ = map⁻; cong = map⁻-cong}
  ; inverse-of = record {left-inverse-of = map⁻∘map⁺; right-inverse-of = map⁺∘map⁻}
  }
  where
  open Setoid
  open Membership using (transport)
  A₀ = Setoid.Carrier A
```

```
  open Locations
  _∼_ = _≋_ {S = B}
  P₀ = λ e → Setoid.Carrier (P ⟨$⟩ e)

  map⁺ : {xs : List A₀} → Some₀ A (P₀ ⊚ _⟨$⟩_ f) xs → Some₀ B P₀ (map (_⟨$⟩_ f) xs)
  map⁺ (here a≈x p) = here (Π.cong f a≈x) p
  map⁺ (there p) = there $ map⁺ p

  map⁻ : {xs : List A₀} → Some₀ B P₀ (map (_⟨$⟩_ f) xs) → Some₀ A (P₀ ⊚ (_⟨$⟩_ f)) xs
  map⁻ {[]} ()
  map⁻ {x :: xs} (here {b} b≈x p) = here (refl A) (Equivalence.to (Π.cong P b≈x) ⟨$⟩ p)
  map⁻ {x :: xs} (there p) = there (map⁻ {xs = xs} p)

  map⁺∘map⁻ : {xs : List A₀} → (p : Some₀ B P₀ (map (_⟨$⟩_ f) xs)) → map⁺ (map⁻ p) ∼ p
  map⁺∘map⁻ {[]} ()
  map⁺∘map⁻ {x :: xs} (here b≈x p) = hereEq (transport B P p b≈x) p (Π.cong f (refl A)) b≈x
  map⁺∘map⁻ {x :: xs} (there p) = thereEq (map⁺∘map⁻ p)

  map⁻∘map⁺ : {xs : List A₀} → (p : Some₀ A (P₀ ⊚ (_⟨$⟩_ f)) xs)
    → let _∼₂_ = _≋_ {P₀ = P₀ ⊚ (_⟨$⟩_ f)} in map⁻ (map⁺ p) ∼₂ p
  map⁻∘map⁺ {[]} ()
  map⁻∘map⁺ {x :: xs} (here a≈x p) = hereEq (transport A (P ∘ f) p a≈x) p (refl A) a≈x
  map⁻∘map⁺ {x :: xs} (there p) = thereEq (map⁻∘map⁺ p)

  map⁺-cong : {ys : List A₀} {i j : Some₀ A (P₀ ⊚ _⟨$⟩_ f) ys} → _≋_ {P₀ = P₀ ⊚ _⟨$⟩_ f} i j → map⁺ i ∼ map⁺ j
  map⁺-cong (hereEq px py x≈z y≈z) = hereEq px py (Π.cong f x≈z) (Π.cong f y≈z)
  map⁺-cong (thereEq i∼j) = thereEq (map⁺-cong i∼j)

  map⁻-cong : {ys : List A₀} {i j : Some₀ B P₀ (map (_⟨$⟩_ f) ys)} → i ∼ j → _≋_ {P₀ = P₀ ⊚ _⟨$⟩_ f} (map⁻ i) (map⁻ j)
  map⁻-cong {[]} ()
  map⁻-cong {z :: zs} (hereEq {x = x} {y} px py x≈z y≈z) =
    hereEq (transport B P px x≈z) (transport B P py y≈z) (refl A) (refl A)
  map⁻-cong {z :: zs} (thereEq i∼j) = thereEq (map⁻-cong i∼j)
```

28.6 FindLose

```
module FindLose {ℓS ℓs ℓP ℓp : Level} {A : Setoid ℓS ℓs} (P : A ⟶ ProofSetoid ℓP ℓp) where
  open Membership A
  open Setoid A
  open Π
  open _≅_
  open Locations
  private
    P₀ = λ e → Setoid.Carrier (P ⟨$⟩ e)
    Support = λ ys → Σ y : Carrier • y ∈₀ ys × P₀ y

  find : {ys : List Carrier} → Some₀ A P₀ ys → Support ys
  find {y :: ys} (here {a} a≈y p) = a , here a≈y (sym a≈y) , transport P p a≈y
  find {y :: ys} (there p) = let (a , a∈ys , Pa) = find p
                             in a , there a∈ys , Pa

  lose : {ys : List Carrier} → Support ys → Some₀ A P₀ ys
  lose (y , here b≈y py , Py) = here b≈y (Equivalence.to (Π.cong P py) Π.⟨$⟩ Py)
  lose (y , there {b} y∈ys , Py) = there (lose (y , y∈ys , Py))
```

28.7 Σ-Setoid

[WK: *Abstruse name!* **]** **[JC:** *Feel free to rename. I agree that it is not a good name. I was more concerned with the semantics, and then could come back to clean up once it worked.* **]**

This is an "unpacked" version of Some, where each piece (see Support below) is separated out. For some equivalences, it seems to work with this representation.

```
module _ {ℓS ℓs ℓP ℓp : Level} (A : Setoid ℓS ℓs) (P : A ⟶ ProofSetoid ℓP ℓp) where
  open Membership A
  open Setoid A
  private
    P₀ : (e : Carrier) → Set ℓP
    P₀ = λ e → Setoid.Carrier (P ⟨$⟩ e)
    Support : (ys : List Carrier) → Set (ℓS ⊔ (ℓs ⊔ ℓP))
    Support = λ ys → Σ y ∶ Carrier • y ∈₀ ys × P₀ y
    squish : {x y : Setoid.Carrier A} → P₀ x → P₀ y → Set ℓp
    squish _ _ = ⊤

  open Locations
  open BagEq

    -- FIXME : this definition is still not right. ≋₀ or ≋ + ∈₀-subst₁ ?
  _∻_ : {ys : List Carrier} → Support ys → Support ys → Set ((ℓs ⊔ ℓS) ⊔ ℓp)
  (a , a∈xs , Pa) ∻ (b , b∈xs , Pb) =
    Σ (a ≈ b) (λ a≈b → a∈xs ≋₀ b∈xs × squish Pa Pb)

  Σ-Setoid : (ys : List Carrier) → Setoid ((ℓS ⊔ ℓs) ⊔ ℓP) ((ℓS ⊔ ℓs) ⊔ ℓp)
  Σ-Setoid [] = ⊥⊥ {ℓP ⊔ (ℓS ⊔ ℓs)}
  Σ-Setoid (y :: ys) = record
    {Carrier = Support (y :: ys)
    ; _≈_ = _∻_
    ; isEquivalence = record
      {refl = λ {s} → Refl {s}
      ; sym = λ {s} {t} eq → Sym {s} {t} eq
      ; trans = λ {s} {t} {u} a b → Trans {s} {t} {u} a b
      }
    }
    where
      Refl : Reflexive _∻_
      Refl {a₁ , here sm px , Pa} = refl , hereEq sm px sm px , tt
      Refl {a₁ , there a∈xs , Pa} = refl , thereEq ≋₀-refl , tt

      Sym : Symmetric _∻_
      Sym (a≈b , a∈xs≋b∈xs , Pa≈Pb) = sym a≈b , ≋₀-sym a∈xs≋b∈xs , tt

      Trans : Transitive _∻_
      Trans (a≈b , a∈xs≋b∈xs , Pa≈Pb) (b≈c , b∈xs≋c∈xs , Pb≈Pc) = trans a≈b b≈c , ≋₀-trans a∈xs≋b∈xs b∈xs≋c∈xs , tt

  module ∻ {ys} where open Setoid (Σ-Setoid ys) public

  open FindLose P

  find-cong : {xs : List Carrier} {p q : Some₀ A P₀ xs} → p ≋ q → find p ∻ find q
  find-cong {p = ∘ (here x≈z px)} {∘ (here y≈z qy)} (hereEq px qy x≈z y≈z) =
    refl , hereEq x≈z (sym x≈z) y≈z (sym y≈z) , tt
  find-cong {p = ∘ (there _)} {∘ (there _)} (thereEq p≋q) =
    proj₁ (find-cong p≋q) , thereEq (proj₁ (proj₂ (find-cong p≋q))) , proj₂ (proj₂ (find-cong p≋q))

  forget-cong : {xs : List Carrier} {i j : Support xs} → i ∻ j → lose i ≋ lose j
  forget-cong {i = a₁ , here sm px , Pa} {b , here sm₁ px₁ , Pb} (i≈j , a∈xs≋b∈xs) =
    hereEq (transport P Pa px) (transport P Pb px₁) sm sm₁
  forget-cong {i = a₁ , here sm px , Pa} {b , there b∈xs , Pb} (i≈j , () , _)
  forget-cong {i = a₁ , there a∈xs , Pa} {b , here sm px , Pb} (i≈j , () , _)
  forget-cong {i = a₁ , there a∈xs , Pa} {b , there b∈xs , Pb} (i≈j , thereEq pf , Pa≈Pb) =
    thereEq (forget-cong (i≈j , pf , Pa≈Pb))

  left-inv : {zs : List Carrier} (x∈zs : Some₀ A P₀ zs) → lose (find x∈zs) ≋ x∈zs
  left-inv (here {a} {x} a≈x px) = hereEq (transport P (transport P px a≈x) (sym a≈x)) px a≈x a≈x
```

```
  left-inv (there x∈ys) = thereEq (left-inv x∈ys)

  right-inv : {ys : List Carrier} (pf : Σ y ∶ Carrier • y ∈₀ ys × P₀ y) → find (lose pf) ≋ pf
  right-inv (y , here a≈x px , Py) = trans (sym a≈x) (sym px) , hereEq a≈x (sym a≈x) a≈x px , tt
  right-inv (y , there y∈ys , Py) =
    let (α₁ , α₂ , α₃) = right-inv (y , y∈ys , Py) in
    (α₁ , thereEq α₂ , α₃)

  Σ-Some : (xs : List Carrier) → Some {S = A} P₀ xs ≅ Σ-Setoid xs
  Σ-Some [] = ≅-sym (⊥≅Some[] {S = A} {P})
  Σ-Some (x :: xs) = record
    {to = record {_⟨$⟩_ = find; cong = find-cong}
    ; from = record {_⟨$⟩_ = lose; cong = forget-cong}
    ; inverse-of = record
      {left-inverse-of = left-inv
      ; right-inverse-of = right-inv
      }
    }

  Σ-cong : {xs ys : List Carrier} → BagEq xs ys → Σ-Setoid xs ≅ Σ-Setoid ys
  Σ-cong {[]} {[]} iso = ≅-refl
  Σ-cong {[]} {z :: zs} iso = ⊥-elim (_≅_.from (⊥≅Some[] {S = A} {setoid≈ z}) ⟨$⟩ (_≅_.from (permut iso) ⟨$⟩ here refl refl))
  Σ-cong {x :: xs} {[]} iso = ⊥-elim (_≅_.from (⊥≅Some[] {S = A} {setoid≈ x}) ⟨$⟩ (_≅_.to (permut iso) ⟨$⟩ here refl refl))
  Σ-cong {x :: xs} {y :: ys} xs≅ys = record
    {to   = record {_⟨$⟩_ = xs→ys xs≅ys; cong = λ {i j} → xs→ys-cong xs≅ys {i} {j}}
    ; from = record {_⟨$⟩_ = xs→ys (BE-sym xs≅ys); cong = λ {i j} → xs→ys-cong (BE-sym xs≅ys) {i} {j}}
    ; inverse-of = record
      {left-inverse-of = λ {(z , z∈xs , Pz) → refl , ≋→≋₀ (left-inverse-of (permut xs≅ys) z∈xs) , tt}
      ; right-inverse-of = λ {(z , z∈ys , Pz) → refl , ≋→≋₀ (right-inverse-of (permut xs≅ys) z∈ys) , tt}
      }
    }
    where
      open _≅_
      xs→ys : {zs ws : List Carrier} → BagEq zs ws → Support zs → Support ws
      xs→ys eq (a , a∈xs , Pa) = (a , ∈₀-subst₂ eq a∈xs , Pa)
        -- ∈₀-subst₁-equiv : x ≈ y → (x ∈ xs) ≅ (y ∈ xs)
      xs→ys-cong : {zs ws : List Carrier} (eq : BagEq zs ws) {i j : Support zs} →
        i ≋ j → xs→ys eq i ≋ xs→ys eq j
      xs→ys-cong eq {_ , a∈zs , _} {_ , b∈zs , _} (a≈b , pf , Pa≈Pb) =
        a≈b , repr-indep-to eq a≈b pf , tt
```

28.8 Some-cong

This isn't quite the full-powered cong, but is all we need.

$\boxed{\textbf{[WK:}}$ *It has position preservation neither in the assumption (list-rel), nor in the conclusion. Why did you bother with position preservation for _≋_?* $\boxed{\textbf{]}}$ $\boxed{\textbf{[JC:}}$ *Because _≋_ is about showing that two positions* in the same list *are equivalent. And list-rel is a permutation between two lists. I agree that _≋_ could be "loosened" to be up to permutation of elements which are _≈_ to a given one.*

But if our notion of permutation is BagEq, which depends on _∈_, which depends on Some, which depends on _≋_. If that now depends on BagEq, we've got a mutual recursion that seems unecessary. $\boxed{\textbf{]}}$

```
module _ {ℓS ℓs ℓP : Level} {A : Setoid ℓS ℓs} {P : A ⟶ ProofSetoid ℓP ℓs} where
  open Membership A
  open Setoid A
  private
```

```
    P₀ = λ e → Setoid.Carrier (P ⟨$⟩ e)
  Some-cong : {xs₁ xs₂ : List Carrier} →
    BagEq xs₁ xs₂ →
    Some P₀ xs₁ ≅ Some P₀ xs₂
  Some-cong {xs₁} {xs₂} xs₁≅xs₂ =
    Some P₀ xs₁       ≅⟨ Σ-Some A P xs₁ ⟩
    Σ-Setoid A P xs₁ ≅⟨ Σ-cong A P xs₁≅xs₂ ⟩
    Σ-Setoid A P xs₂ ≅⟨ ≅-sym (Σ-Some A P xs₂) ⟩
    Some P₀ xs₂ ■
```

29 CounterExample

This code used to be part of Some. It shows the reason why BagEq xs ys is not just {x} → x ∈ xs ≅ x ∈ ys : This is insufficiently representation independent.

```
module CounterExample where
open import Level renaming (zero to lzero; suc to lsuc) hiding (lift)
open import Relation.Binary    using (Setoid)
open import Function.Equality using (Π; _⟨$⟩_)
open import Data.List          using (List; _::_; [])
open import DataProperties     using (⊥)
open import SetoidEquiv
open import Some
```

29.1 Preliminaries

Define a kind of heterogeneous version of $\_\approx_0\_$, and some normal 'kit' to go with it.

```
module HetEquiv {ℓS ℓs : Level} (S : Setoid ℓS ℓs) where
  open Locations
  open Setoid S renaming (trans to _⟨≈≈⟩_)
  open Membership S
  ≋₀-strengthen : {ys : List Carrier} {y : Carrier} {pf pf' : y ∈₀ ys}
                → pf ≋₀ pf' → pf ≋ pf'
  ≋₀-strengthen (hereEq y≈x z≈y y′≈x z′≈y′) = hereEq z≈y z′≈y′ y≈x y′≈x
  ≋₀-strengthen (thereEq eq) = thereEq (≋₀-strengthen eq)
  infix 3 _□₀
  infixr 2 _≋₀⟨_⟩_
  infixr 2 _≋₀˘⟨_⟩_
  _≋₀⟨_⟩_ : {x y z : Carrier} {xs : List Carrier} (X : x ∈₀ xs) {Y : y ∈₀ xs} {Z : z ∈₀ xs}
        → X ≋₀ Y → Y ≋₀ Z → X ≋₀ Z
  X ≋₀⟨ X≋₀Y ⟩ Y≋₀Z = ≋₀-trans X≋₀Y Y≋₀Z
  _≋₀˘⟨_⟩_ : {x y z : Carrier} {xs : List Carrier} (X : x ∈₀ xs) {Y : y ∈₀ xs} {Z : z ∈₀ xs}
        → Y ≋₀ X → Y ≋₀ Z → X ≋₀ Z
  X ≋₀˘⟨ Y≋₀X ⟩ Y≋₀Z = ≋₀-trans (≋₀-sym Y≋₀X) Y≋₀Z
  _□₀ : {x : Carrier} {xs : List Carrier} (X : x ∈₀ xs) → X ≋₀ X
  X □₀ = ≋₀-refl
  ∈₀-subst₁-elim′ : {x y : Carrier} {xs : List Carrier} (x≈y : x ≈ y) (x∈xs : x ∈₀ xs) →
    ∈₀-subst₁ x≈y x∈xs ≋₀ x∈xs
  ∈₀-subst₁-elim′ x≈y (here sm px) = hereEq _ _ _ _
  ∈₀-subst₁-elim′ x≈y (there x∈xs) = thereEq (∈₀-subst₁-elim′ x≈y x∈xs)
```

```
∈₀-subst₁-cong′ : {x y : Carrier} {xs : List Carrier} (x≈y : x ≈ y)
                 {i j : x ∈₀ xs} → i ≋₀ j → ∈₀-subst₁ x≈y i ≋₀ ∈₀-subst₁ x≈y j
∈₀-subst₁-cong′ x≈y (hereEq px qy x≈z y≈z) = hereEq _ _ _ _ -- (sym x≈y ⟨≈≈⟩ px ) (sym x≈y ⟨≈≈⟩ qy) x≈z y≈z
∈₀-subst₁-cong′ x≈y (thereEq i≋j) = thereEq (∈₀-subst₁-cong′ x≈y i≋j)
```

29.2 Unfinished

[**WK:** *Trying — unfinished — ∈₀-subst₁-elim′′ would be sufficient for ∈₀-subst₂-cong′ — commented out:*

```
∈₀-subst₁-elim′′ : {xs ys : List Carrier} (xs≅ys : BagEq xs ys) {x x′ : Carrier} (x≈x′ : x ≈ x′) (x∈xs : x ∈₀ xs) →
    ∈₀-subst₂ xs≅ys (∈₀-subst₁ x≈x′ x∈xs) ≋₀ ∈₀-subst₂ xs≅ys x∈xs
∈₀-subst₁-elim′′ xs≅ys x≈x′ x∈xs@(here sm px) with ∈₀-subst₁ x≈x′ x∈xs | inspect (∈₀-subst₁ x≈x′) x∈xs
∈₀-subst₁-elim′′ xs≅ys x≈x′ (here sm px) | here sm₁ px₁ | [ eq ] = {!!}
∈₀-subst₁-elim′′ xs≅ys x≈x′ (here sm px) | there p | [ () ]
∈₀-subst₁-elim′′ xs≅ys x≈x′ (there p) = {!!}

∈₀-subst₂-cong′ : {x x′ : Carrier} {xs ys : List Carrier} (xs≅ys : BagEq xs ys)
                 → x ≈ x′
                 → {p : x ∈₀ xs} {q : x′ ∈₀ xs}
                 → p ≋₀ q
                 → ∈₀-subst₂ xs≅ys p ≋₀ ∈₀-subst₂ xs≅ys q
∈₀-subst₂-cong′ xs≅ys x≈x′ {p} {q} p≋₀q =
    ∈₀-subst₂ xs≅ys p
  ≋₀⟨ {!!} ⟩
    ∈₀-subst₂ xs≅ys (∈₀-subst₁ x≈x′ p)
  ≋₀⟨ ≋→≋₀ (cong (∈₀-Subst₂ xs≅ys)
      (≋₀-strengthen (
          ∈₀-subst₁ x≈x′ p
        ≋₀⟨ ∈₀-subst₁-elim′ x≈x′ p ⟩
          p
        ≋₀⟨ p≋₀q ⟩
          q
        □₀))) ⟩
    ∈₀-subst₂ xs≅ys q
  □₀

∈₀-subst₁-to : {a b : Carrier} {zs ws : List Carrier} {a≈b : a ≈ b}
    → (zs≅ws : BagEq zs ws) (a∈zs : a ∈₀ zs)
    → ∈₀-subst₁ a≈b (∈₀-subst₂ zs≅ws a∈zs) ≋ ∈₀-subst₂ zs≅ws (∈₀-subst₁ a≈b a∈zs)
∈₀-subst₁-to {a} {b} {zs} {ws} {a≈b} zs≅ws a∈zs =
  ≋₀-strengthen (
    ∈₀-subst₁ a≈b (∈₀-subst₂ zs≅ws a∈zs)
  ≋₀⟨ ∈₀-subst₁-elim′ a≈b (∈₀-subst₂ zs≅ws a∈zs) ⟩
    ∈₀-subst₂ zs≅ws a∈zs
  ≋₀˘⟨ ∈₀-subst₂-cong′ zs≅ws (sym a≈b) (∈₀-subst₁-elim′ a≈b a∈zs) ⟩
    ∈₀-subst₂ zs≅ws (∈₀-subst₁ a≈b a∈zs)
  □₀)
```

]

29.3 module NICE

∈₀-subst₂-cong′ and ∈₀-subst₁-to actually do not hold — the following module serves to provide a counterexample:

```
module NICE where
  data E : Set where
```

```
    E1 E2 E3 : E
  data _≈E_ : E → E → Set where
    ≈E-refl : {x : E} → x ≈E x
    E12 : E1 ≈E E2
    E21 : E2 ≈E E1
  ≈E-sym : {x y : E} → x ≈E y → y ≈E x
  ≈E-sym ≈E-refl = ≈E-refl
  ≈E-sym E12 = E21
  ≈E-sym E21 = E12
  ≈E-trans : {x y z : E} → x ≈E y → y ≈E z → x ≈E z
  ≈E-trans ≈E-refl ≈E-refl = ≈E-refl
  ≈E-trans ≈E-refl E12 = E12
  ≈E-trans ≈E-refl E21 = E21
  ≈E-trans E12 ≈E-refl = E12
  ≈E-trans E12 E21 = ≈E-refl
  ≈E-trans E21 ≈E-refl = E21
  ≈E-trans E21 E12 = ≈E-refl
  E-setoid : Setoid lzero lzero
  E-setoid = record
    {Carrier = E
    ; _≈_ = _≈E_
    ; isEquivalence = record
      {refl = ≈E-refl
      ; sym = ≈E-sym
      ; trans = ≈E-trans
      }
    }
  xs ys : List E
  xs = E1 :: E1 :: E3 :: []
  ys = E3 :: E1 :: E1 :: []
  open Membership E-setoid
  open HetEquiv E-setoid
  open Locations
  xs⇒ys : (x : E) → x ∈0 xs → x ∈0 ys
  xs⇒ys E1 (here sm px) = there (here sm px)
  xs⇒ys E1 (there p) = there (there (here ≈E-refl ≈E-refl))
  xs⇒ys E2 (here sm px) = there (there (here sm px))
  xs⇒ys E2 (there p) = there (here ≈E-refl E21)
  xs⇒ys E3 p = here ≈E-refl ≈E-refl
  xs⇒ys-cong : (x : E) {p p′ : x ∈0 xs} → p ≋ p′ → xs⇒ys x p ≋ xs⇒ys x p′
  xs⇒ys-cong E1 (hereEq px qy x≈z y≈z) = thereEq (hereEq _ _ _ _)
  xs⇒ys-cong E1 (thereEq e) = thereEq (thereEq (hereEq _ _ _ _))
  xs⇒ys-cong E2 (hereEq px qy x≈z y≈z) = thereEq (thereEq (hereEq _ _ _ _))
  xs⇒ys-cong E2 (thereEq e) = thereEq (hereEq _ _ _ _)
  xs⇒ys-cong E3 e = hereEq _ _ _ _
  ys⇒xs : (x : E) → x ∈0 ys → x ∈0 xs
  ys⇒xs E1 (here ≈E-refl ())
  ys⇒xs E1 (there (here sm px)) = here sm px
  ys⇒xs E1 (there (there e)) = there (here ≈E-refl ≈E-refl)
  ys⇒xs E2 (here ≈E-refl ())
  ys⇒xs E2 (there (here sm px)) = there (here E21 ≈E-refl)
  ys⇒xs E2 (there (there e)) = here ≈E-refl E21
  ys⇒xs E3 e = there (there (here ≈E-refl ≈E-refl))
  ys⇒xs-cong : (x : E) {p p′ : x ∈0 ys} → p ≋ p′ → ys⇒xs x p ≋ ys⇒xs x p′
```

```
ys⇒xs-cong E₁ (hereEq ≈E-refl ≈E-refl x≈z ())
ys⇒xs-cong E₁ (hereEq ≈E-refl E₁₂ x≈z ())
ys⇒xs-cong E₁ (hereEq E₁₂ ≈E-refl x≈z ())
ys⇒xs-cong E₁ (hereEq E₁₂ E₁₂ x≈z ())
ys⇒xs-cong E₁ (thereEq (hereEq px qy x≈z y≈z)) = hereEq px qy x≈z y≈z
ys⇒xs-cong E₁ (thereEq (thereEq eq)) = thereEq (hereEq _ _ _ _)
ys⇒xs-cong E₂ (hereEq ≈E-refl ≈E-refl x≈z ())
ys⇒xs-cong E₂ (hereEq ≈E-refl E₂₁ x≈z ())
ys⇒xs-cong E₂ (hereEq E₂₁ ≈E-refl x≈z ())
ys⇒xs-cong E₂ (hereEq E₂₁ E₂₁ x≈z ())
ys⇒xs-cong E₂ (thereEq (hereEq px qy x≈z y≈z)) = thereEq (hereEq _ _ _ _)
ys⇒xs-cong E₂ (thereEq (thereEq eq)) = hereEq _ _ _ _
ys⇒xs-cong E₃ _ = thereEq (thereEq (hereEq _ _ _ _))
leftInv : (e : E) (p : e ∈₀ xs) → ys⇒xs e (xs⇒ys e p) ≋ p
leftInv E₁ (here sm px) = hereEq px px sm sm
leftInv E₁ (there (here sm px)) = thereEq (hereEq ≈E-refl px ≈E-refl sm)
leftInv E₁ (there (there (here ≈E-refl ())))
leftInv E₁ (there (there (there ())))
leftInv E₂ (here sm px) = hereEq E₂₁ px ≈E-refl sm
leftInv E₂ (there (here sm px)) = thereEq (hereEq ≈E-refl px E₂₁ sm)
leftInv E₂ (there (there (here ≈E-refl ())))
leftInv E₂ (there (there (there ())))
leftInv E₃ (here ≈E-refl ())
leftInv E₃ (here E₂₁ ())
leftInv E₃ (there (here ≈E-refl ()))
leftInv E₃ (there (here E₂₁ ()))
leftInv E₃ (there (there (here sm px))) = thereEq (thereEq (hereEq ≈E-refl px ≈E-refl sm))
leftInv E₃ (there (there (there ())))
rightInv : (e : E) (p : e ∈₀ ys) → xs⇒ys e (ys⇒xs e p) ≋ p
rightInv E₁ (here ≈E-refl ())
rightInv E₁ (there (here sm px)) = thereEq (hereEq px px sm sm)
rightInv E₁ (there (there (here sm px))) = thereEq (thereEq (hereEq ≈E-refl px ≈E-refl sm))
rightInv E₁ (there (there (there ())))
rightInv E₂ (here ≈E-refl ())
rightInv E₂ (there (here sm px)) = thereEq (hereEq E₂₁ px ≈E-refl sm)
rightInv E₂ (there (there (here sm px))) = thereEq (thereEq (hereEq E₂₁ px ≈E-refl sm))
rightInv E₂ (there (there (there ())))
rightInv E₃ (here sm px) = hereEq ≈E-refl px ≈E-refl sm
rightInv E₃ (there (here ≈E-refl ()))
rightInv E₃ (there (here E₂₁ ()))
rightInv E₃ (there (there (here ≈E-refl ())))
rightInv E₃ (there (there (here E₂₁ ())))
rightInv E₃ (there (there (there ())))
OldBagEq : (xs ys : List E) → Set
OldBagEq xs ys = {x : E} → (x ∈ xs) ≅ (x ∈ ys)
xs≅ys : OldBagEq xs ys
xs≅ys {e} = record
  {to = record {_⟨$⟩_ = xs⇒ys e; cong = xs⇒ys-cong e}
  ; from = record {_⟨$⟩_ = ys⇒xs e; cong = ys⇒xs-cong e}
  ; inverse-of = record
    {left-inverse-of = leftInv e
    ; right-inverse-of = rightInv e
    }
  }
¬-∈₀-subst₂-cong′ : ({x x′ : E} {xs ys : List E} (xs≅ys : OldBagEq xs ys)
```

```
    → x ≈E x′
    → {p : x ∈₀ xs} {q : x′ ∈₀ xs}
    → p ≋₀ q
    → _≅_.to xs≅ys ⟨$⟩ p ≋₀ _≅_.to xs≅ys ⟨$⟩ q) → ⊥ {lzero}
¬-∈₀-subst₂-cong′ ∈₀-subst₂-cong′ with
  ∈₀-subst₂-cong′ {E₁} {E₂} {xs} {ys} xs≅ys E₁₂ {here {a = E₁} ≈E-refl ≈E-refl} {here {a = E₂} E₂₁ ≈E-refl} (hereEq _ _ _ _)
¬-∈₀-subst₂-cong′ ∈₀-subst₂-cong′ | thereEq ()

¬-∈₀-subst₁-to : ({a b : E} {zs ws : List E} {a≈b : a ≈E b}
      → (zs≅ws : OldBagEq zs ws) (a∈zs : a ∈₀ zs)
      → ∈₀-subst₁ a≈b (_≅_.to zs≅ws ⟨$⟩ a∈zs) ≋ _≅_.to zs≅ws ⟨$⟩ (∈₀-subst₁ a≈b a∈zs)
      ) → ⊥ {lzero}
¬-∈₀-subst₁-to ∈₀-subst₁-to with
  ∈₀-subst₁-to {E₁} {E₂} {xs} {ys} {E₁₂} xs≅ys (here {a = E₁} ≈E-refl ≈E-refl)
¬-∈₀-subst₁-to ∈₀-subst₁-to | thereEq ()
```

30 Conclusion and Outlook

???