Theories and Data Structures

Jacques Carette, Musa Al-hassy, Wolfram Kahl

June 15, 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`

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.

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 $\_\uplus\_$ takes types to types, its "map" variant $\_\uplus_1\_$ 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 $D : Set \to Set$ allows us to extract elements of the underlying type and so we have a natural transfomation $D \longrightarrow \mathbf{I}$, where $\mathbf{I}$ 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 $D\ A = 2 \times A = A \uplus A$:

```
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 $2 \times\_ \longrightarrow \mathbf{I}$, is diag, the transformation $\mathbf{I} \longrightarrow \_^2$.

```
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
```

6 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 DataProperties using (⊤; tt)
open import SetoidEquiv
```

Setoid of setoids SSetoid, and "setoid" of equality proofs.

```
SSetoid : (ℓ o : Level) → Setoid (lsuc o ⊍ lsuc ℓ) (o ⊍ ℓ)
SSetoid ℓ o = record
  {Carrier = Setoid ℓ o
  ; _≈_ = _≅_
  ; isEquivalence = record {refl = ≅-refl; sym = ≅-sym; trans = ≅-trans}}
```

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 $\mathsf{e}_1\ \mathsf{e}_2$: Setoid.Carrier A, then $\mathsf{e}_1 \approx_s \mathsf{e}_2$, as a type, is contractible.

```
_≈S_ : ∀ {a ℓa} {A : Setoid a ℓa} → (e₁ e₂ : Setoid.Carrier A) → Setoid ℓa ℓa
_≈S_ {A = A} e₁ e₂ = let open Setoid A renaming (_≈_ to _≈ₛ_) in
  record {Carrier = e₁ ≈ₛ e₂; _≈_ = λ _ _ → ⊤
    ; isEquivalence = record {refl = tt; sym = λ _ → tt; trans = λ _ _ → tt}}
```

7 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
```

7.1 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
```

7.2 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}
  }
```

7.3 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}
  }
```

7.4 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
  }
```

7.5 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
  }
```

7.6 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
  }
```

8 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)
```

8.1 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
```

8.2 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
  }
```

8.3 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 CoFree³ ≅ CoFree³′. 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.

8.4 ???

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
  }
```

9 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
```

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

```
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
```

9.1 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
```

9.2 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?]

9.3 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
```

10 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
```

10.1 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
```

10.2 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
```

10.3 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 step^n (base a) for a : A. This leads to an alternative presentation, Eventually A ≅ Σ n : ℕ • A viz step^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, ⟦ $_B$, $_S$ ⟧ ∘ step^n ≐ $_S{}^n$ ∘ ⟦ $_B$, $_S$ ⟧. Essentially, ⟦ $_B$, $_S$ ⟧ (step^n base x) ≈ $_S{}^n$ $_B$ 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 (stepⁿ base x) ≈ fⁿ 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.

10.4 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)
```

10.5 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: $\boxed{???}$

11 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. $\boxed{???}$

```
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
```

11.1 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
```

11.2 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
```

11.3 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-⟦⟧: $\boxed{\text{???}}$
- 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 : $\boxed{\text{???}}$
- eval-naturality : $\boxed{\text{???}}$
- as-id : $\boxed{\text{???}}$

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.

12 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
```

12.1 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
```

12.2 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}
```

12.3 Free Structure

The non-empty lists constitute a free semigroup algebra.

They can be presented as X × List X or via Σ n : ℕ • Σ xs : Vec n X • n ≢ 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 ⟦ = ⟦ f , Op S ⟧
    ⟦⟧-over-++ : {xs ys : List₁ X} → ⟦ (xs ++ ys) ≡ (⟦ xs) ⟨ S ⟩ (⟦ 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)
```

12.4 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.

12.5 Non-empty lists are trees

```
open import Structures.Magma renaming (Hom to MagmaHom)
open MagmaHom using () renaming (mor to mor_m)
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}
      → mor_m (η unit Y) ∘ mor_m F ≐ mor_m (η unit Y) ∘ mor_m G
      → mor_m F ≐ mor_m 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
      ∎
```

13 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
```

13.1 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 c : Srcs → Type becomes an argument (ss : Srcs) → X ss → X (c ss), more or less :-) to obtain a "recursion theorem" like principle. The second piece X 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 {ss : Srcs} → X ss → X (c 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.

13.2 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
```

13.3 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
```

13.4 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
```

14 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; _∘_; _⟨$⟩_)
open import Function using (_$_) renaming (id to id₀; _∘_ to _⊚_)
open import Data.List    using (List; []; _++_; _::_; map)
open import Data.Product using (∃)
open import Data.Nat     using (ℕ; zero; suc)
open import EqualityCombinators
open import DataProperties
open import SetoidEquiv
open import TypeEquiv using (swap₊)
open import SetoidSetoid
open import Relation.Binary.Sum
open import Relation.Binary.PropositionalEquality using (inspect)
```

$\boxed{\textbf{[WK:}}$ *Goal?* $\boxed{\textbf{]}}$

14.1 Some_0

Setoid based variant of Any.

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

[**WK:** *A ⟶ SSetoid _ _ is a pretty strong assumption. Logical equivalence does not ask for the two morphisms back and forth to be inverse.*] [**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.*]

```
module _ {a ℓa} {A : Setoid a ℓa} (P : A ⟶ SSetoid ℓa ℓa) where
  open Setoid A
  private P₀ = λ e → Setoid.Carrier (Π._⟨$⟩_ P e)
  data Some₀ : List Carrier → Set (a ⊔ ℓa) where
    here : {x : Carrier} {xs : List Carrier} (px : P₀ x) → Some₀ (x :: xs)
    there : {x : Carrier} {xs : List Carrier} (pxs : Some₀ xs) → Some₀ (x :: xs)
```

Inhabitants of Some_0 really are just locations: Some_0 P xs ≅ Σ i : Fin (length xs) • P (x ! i). Thus one possibility is to go with natural numbers directly, and entirely ignore the proof contained in a Some_0 P xs.

```
toℕ : {xs : List Carrier} → Some₀ xs → ℕ
toℕ (here _) = 0
toℕ (there pf) = suc (toℕ pf)
_~S_ : {xs : List Carrier} → Some₀ xs → Some₀ xs → Set
s₁ ~S s₂ = toℕ s₁ ≡ toℕ s₂
```

Instead, we choose a more direct approach: _≋_. This is an extremely strong relation: two proofs, of different properties of elements of different lists are considered related when the "witness" for the property is in the same location in both lists.

```
module _ {a ℓa} {A : Setoid a ℓa} {P : A ⟶ SSetoid ℓa ℓa} {Q : A ⟶ SSetoid ℓa ℓa} where
  open Setoid A
  private P₀ = λ e → Setoid.Carrier (Π._⟨$⟩_ P e)
  private Q₀ = λ e → Setoid.Carrier (Π._⟨$⟩_ Q e)
  infix 3 _≋_
  data _≋_ : {xs ys : List Carrier} (pf : Some₀ P xs) (pf' : Some₀ Q ys) → Set ℓa where
    hereEq : {xs ys : List Carrier} {x y : Carrier} (px : P₀ x) (qy : Q₀ y)
      → _≋_ (here {x = x} {xs} px) (here {x = y} {ys} qy)
    thereEq : {xs ys : List Carrier} {x y : Carrier} {pxs : Some₀ P xs} {qys : Some₀ Q ys}
      → _≋_ pxs qys → _≋_ (there {x = x} pxs) (there {x = y} qys)
```

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

```
module _ {a ℓa} {A : Setoid a ℓa} {P : A ⟶ SSetoid ℓa ℓa} where
  open Setoid A
  ≋-refl : {xs : List Carrier} {p : Some₀ P xs} → p ≋ p
  ≋-refl {p = here px} = hereEq px px
  ≋-refl {p = there p} = thereEq ≋-refl
module _ {a ℓa} {A : Setoid a ℓa} {P : A ⟶ SSetoid ℓa ℓa} {Q : A ⟶ SSetoid ℓa ℓa} where
  open Setoid A
  ≋-sym : {xs : List Carrier} {p : Some₀ P xs} {q : Some₀ Q xs} → p ≋ q → q ≋ p
```

```
  ≋-sym (hereEq px py) = hereEq py px
  ≋-sym (thereEq eq) = thereEq (≋-sym eq)
module _ {a ℓa} {A : Setoid a ℓa} {P : A ⟶ SSetoid ℓa ℓa} {Q : A ⟶ SSetoid ℓa ℓa} {R : A ⟶ SSetoid ℓa ℓa} where
  open Setoid A
  ≋-trans : {xs : List Carrier} {p : Some₀ P xs} {q : Some₀ Q xs} {r : Some₀ R xs}
    → p ≋ q → q ≋ r → p ≋ r
  ≋-trans (hereEq px py) (hereEq .py pz) = hereEq px pz
  ≋-trans (thereEq e) (thereEq f) = thereEq (≋-trans e f)
module _ {a ℓa} {A : Setoid a ℓa} (P : A ⟶ SSetoid ℓa ℓa) where
  open Setoid A
  private P₀ = λ e → Setoid.Carrier (Π._⟨$⟩_ P e)
  Some : List Carrier → Setoid (ℓa ⊔ a) ℓa
  Some xs = record
    {Carrier        = Some₀ P xs
    ; _≈_           = _≋_
    ; isEquivalence = record {refl = ≋-refl; sym = ≋-sym; trans = ≋-trans}
    }
≡→Some : {a ℓa : Level} {A : Setoid a ℓa} {P : A ⟶ SSetoid ℓa ℓa}
  {xs ys : List (Setoid.Carrier A)} → xs ≡ ys → Some P xs ≅ Some P ys
≡→Some {A = A} ≡.refl = ≅-refl
```

14.2 Membership module

```
module Membership {a ℓ} (S : Setoid a ℓ) where
  open Setoid S renaming (trans to _⟨≈≈⟩_)
  infix 4 _∈₀_ _∈_
```

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".

```
  setoid≈ : Carrier → S ⟶ SSetoid ℓ ℓ
  setoid≈ x = record
    {_⟨$⟩_ = λ (y : Carrier) → _≈S_ {A = S} x y
    ; cong = λ i≈j → record
      {to = record {_⟨$⟩_ = λ x≈i → x≈i ⟨≈≈⟩ i≈j; cong = λ _ → tt}
      ; from = record {_⟨$⟩_ = λ x≈j → x≈j ⟨≈≈⟩ sym i≈j; cong = λ _ → tt}
      ; inverse-of = record
        {left-inverse-of = λ _ → tt
        ; right-inverse-of = λ _ → tt
        }
      }
    }
  _∈_ : Carrier → List Carrier → Setoid (a ⊔ ℓ) ℓ
  x ∈ xs = Some (setoid≈ x) xs
  _∈₀_ : Carrier → List Carrier → Set (ℓ ⊔ a)
  x ∈₀ xs = Setoid.Carrier (x ∈ xs)
  BagEq : (xs ys : List Carrier) → Set (ℓ ⊔ a)
  BagEq xs ys = {x : Carrier} → (x ∈ xs) ≅ (x ∈ ys)
```

14.3 Parallel Composition

To avoid absurd patterns that we do not use, when using _⊎-Rel_, we make this: As such, we introduce the parallel composition of heterogeneous relations.

```
data _∥_ {a₁ b₁ c₁ a₂ b₂ c₂ : Level}
  {A₁ : Set a₁} {B₁ : Set b₁} (_~₁_ : A₁ → B₁ → Set c₁)
  {A₂ : Set a₂} {B₂ : Set b₂} (_~₂_ : A₂ → B₂ → Set c₂)
  : A₁ ⊎ A₂ → B₁ ⊎ B₂ → Set (a₁ ⊔ b₁ ⊔ c₁ ⊔ a₂ ⊔ b₂ ⊔ c₂) where
  left : {x : A₁} {y : B₁} (x~₁y : x ~₁ y) → (_~₁_ ∥ _~₂_) (inj₁ x) (inj₁ y)
  right : {x : A₂} {y : B₂} (x~₂y : x ~₂ y) → (_~₁_ ∥ _~₂_) (inj₂ x) (inj₂ y)

  -- Non-working “eliminator” for this type.
[_∥_] : {a₁ b₁ c₁ a₂ b₂ c₂ ℓ : Level}
      {A₁ : Set a₁} {B₁ : Set b₁} {_~₁_ : A₁ → B₁ → Set c₁}
      {A₂ : Set a₂} {B₂ : Set b₂} {_~₂_ : A₂ → B₂ → Set c₂}
  →
      {Z : {a : A₁ ⊎ A₂} {b : B₁ ⊎ B₂} → (_~₁_ ∥ _~₂_) a b → Set ℓ}
      (F : {a : A₁} {b : B₁} (a~b : a ~₁ b) → Z (left a~b))
      (G : {a : A₂} {b : B₂} (a~b : a ~₂ b) → Z (right a~b))
  →
      {x : A₁ ⊎ A₂} {y : B₁ ⊎ B₂}
  → (x∥y : (_~₁_ ∥ _~₂_) x y) → Z x∥y
[ F ∥ G ] (left x~y)  =  F x~y
[ F ∥ G ] (right x~y)  =  G x~y

  -- If the argument relations are symmetric then so is their parallel composition.
∥-sym : {a a′ c c′ : Level} {A : Set a} {_~_ : A → A → Set c}
  {A′ : Set a′} {_~′_ : A′ → A′ → Set c′}
  (sym₁ : {x y : A} → x ~ y → y ~ x) (sym₂ : {x y : A′} → x ~′ y → y ~′ x)
  {x y : A ⊎ A′}
  →
    (_~_ ∥ _~′_) x y → (_~_ ∥ _~′_) y x
∥-sym sym₁ sym₂ (left x~y)    =  left (sym₁ x~y)
∥-sym sym₁ sym₂ (right x~y)  =  right (sym₂ x~y)
  --
  -- ought to be just: [ left ∘ sym₁ ∥ right ∘ sym₂ ]
  --
  -- Instead, I can use, with much distasteful yellow,
  -- ∥-sym sym₁ sym₂  =  [ (λ pf → left (sym₁ pf)) ∥ (λ pf → right (sym₂ pf)) ]
infix 999 _⊎⊎_
_⊎⊎_ : {i₁ i₂ k₁ k₂ : Level} → Setoid i₁ k₁ → Setoid i₂ k₂ → Setoid (i₁ ⊔ i₂) (i₁ ⊔ i₂ ⊔ k₁ ⊔ k₂)
A ⊎⊎ B  =  record
  {Carrier  =  A₀ ⊎ B₀
  ; _≈_  =  ≈₁ ∥ ≈₂
  ; isEquivalence  =  record
    {refl    =  λ {{inj₁ x} → left refl₁; {inj₂ x} → right refl₂}
    ; sym    =  λ {(left eq) → left (sym₁ eq); (right eq) → right (sym₂ eq)}
                -- ought to be writable as [ left ∘ sym₁ ∥ right ∘ sym₂ ]
    ; trans  =  λ {(left eq) (left       eqq) → left (trans₁ eq eqq)
                ; (right eq) (right eqq) → right (trans₂ eq eqq)
                }
    }
  }
  where
    open Setoid A renaming (Carrier to A₀; _≈_ to ≈₁; refl to refl₁; sym to sym₁; trans to trans₁)
    open Setoid B renaming (Carrier to B₀; _≈_ to ≈₂; refl to refl₂; sym to sym₂; trans to trans₂)
```

14.4 ⊎⊎-comm

```
⊎⊎-comm : {a b aℓ bℓ : Level} {A : Setoid a aℓ} {B : Setoid b bℓ} → A ⊎⊎ B ≅ B ⊎⊎ A
⊎⊎-comm {A = A} {B}  =  record
```

```
  {to         = record {_⟨$⟩_ = swap₊; cong = swap-on-∥}
  ; from      = record {_⟨$⟩_ = swap₊; cong = swap-on-∥′}
  ; inverse-of = record {left-inverse-of = swap²≈∥≈id; right-inverse-of = swap²≈∥≈id′}
  }
  where
    open Setoid A renaming (Carrier to A₀; _≈_ to ≈₁; refl to refl₁)
    open Setoid B renaming (Carrier to B₀; _≈_ to ≈₂; refl to refl₂)
    swap-on-∥ : {i j : A₀ ⊎ B₀} → (≈₁ ∥ ≈₂) i j → (≈₂ ∥ ≈₁) (swap₊ i) (swap₊ j)
    swap-on-∥ (left x~₁y)   = right x~₁y
    swap-on-∥ (right x~₂y)  = left x~₂y
    swap²≈∥≈id : (z : A₀ ⊎ B₀) → (≈₁ ∥ ≈₂) (swap₊ (swap₊ z)) z
    swap²≈∥≈id (inj₁ _) = left refl₁
    swap²≈∥≈id (inj₂ _) = right refl₂
     {-Tried to obtain the following via ∥-sym ... -}
    swap-on-∥′ : {i j : B₀ ⊎ A₀} → (≈₂ ∥ ≈₁) i j → (≈₁ ∥ ≈₂) (swap₊ i) (swap₊ j)
    swap-on-∥′ (left x~y)  = right x~y
    swap-on-∥′ (right x~y) = left x~y
    swap²≈∥≈id′ : (z : B₀ ⊎ A₀) → (≈₂ ∥ ≈₁) (swap₊ (swap₊ z)) z
    swap²≈∥≈id′ (inj₁ _) = left   refl₂
    swap²≈∥≈id′ (inj₂ _) = right refl₁
```

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

```
module _ {a ℓa : Level} {A : Setoid a ℓa} {P : A ⟶ SSetoid ℓa ℓa} where
  ++≅ : {xs ys : List (Setoid.Carrier A)} → (Some P xs ⊎⊎ Some P ys) ≅ Some P (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
      _~_ = _~S_ P
      _∽_ = _≋_; ∽-refl = ≋-refl {P = P}
        -- "ealier"
      ⊎→ˡ : ∀ {ws zs} → Some₀ P ws → Some₀ P (ws + zs)
      ⊎→ˡ (here p) = here p
      ⊎→ˡ (there p) = there (⊎→ˡ p)
```

The following absurd patterns are what led me to make a new type for equalities. **["me":** *Commented out:*

```
yo : {xs : List Carrier} {x y : Some₀ P xs} → x ~ y → ⊎→ˡ x ~ ⊎→ˡ y
yo {x = here px} {here px₁} Relation.Binary.PropositionalEquality.refl = ≡.refl
yo {x = here px} {there y} ()
yo {x = there x₁} {here px} ()
yo {x = there x₁} {there y} pf = ≡.cong suc (yo {!!})
```

]

```
yo : {xs : List Carrier} {x y : Some₀ P 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 P ys → Some₀ P (xs + ys)
⊎→ʳ [] p = p
⊎→ʳ (x :: xs) p = there (⊎→ʳ xs p)

oy : (xs : List Carrier) {x y : Some₀ P 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 P zs ⊎ Some₀ P ws) → Some₀ P (zs + ws)
⊎→++ (inj₁ x) = ⊎→ˡ x
⊎→++ {zs} (inj₂ y) = ⊎→ʳ zs y

++→⊎ : ∀ xs {ys} → Some₀ P (xs + ys) → Some₀ P xs ⊎ Some₀ P 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 P xs ⊎ Some₀ P 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 P xs → Some₀ P us) (F-cong : {p q : Some₀ P xs} → p ∽ q → F p ∽ F q)
          → (G : Some₀ P ys → Some₀ P vs) (G-cong : {p q : Some₀ P ys} → p ∽ q → G p ∽ G q)
          → {pf pf' : Some₀ P xs ⊎ Some₀ P 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 P (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 P xs ⊎ Some₀ P 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 P (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
```

14.6 Bottom as a setoid

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

```
module _ {a ℓa : Level} {A : Setoid a ℓa} {P : A ⟶ SSetoid ℓa ℓa} where
  ⊥≅Some[] : ⊥⊥ {a} {ℓa} ≅ Some P []
  ⊥≅Some[] = record
    {to         = record {_⟨$⟩_ = λ {()}; cong = λ {{()}}}
    ; from       = record {_⟨$⟩_ = λ {()}; cong = λ {{()}}}
    ; inverse-of = record {left-inverse-of = λ _ → tt; right-inverse-of = λ {()}}
    }
```

14.7 map≅ : ⋯→ Some (P ∘ f) xs ≅ Some P (map (_⟨$⟩_ f) xs)

```
map≅ : ∀ {a ℓa} {A B : Setoid a ℓa} {P : B ⟶ SSetoid ℓa ℓa} {f : A ⟶ B} {xs : List (Setoid.Carrier A)} →
  Some (P ∘ f) xs ≅ Some P (map (_⟨$⟩_ f) xs)
map≅ {A = A} {B} {P} {f} = record
  {to = record {_⟨$⟩_ = map⁺; cong = map⁺-cong}
  ; from = record {_⟨$⟩_ = map⁻; cong = map⁻-cong}
  ; inverse-of = record {left-inverse-of = map⁻∘map⁺; right-inverse-of = map⁺∘map⁻}
  }
  where
  g = _⟨$⟩_ f
  A₀ = Setoid.Carrier A
  _~_ = _≋_ {P = P}
  map⁺ : {xs : List A₀} → Some₀ (P ∘ f) xs → Some₀ P (map g xs)
  map⁺ (here p) = here p
  map⁺ (there p) = there $ map⁺ p
  map⁻ : {xs : List A₀} → Some₀ P (map g xs) → Some₀ (P ∘ f) xs
  map⁻ {[]} ()
  map⁻ {x :: xs} (here p) = here p
  map⁻ {x :: xs} (there p) = there (map⁻ {xs = xs} p)
  map⁺∘map⁻ : {xs : List A₀} → (p : Some₀ P (map g xs)) → map⁺ (map⁻ p) ~ p
  map⁺∘map⁻ {[]} ()
  map⁺∘map⁻ {x :: xs} (here p) = hereEq p p
  map⁺∘map⁻ {x :: xs} (there p) = thereEq (map⁺∘map⁻ p)
  map⁻∘map⁺ : {xs : List A₀} → (p : Some₀ (P ∘ f) xs)
    → let _~₂_ = _≋_ {P = P ∘ f} in map⁻ (map⁺ p) ~₂ p
  map⁻∘map⁺ {[]} ()
  map⁻∘map⁺ {x :: xs} (here p) = hereEq p p
  map⁻∘map⁺ {x :: xs} (there p) = thereEq (map⁻∘map⁺ p)
  map⁺-cong : {ys : List A₀} {i j : Some₀ (P ∘ f) ys} → _≋_ {P = P ∘ f} i j → map⁺ i ~ map⁺ j
  map⁺-cong (hereEq px py) = hereEq px py
  map⁺-cong (thereEq i~j) = thereEq (map⁺-cong i~j)
  map⁻-cong : {ys : List A₀} {i j : Some₀ P (map g ys)} → i ~ j → _≋_ {P = P ∘ f} (map⁻ i) (map⁻ j)
  map⁻-cong {[]} ()
```

```
  map--cong {x :: ys} (hereEq px py)  =  hereEq px py
  map--cong {x :: ys} (thereEq i~j)  =  thereEq (map--cong i~j)


module FindLose {a ℓa : Level} {A : Setoid a ℓa} (P : A ⟶ SSetoid ℓa ℓa) where
open Membership A
open Setoid A
open Π
open _≅_
private
  P0  =  λ e → Setoid.Carrier (Π._⟨$⟩_ P e)
  Support  =  λ ys → Σ y : Carrier • y ∈0 ys × P0 y
find : {ys : List Carrier} → Some0 P ys → Support ys
find {y :: ys} (here p)  =  y , here refl , p
find {y :: ys} (there p)  =  let (a , a∈ys , Pa)  =  find p
                               in a , there a∈ys , Pa
lose : {ys : List Carrier} → Support ys → Some0 P ys
lose (y , here py , Py)  =  here (_≅_.to (Π.cong P py) Π.⟨$⟩ Py)
lose (y , there y∈ys , Py)  =  there (lose (y , y∈ys , Py))
  -- "If an element of ys has a property P, then some element of ys has property P"
  -- cf copy below
Some-Intro : {y : Carrier} {ys : List Carrier}
  → y ∈0 ys → P0 y → Some0 P ys
Some-Intro {y} y∈ys Qy  =  lose (y , y∈ys , Qy)
bag-as-⇒ : {xs ys : List Carrier} → BagEq xs ys → Some0 P xs → Some0 P ys
bag-as-⇒ xs≅ys Pxs  =  let (x , x∈xs , Px)  =  find Pxs in
  let x∈ys  =  to xs≅ys ⟨$⟩ x∈xs
  in lose (x , x∈ys , Px)
module FindLoseCong {a ℓa : Level} {A : Setoid a ℓa} {P : A ⟶ SSetoid ℓa ℓa} {Q : A ⟶ SSetoid ℓa ℓa} where
open Membership A
open Setoid A
private
  P0  =  λ e → Setoid.Carrier (Π._⟨$⟩_ P e)
  Q0  =  λ e → Setoid.Carrier (Π._⟨$⟩_ Q e)
  PSupport  =  λ ys → Σ y : Carrier • y ∈0 ys × P0 y
  QSupport  =  λ ys → Σ y : Carrier • y ∈0 ys × Q0 y
_≁_ : {xs ys : List Carrier} → PSupport xs → QSupport ys → Set ℓa
(a , a∈xs , Pa) ≁ (b , b∈ys , Qb)  =  a ≈ b × a∈xs ≋ b∈ys
open FindLose
find-cong : {ys : List Carrier} {p : Some0 P ys} {q : Some0 Q ys} → p ≋ q → find P p ≁ find Q q
find-cong (hereEq px qy)  =  refl , ≋-refl
find-cong (thereEq eq)  =  let (fst , snd)  =  find-cong eq in fst , thereEq snd
private
  P+ : {x y : Carrier} → x ≈ y → P0 x → P0 y
  P+ x≈y  =  Π._⟨$⟩_ (_≅_.to (Π.cong P x≈y))
  Q+ : {x y : Carrier} → x ≈ y → Q0 x → Q0 y
  Q+ x≈y  =  Π._⟨$⟩_ (_≅_.to (Π.cong Q x≈y))
lose-cong : {xs ys : List Carrier} {p : PSupport xs} {q : QSupport ys} → p ≁ q → lose P p ≋ lose Q q
lose-cong {p  =  a , here a≈x , Pa} {b , here b≈x , Qb} (fst , hereEq .a≈x .b≈x)  =  hereEq (P+ a≈x Pa) (Q+ b≈x Qb)
lose-cong {p  =  a , here a≈x , Pa} {b , there b∈ys , Qb} (fst , ())
lose-cong {p  =  a , there a∈xs , Pa} {b , here px , Qb} (fst , ())
lose-cong {p  =  a , there a∈xs , Pa} {b , there b∈ys , Qb} (a≈b , thereEq a∈xs≋b∈ys)  =  thereEq (lose-cong (a≈b , a∈xs≋b∈ys))
cong-fwd : {xs ys : List Carrier} {xs≅ys : BagEq xs ys} {p : Some0 P xs} {q : Some0 Q xs}
  → p ≋ q → bag-as-⇒ P xs≅ys p ≋ bag-as-⇒ Q xs≅ys q
```

```
cong-fwd {xs} {ys} {xs≅ys} {p} {q} p≋q with find P p | find Q q | find-cong p≋q
...| (x , x∈xs , px) | (y , y∈xs , py) | (x≈y , x∈xs≋y∈xs) = lose-cong (x≈y , goal)
  where
    open _≅_ (xs≅ys {x}) using () renaming (to to F)
    open _≅_ (xs≅ys {y}) using () renaming (to to G)
    F-cong : {a b : x ∈₀ xs} → a ≋ b → F ⟨$⟩ a ≋ F ⟨$⟩ b
    F-cong = Π.cong F
    G-cong : {a b : y ∈₀ xs} → a ≋ b → G ⟨$⟩ a ≋ G ⟨$⟩ b
    G-cong = Π.cong G
    To = λ {i} → Π._⟨$⟩_ (_≅_.to (xs≅ys {i}))
    postulate helper : {i j : Carrier} → i ≈ j → {!To {i} ≐ To {j} !}
      -- switch to john major equality in the defn of ≐ ?
    goal : F ⟨$⟩ x∈xs ≋ G ⟨$⟩ y∈xs
    goal = {!Π.cong F!}
    y∈ysT : y ∈₀ xs
    y∈ysT = y∈xs
```

[Somebody: *Commented out:*

```
bag-as-⇒ : {xs ys : List Carrier} → BagEq xs ys → Some₀ P xs → Some₀ P ys
bag-as-⇒ xs≅ys Pxs = let (x , x∈xs , Px) = find Pxs in
  let x∈ys = to xs≅ys ⟨$⟩ x∈xs
  in lose (x , x∈ys , Px)
```

]

14.8 Some-cong and holes

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

```
module _ {a ℓa : Level} {A : Setoid a ℓa} {P : A ⟶ SSetoid ℓa ℓa} where
open Membership A
open Setoid A
private
  P₀ = λ e → Setoid.Carrier (Π._⟨$⟩_ P e)
  Support = λ ys → Σ y : Carrier • y ∈₀ ys × P₀ y
_≉_ : {ys : List Carrier} → Support ys → Support ys → Set ℓa
(a , a∈xs , Pa) ≉ (b , b∈xs , Pb) = a ≈ b × a∈xs ≋ b∈xs
Σ-Setoid : (ys : List Carrier) → Setoid (ℓa ⊔ a) ℓa
Σ-Setoid ys = record
  {Carrier = Support 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 {a , a∈xs , Pa} = refl , ≋-refl
    Sym : Symmetric _≉_
    Sym (a≈b , a∈xs≋b∈xs) = sym a≈b , ≋-sym a∈xs≋b∈xs
```

```
    Trans : Transitive _≋_
    Trans (a≈b , a∈xs≋b∈xs) (b≈c , b∈xs≋c∈xs) = trans a≈b b≈c , ≋-trans a∈xs≋b∈xs b∈xs≋c∈xs
module ≋ {ys} where open Setoid (Σ-Setoid ys) public
open FindLose P
open FindLoseCong hiding (_≋_)
left-inv : {ys : List Carrier} (x∈ys : Some₀ P ys) → lose (find x∈ys) ≋ x∈ys
left-inv (here px) = hereEq _ px
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 px , Py) = (sym px) , (hereEq refl px)
right-inv (y , there y∈ys , Py) = (proj₁ (right-inv (y , y∈ys , Py))) , (thereEq (proj₂ (right-inv (y , y∈ys , Py))))
Σ-Some : (xs : List Carrier) → Some P xs ≅ Σ-Setoid xs
Σ-Some xs = record
  {to = record {_⟨$⟩_ = find {xs}; cong = find-cong}
  ; from = record {_⟨$⟩_ = lose; cong = lose-cong}
  ; inverse-of = record
    {left-inverse-of = left-inv
    ; right-inverse-of = right-inv
    }
  }


module _ {a ℓa : Level} {A : Setoid a ℓa} {P : A ⟶ SSetoid ℓa ℓa} where
open Membership A
open Setoid A
private P₀ = λ e → Setoid.Carrier (Π._⟨$⟩_ P e)
Some-cong : {xs₁ xs₂ : List Carrier} →
  (∀ {x} → (x ∈ xs₁) ≅ (x ∈ xs₂)) →
  Some P xs₁ ≅ Some P xs₂
Some-cong {xs₁} {xs₂} list-rel = record
  {to          = record
    {_⟨$⟩_ = bag-as-⇒ list-rel
    ; cong = FindLoseCong.cong-fwd {P = P} {Q = P} {xs≅ys = list-rel}
    }
  ; from        = record {_⟨$⟩_ = xs₁→xs₂ (≅-sym list-rel); cong = {! {- ??? -} !}}
  ; inverse-of = record {left-inverse-of = {! {- ??? -} !}; right-inverse-of = {! {- ??? -} !}}
  }
  where
  open FindLose P using (bag-as-⇒; find)
    -- this is probably a specialized version of Respects.
    -- is also related to an uncurried version of lose.
  copy : ∀ {x} {ys} {Q : A ⟶ SSetoid ℓa ℓa} → x ∈₀ ys → (Setoid.Carrier (Π._⟨$⟩_ Q x)) → Some₀ Q ys
  copy {Q = Q} (here p) pf = here (_≅_.to (Π.cong Q p) ⟨$⟩ pf)
  copy (there p) pf = there (copy p pf)
    -- [ Somebody: this should be generalized to qy coming from Q₀ x. ]
  copy-cong : {x y : Carrier} {xs ys : List Carrier} {Q : A ⟶ SSetoid ℓa ℓa}
    (px : P₀ x) (qy : Setoid.Carrier (Π._⟨$⟩_ Q y)) (x∈xs : x ∈₀ xs) (y∈ys : y ∈₀ ys) →
    (x∈xs ≋ y∈ys) → copy {Q = P} x∈xs px ≋ copy {Q = Q} y∈ys qy
  copy-cong px qy₁ (here px₁) ∘ (here qy) (hereEq .px₁ qy) = hereEq _ _
  copy-cong px qy (there i) ∘ (there _) (thereEq i≋j) = thereEq (copy-cong px qy _ _ i≋j)
  xs₁→xs₂ : ∀ {xs ys} → (∀ {x} → (x ∈ xs) ≅ (x ∈ ys)) → Some₀ P xs → Some₀ P ys
  xs₁→xs₂ {xs} rel p =
    let pos = find {ys = xs} p in
```

```
    copy (_≅_.to rel ⟨$⟩ proj₁ (proj₂ pos)) (proj₂ (proj₂ pos))
  cong-fwd : {i j : Some₀ P xs₁} →
    i ≈ j → xs₁→xs₂ list-rel i ≈ xs₁→xs₂ list-rel j
  cong-fwd {i} {j} i≈j = copy-cong _ _ _ _ {! {- ??? -} !}
```

15 Conclusion and Outlook

???