

Theories and Data Structures

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

Jacques Carette, Musa Al-hassy, Wolfram Kahl

June 24, 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 explicitly 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?

???

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

Contents

1	Introduction	4
2	Overview	4
3	Obtaining Forgetful Functors	4
4	Equality Combinators	5
4.1	Propositional Equality	5
4.2	Function Extensionality	6
4.3	Equiv	6
4.4	Making <code>symmetry</code> calls less intrusive	7
5	Properties of Sums and Products	7
5.1	Generalised Bot and Top	7
5.2	Sums	8
5.3	Products	8
6	SetoidSetoid	9
7	Two Sorted Structures	9
7.1	Definitions	10
7.2	Category and Forgetful Functors	10
7.3	Free and CoFree	11
7.4	Adjunction Proofs	12
7.5	Merging is adjoint to duplication	13
7.6	Duplication also has a left adjoint	13
8	Binary Heterogeneous Relations — [MA:] <i>What named data structure do these correspond to in programming?</i> 1	14
8.1	Definitions	14
8.2	Category and Forgetful Functors	15
8.3	Free and CoFree Functors	15
8.4	???	19
9	Pointed Algebras: Nullable Types	20
9.1	Definition	21
9.2	Category and Forgetful Functors	21
9.3	A Free Construction	22
10	UnaryAlgebra	23
10.1	Definition	23
10.2	Category and Forgetful Functor	23

10.3 Free Structure	24
10.4 The Toolki Appears Naturally: Part 1	25
10.5 The Toolki Appears Naturally: Part 2	26
11 Magmas: Binary Trees	27
11.1 Definition	27
11.2 Category and Forgetful Functor	28
11.3 Syntax	28
12 Semigroups: Non-empty Lists	30
12.1 Definition	30
12.2 Category and Forgetful Functor	30
12.3 Free Structure	31
12.4 Adjunction Proof	32
12.5 Non-empty lists are trees	33
13 Monoids: Lists	34
13.1 Some remarks about recursion principles	35
13.2 Definition	35
13.3 Category	35
13.4 Forgetful Functors ???	36
14 Involutive Algebras: Sum and Product Types	36
14.1 Definition	36
14.2 Category and Forgetful Functor	37
14.3 Free Adjunction: Part 1 of a toolkit	37
14.4 CoFree Adjunction	39
14.5 Monad constructions	40
15 Some	40
15.1 Some_0	41
15.2 Membership module	42
15.3 $++\cong : \dots \rightarrow (\text{Some } P \text{ } xs \uplus \text{Some } P \text{ } ys) \cong \text{Some } P \text{ } (xs + ys)$	46
15.4 Bottom as a setoid	47
15.5 $\text{map}\cong : \dots \rightarrow \text{Some } (P \circ f) \text{ } xs \cong \text{Some } P \text{ } (\text{map } (_ \$) _ f) \text{ } xs)$	47
15.6 FindLose	48
15.7 Σ -Setoid	49
15.8 Some-cong	50
16 Conclusion and Outlook	51

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 by 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 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 constitute 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-o : {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
  ; _o_      = comp
  ; assoc    = λ {A B C D} {F} {G} {H} → begin⟨ ≐-setoid (Carrier A) (Carrier D) ⟩
    mor (comp (comp H G) F) ≈⟨ comp-is-o ⟩
    mor (comp H G) o mor F   ≈⟨ o-≐-cong1 _ comp-is-o ⟩
    mor H o mor G o mor F    ≈⟨ o-≐-cong2 (mor H) comp-is-o ⟩
    mor H o mor (comp G F)   ≈⟨ comp-is-o ⟩
    mor (comp H (comp G F)) ■
  ; identityl = λ {f = f} → comp-is-o ⟨ ≐ ⟩ Id-is-id o mor f
  ; identityr = λ {f = f} → comp-is-o ⟨ ≐ ⟩ ≡.cong (mor f) o Id-is-id
  ; equiv     = record { IsEquivalence ≐-isEquivalence }
  ; o-resp-≡  = λ f≈h g≈k → comp-is-o ⟨ ≐ ⟩ o-resp-≐ f≈h g≈k ⟨ ≐ ⟩ ≐-sym comp-is-o
  }
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-o $i
  ; F-resp-≡  = _$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, $_ \doteq _$, 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 (o-resp-≐) -- To do: port this over here!
  renaming (cong o to o-≐-cong2; cong o to o-≐-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 transitional 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 $_ \sqcup _$, which is rather odd since infix operators ought to have higher precedence than equality combinators, yet the standard library assigns $_ \approx \langle _ \rangle _$ 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
```

\perp -elim ()

record $\top \{ \ell : \text{Level} \} : \text{Set } \ell$ **where**
 constructor tt

5.2 Sums

Just as $_ \sqcup _$ takes types to types, its “map” variant $_ \sqcup_1 _$ takes functions to functions and is a functorial congruence: It preserves identity, distributes over composition, and preserves extensional equality.

$\sqcup\text{-id} : \{a\ b : \text{Level}\} \{A : \text{Set } a\} \{B : \text{Set } b\} \rightarrow \text{id} \sqcup_1 \text{id} \doteq \text{id} \{A = A \sqcup B\}$
 $\sqcup\text{-id} = [\doteq\text{-refl} , \doteq\text{-refl}]$
 $\sqcup\text{-o} : \{a\ b\ c\ a' \ b' \ c' : \text{Level}\}$
 $\{A : \text{Set } a\} \{A' : \text{Set } a'\} \{B : \text{Set } b\} \{B' : \text{Set } b'\} \{C : \text{Set } c\} \{C' : \text{Set } c'\}$
 $\{f : A \rightarrow A'\} \{g : B \rightarrow B'\} \{f' : A' \rightarrow C\} \{g' : B' \rightarrow C'\}$
 $\rightarrow (f' \circ f) \sqcup_1 (g' \circ g) \doteq (f' \sqcup_1 g') \circ (f \sqcup_1 g) \quad \text{-- aka “the exchange rule for sums”}$
 $\sqcup\text{-o} = [\doteq\text{-refl} , \doteq\text{-refl}]$
 $\sqcup\text{-cong} : \{a\ b\ c\ d : \text{Level}\} \{A : \text{Set } a\} \{B : \text{Set } b\} \{C : \text{Set } c\} \{D : \text{Set } d\} \{f\ f' : A \rightarrow C\} \{g\ g' : B \rightarrow D\}$
 $\rightarrow f \doteq f' \rightarrow g \doteq g' \rightarrow f \sqcup_1 g \doteq f' \sqcup_1 g'$
 $\sqcup\text{-cong } f \approx f' \ g \approx g' = [\circ\text{-}\doteq\text{-cong}_2 \text{ inj}_1 \ f \approx f' , \circ\text{-}\doteq\text{-cong}_2 \text{ inj}_2 \ g \approx g']$

Composition post-distributes into casing,

$\circ\text{-}[.] : \{a\ b\ c\ d : \text{Level}\} \{A : \text{Set } a\} \{B : \text{Set } b\} \{C : \text{Set } c\} \{D : \text{Set } d\} \{f : A \rightarrow C\} \{g : B \rightarrow C\} \{h : C \rightarrow D\}$
 $\rightarrow h \circ [f , g] \doteq [h \circ f , h \circ g] \quad \text{-- aka “fusion”}$
 $\circ\text{-}[.] = [\doteq\text{-refl} , \doteq\text{-refl}]$

It is common that a data-type constructor $D : \text{Set} \rightarrow \text{Set}$ allows us to extract elements of the underlying type and so we have a natural transformation $D \rightarrow \mathbf{I}$, where \mathbf{I} is the identity functor. These kind of results will occur for our other simple data-structures as well. In particular, this is the case for $D\ A = 2 \times A = A \sqcup A$:

$\text{from}\sqcup : \{ \ell : \text{Level} \} \{ A : \text{Set } \ell \} \rightarrow A \sqcup A \rightarrow A$
 $\text{from}\sqcup = [\text{id} , \text{id}]$
 -- from \sqcup is a natural transformation
 --
 $\text{from}\sqcup\text{-nat} : \{a\ b : \text{Level}\} \{A : \text{Set } a\} \{B : \text{Set } b\} \{f : A \rightarrow B\} \rightarrow f \circ \text{from}\sqcup \doteq \text{from}\sqcup \circ (f \sqcup_1 f)$
 $\text{from}\sqcup\text{-nat} = [\doteq\text{-refl} , \doteq\text{-refl}]$
 -- from \sqcup is injective and so is pre-invertible,
 --
 $\text{from}\sqcup\text{-preInverse} : \{a\ b : \text{Level}\} \{A : \text{Set } a\} \{B : \text{Set } b\} \rightarrow \text{id} \doteq \text{from}\sqcup \{A = A \sqcup B\} \circ (\text{inj}_1 \sqcup_1 \text{inj}_2)$
 $\text{from}\sqcup\text{-preInverse} = [\doteq\text{-refl} , \doteq\text{-refl}]$

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

5.3 Products

Dual to $\text{from}\sqcup$, a natural transformation $2 \times _ \rightarrow \mathbf{I}$, is diag , the transformation $\mathbf{I} \rightarrow _ ^2$.

$\text{diag} : \{ \ell : \text{Level} \} \{ A : \text{Set } \ell \} (a : A) \rightarrow A \times A$
 $\text{diag } a = a , a$

[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 Function.Equivalence using (Equivalence; id; _◦_; sym)
open import Function using (flip)
open import DataProperties using (T; tt)
open import SetoidEquiv

```

Setoid of setoids SSetoid, and “setoid” of equality proofs.

```

SSetoid : (o ℓ : Level) → Setoid (lsuc ℓ ⊔ lsuc o) (ℓ ⊔ o)
SSetoid o ℓ = record
  { Carrier = Setoid ℓ o
  ; _≈_ = 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 $e_1 e_2 : \text{Setoid.Carrier } A$, then $e_1 \approx_s e_2$, 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 _ _ → T
  ; 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;  $\_ \circ \_$ ; 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  $\ell$  : Set (lsuc  $\ell$ ) where
  constructor MkTwo
  field
    One : Set  $\ell$ 
    Two : Set  $\ell$ 
open TwoSorted

```

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

```

record Hom  $\{\ell\}$  (Src Tgt : TwoSorted  $\ell$ ) : Set  $\ell$  where
  constructor MkHom
  field
    one : One Src  $\rightarrow$  One Tgt
    two : Two Src  $\rightarrow$  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 : ( $\ell$  : Level)  $\rightarrow$  Category (lsuc  $\ell$ )  $\ell$   $\ell$ 
Twos  $\ell$  = record
  { Obj      = TwoSorted  $\ell$ 
  ;  $\_ \Rightarrow \_$  = Hom
  ;  $\_ \equiv \_$     =  $\lambda$  F G  $\rightarrow$  one F  $\doteq$  one G  $\times$  two F  $\doteq$  two G
  ; id        = MkHom id id
  ;  $\_ \circ \_$       =  $\lambda$  F G  $\rightarrow$  MkHom (one F  $\circ$  one G) (two F  $\circ$  two G)
  ; assoc     =  $\doteq$ -refl ,  $\doteq$ -refl
  ; identityl =  $\doteq$ -refl ,  $\doteq$ -refl
  ; identityr =  $\doteq$ -refl ,  $\doteq$ -refl
  ; equiv     = record
    { refl    =  $\doteq$ -refl ,  $\doteq$ -refl
    ; sym     =  $\lambda$  {(oneEq , twoEq)  $\rightarrow$   $\doteq$ -sym oneEq ,  $\doteq$ -sym twoEq}
    ; trans   =  $\lambda$  {(oneEq1 , twoEq1) (oneEq2 , twoEq2)  $\rightarrow$   $\doteq$ -trans oneEq1 oneEq2 ,  $\doteq$ -trans twoEq1 twoEq2}
    }
  ; o-resp $\equiv$  =  $\lambda$  {(g1 k , g2 k) (f1 h , f2 h)  $\rightarrow$  o-resp $\doteq$  g1 k f1 h , o-resp $\doteq$  g2 k f2 h}
  }

```

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 singleton type and so we have a free and a co-free constructions. Intuitively, the first is free since the singleton 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 naturally 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 represent 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) → ⊔-o 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.

$\text{Rels} : (\ell \ell' : \text{Level}) \rightarrow \text{Category} (\text{Isuc} (\ell \sqcup \ell')) (\ell \sqcup \ell') \ell$

$\text{Rels } \ell \ell' = \text{record}$

```
{Obj      = HetRel 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}
  }
; o-resp≡ = λ {(g≈1k, g≈2k) (f≈1h, f≈2h) → o-resp≐ g≈1k f≈1h, o-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 :-)

$\text{Forget}^1 : (\ell \ell' : \text{Level}) \rightarrow \text{Functor} (\text{Rels } \ell \ell') (\text{Sets } \ell)$

$\text{Forget}^1 \ell \ell' = \text{record}$

```
{F0      = HetRel.One
; F1      = Hom.one
; identity = ≡.refl
; homomorphism = ≡.refl
; F-resp≡ = λ {(F≈1G, F≈2G) {x} → F≈1G x}
}
```

$\text{Forget}^2 : (\ell \ell' : \text{Level}) \rightarrow \text{Functor} (\text{Rels } \ell \ell') (\text{Sets } \ell)$

$\text{Forget}^2 \ell \ell' = \text{record}$

```
{F0      = HetRel.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

$\text{Forget}^3 : (\ell \ell' : \text{Level}) \rightarrow \text{Functor} (\text{Rels } \ell \ell') (\text{Twos } \ell)$

$\text{Forget}^3 \ell \ell' = \text{record}$

```
{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

$\text{Free}^1 : (\ell \ell' : \text{Level}) \rightarrow \text{Functor} (\text{Sets } \ell) (\text{Rels } \ell \ell')$

$\text{Free}^1 \ell \ell' = \text{record}$
 $\{F_0 = \lambda A \rightarrow \text{MkHRel } A \perp (\lambda \{ _ \} \{ _ \})\}$
 $; F_1 = \lambda f \rightarrow \text{MkHom } f \text{ id } (\lambda \{ \{y = _ \} \})\}$
 $; \text{identity} = \dot{=} \text{-refl}, \dot{=} \text{-refl}$
 $; \text{homomorphism} = \dot{=} \text{-refl}, \dot{=} \text{-refl}$
 $; F\text{-resp} \equiv = \lambda f \approx g \rightarrow (\lambda x \rightarrow f \approx g \{x\}), \dot{=} \text{-refl}$
 $\}$
 $-- (\text{MkRel } X \perp \perp \longrightarrow \text{Alg}) \cong (X \longrightarrow \text{One Alg})$

$\text{Left}^1 : (\ell \ell' : \text{Level}) \rightarrow \text{Adjunction} (\text{Free}^1 \ell \ell') (\text{Forget}^1 \ell \ell')$

$\text{Left}^1 \ell \ell' = \text{record}$
 $\{\text{unit} = \text{record}$
 $\{\eta = \lambda _ \rightarrow \text{id}$
 $; \text{commute} = \lambda _ \rightarrow \equiv \text{-refl}$
 $\}$
 $; \text{counit} = \text{record}$
 $\{\eta = \lambda A \rightarrow \text{MkHom} (\lambda z \rightarrow z) (\lambda \{ \{ \} \}) (\lambda \{x\} \{ \})\}$
 $; \text{commute} = \lambda f \rightarrow \dot{=} \text{-refl}, (\lambda \{ \})\}$
 $\}$
 $; \text{zig} = \dot{=} \text{-refl}, (\lambda \{ \})\}$
 $; \text{zag} = \equiv \text{-refl}$
 $\}$

$\text{CoFree}^1 : (\ell : \text{Level}) \rightarrow \text{Functor} (\text{Sets } \ell) (\text{Rels } \ell \ell)$

$\text{CoFree}^1 \ell = \text{record}$
 $\{F_0 = \lambda A \rightarrow \text{MkHRel } A \top (\lambda _ _ \rightarrow A)\}$
 $; F_1 = \lambda f \rightarrow \text{MkHom } f \text{ id } f$
 $; \text{identity} = \dot{=} \text{-refl}, \dot{=} \text{-refl}$
 $; \text{homomorphism} = \dot{=} \text{-refl}, \dot{=} \text{-refl}$
 $; F\text{-resp} \equiv = \lambda f \approx g \rightarrow (\lambda x \rightarrow f \approx g \{x\}), \dot{=} \text{-refl}$
 $\}$

$-- (\text{One Alg} \longrightarrow X) \cong (\text{Alg} \longrightarrow \text{MkRel } X \top (\lambda _ _ \rightarrow X))$

$\text{Right}^1 : (\ell : \text{Level}) \rightarrow \text{Adjunction} (\text{Forget}^1 \ell \ell) (\text{CoFree}^1 \ell)$

$\text{Right}^1 \ell = \text{record}$
 $\{\text{unit} = \text{record}$
 $\{\eta = \lambda _ \rightarrow \text{MkHom id } (\lambda _ \rightarrow \text{tt}) (\lambda \{x\} \{y\} _ \rightarrow x)\}$
 $; \text{commute} = \lambda _ \rightarrow \dot{=} \text{-refl}, (\lambda x \rightarrow \equiv \text{-refl})\}$
 $\}$
 $; \text{counit} = \text{record } \{\eta = \lambda _ \rightarrow \text{id}; \text{commute} = \lambda _ \rightarrow \equiv \text{-refl}\}$
 $; \text{zig} = \equiv \text{-refl}$
 $; \text{zag} = \dot{=} \text{-refl}, \lambda \{ \text{tt} \rightarrow \equiv \text{-refl} \}$
 $\}$

$-- \text{Another cofree functor:}$

$\text{CoFree}^{1'} : (\ell : \text{Level}) \rightarrow \text{Functor} (\text{Sets } \ell) (\text{Rels } \ell \ell)$

$\text{CoFree}^{1'} \ell = \text{record}$
 $\{F_0 = \lambda A \rightarrow \text{MkHRel } A \top (\lambda _ _ \rightarrow \top)\}$
 $; F_1 = \lambda f \rightarrow \text{MkHom } f \text{ id id}$
 $; \text{identity} = \dot{=} \text{-refl}, \dot{=} \text{-refl}$
 $; \text{homomorphism} = \dot{=} \text{-refl}, \dot{=} \text{-refl}$


```

;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 $\text{CoFree}^1 \cong \text{Cofree}^{1'}$. 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 $\text{CoFree}^3 \cong \text{CoFree}^{3'}$. 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≈1 G , F≈2 G) {x , y} → ≡.cong2 _ , _ (F≈1 G x) (F≈2 G 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 represent a `TwoSorted` algebra as a sum, and moreover this is adjoint to the aforementioned duplication functor.

```

Choice : (ℓ : Level) → Functor (TwoSorted 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} → ⊔-o 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**

¹Note 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;  $\_ \circ \_$ )
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 (Isuc 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 algebras 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 (Isuc 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 $\text{step}^n (\text{base } a)$ for $a : A$. This leads to an alternative presentation, $\text{Eventually } A \cong \sum n : \mathbb{N} \bullet A$ viz $\text{step}^n (\text{base } a) \leftrightarrow (n, a) \text{ —cf } \text{Free}^2 \text{ below. Incidentally, or promisingly, } \text{Eventually } \top \cong \mathbb{N}.$

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, ς) we can interpret the terms of $\text{Eventually } A$ where the injection base is reified by B and the unary operation step is reified by ς .

```

open import Function using (const)
[_,_] : {a b : Level} {A : Set a} {B : Set b} (B : A → B) (ϑ : B → B) → Eventually A → B
[ B , ϑ ] = elim (λ {a} → B a) ϑ

```

Notice that: The number of ς steps is preserved, $[B , \varsigma] \circ \text{step}^n \doteq \varsigma^n \circ [B , \varsigma]$. Essentially, $[B , \varsigma] (\text{step}^n \text{base } x) \approx \varsigma^n B x$. 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-ο**: 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 $\text{step}^n \text{base } x \mapsto f^n x$. 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 ο 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) ο f ≐ f ο (f ↑ n)
iter-swap {n = zero} = ≐-refl
iter-swap {f = f} {n = suc n} = ο-≐-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 ο g ≐i h ο f)
  → {n : ℕ} → h ↑ n ο f ≐i f ο g ↑ n
iter-comm leap {zero} = ≡.refl
iter-comm {g = g} {h} leap {suc n} = ≡.cong (h ↑ n) (≡.sym leap) (≡≡) iter-comm leap

-- exponentiation 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} ο (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 ο at 0

```

```

ziggy zero = ≐-refl
ziggy {A = A} (suc n) = begin⟨ ≐-setoid A (ℕ × A) ⟩
  (suc ×1 id) ∘ at n ≈⟨ o-≐-cong2 (suc ×1 id) (ziggy n) ⟩
  (suc ×1 id) ∘ (suc ×1 id {A = A}) ↑ n ∘ at 0 ≈⟨ o-≐-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: ???
- $_ \wedge _$: ???
- iter-swap: ???
- ziggy: ???

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. ???

```

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 (Isuc 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-[]`: ???
- `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` : ???
- `eval-naturality` : ???
- `as-id` : ???

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 (Isuc 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 $X \times \text{List } X$ or via $\sum n : \mathbb{N} \bullet \sum xs : \text{Vec } n \ X \bullet n \neq 0$. 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 List_1 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)

```

$$\begin{aligned} \llbracket - \text{over-} ++ \{xs\} \{ys\} \rrbracket &= \text{rec } \{X = \lambda xs \rightarrow_H (xs + ys) \equiv ({}_H xs) \langle S \rangle ({}_H ys)\} \\ &\quad \doteq \text{-refl } (\lambda x \times xs' \text{ ind} \rightarrow \equiv.\text{cong } (\text{Op } S (f x)) \text{ ind } \langle \equiv \rangle \text{ assoc } S) xs \end{aligned}$$

In particular, the map operation over lists is:

$$\begin{aligned} \text{map} &: \{a \ b : \text{Level}\} \{A : \text{Set } a\} \{B : \text{Set } b\} \rightarrow (A \rightarrow B) \rightarrow \text{List}_1 A \rightarrow \text{List}_1 B \\ \text{map } f &= \llbracket [-] \circ f, _ ++ _ \rrbracket \end{aligned}$$

At the dependent level, we have the induction principle,

$$\begin{aligned} \text{ind} &: \{a \ b : \text{Level}\} \{A : \text{Set } a\} \{P : \text{List}_1 A \rightarrow \text{Set } b\} \\ &\quad \rightarrow (\text{base} : \{x : A\} \rightarrow P \llbracket x \rrbracket) \\ &\quad \rightarrow (\text{ind} : \{x : A\} \{xs : \text{List}_1 A\} \rightarrow P \llbracket x \rrbracket \rightarrow P \text{ xs} \rightarrow P (x :: xs)) \\ &\quad \rightarrow (xs : \text{List}_1 A) \rightarrow P \text{ xs} \\ \text{ind base ind} &= \text{rec } (\lambda y \rightarrow \text{base}) (\lambda y \text{ ys} \rightarrow \text{ind base}) \\ -- \text{ind } \{P = P\} \text{ base ind } \llbracket x \rrbracket &= \text{base} \\ -- \text{ind } \{P = P\} \text{ base ind } (x :: xs) &= \text{ind } \{x\} \{xs\} (\text{base } \{x\}) (\text{ind } \{P = P\} \text{ base ind } xs) \end{aligned}$$

For example, map preserves identity:

$$\begin{aligned} \text{map-id} &: \{a : \text{Level}\} \{A : \text{Set } a\} \rightarrow \text{map id} \doteq \text{id } \{A = \text{List}_1 A\} \\ \text{map-id} &= \text{ind } \equiv.\text{refl } (\lambda \{x\} \{xs\} \text{ refl ind} \rightarrow \equiv.\text{cong } (x :: _) \text{ ind}) \\ \text{map-}\circ &: \{\ell : \text{Level}\} \{A \ B \ C : \text{Set } \ell\} \{f : A \rightarrow B\} \{g : B \rightarrow C\} \\ &\quad \rightarrow \text{map } (g \circ f) \doteq \text{map } g \circ \text{map } f \\ \text{map-}\circ \{f = f\} \{g\} &= \text{ind } \equiv.\text{refl } (\lambda \{x\} \{xs\} \text{ refl ind} \rightarrow \equiv.\text{cong } ((g (f x)) :: _) \text{ ind}) \\ \text{map-cong} &: \{\ell : \text{Level}\} \{A \ B : \text{Set } \ell\} \{f \ g : A \rightarrow B\} \\ &\quad \rightarrow f \doteq g \rightarrow \text{map } f \doteq \text{map } g \\ \text{map-cong } \{f = f\} \{g\} f \doteq g &= \text{ind } (\equiv.\text{cong } [-] (f \doteq g _)) \\ &\quad (\lambda \{x\} \{xs\} \text{ refl ind} \rightarrow \equiv.\text{cong}_2 _ :: _ (f \doteq g x) \text{ ind}) \end{aligned}$$

12.4 Adjunction Proof

$\text{Free} : (\ell : \text{Level}) \rightarrow \text{Functor } (\text{Sets } \ell) (\text{SemigroupCat } \ell)$

$\text{Free } \ell = \text{record}$

$$\begin{aligned} \{F_0 &= \text{List}_1 \text{SG} \\ ; F_1 &= \lambda f \rightarrow \text{list}_1 (\llbracket - \rrbracket \circ f) \\ ; \text{identity} &= \text{map-id} \\ ; \text{homomorphism} &= \text{map-}\circ \\ ; F\text{-resp-}\equiv &= \lambda F \approx G \rightarrow \text{map-cong } (\lambda x \rightarrow F \approx G \{x\}) \\ \} \end{aligned}$$

$\text{Free-isFaithful} : \{\ell : \text{Level}\} \rightarrow \text{Faithful } (\text{Free } \ell)$

$\text{Free-isFaithful } F \ G \ F \approx G \{x\} = \llbracket - \rrbracket\text{-injective } (F \approx G \llbracket x \rrbracket)$

$\text{TreeLeft} : (\ell : \text{Level}) \rightarrow \text{Adjunction } (\text{Free } \ell) (\text{Forget } \ell)$

$\text{TreeLeft } \ell = \text{record}$

$$\begin{aligned} \{ \text{unit} &= \text{record } \{ \eta = \lambda _ \rightarrow [-]; \text{commute} = \lambda _ \rightarrow \equiv.\text{refl} \} \\ ; \text{counit} &= \text{record} \\ &\quad \{ \eta = \lambda S \rightarrow \text{list}_1 \text{id} \\ &\quad ; \text{commute} = \lambda \{X\} \{Y\} F \rightarrow \text{rec } \doteq\text{-refl } (\lambda x \times xs \text{ ind} \rightarrow \equiv.\text{cong } (\text{Op } Y (\text{mor } F x)) \text{ ind } \langle \equiv \rangle \text{ pres } F) \\ &\quad \} \\ ; \text{zig} &= \text{rec } \doteq\text{-refl } (\lambda x \times xs \text{ ind} \rightarrow \equiv.\text{cong } (x :: _) \text{ ind}) \\ ; \text{zag} &= \equiv.\text{refl} \\ \} \end{aligned}$$

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 ()`

`ℳ : Magma lzero`

`ℳ = MkMagma ℕ _⟨_⟩_`

`ℳ : Semigroup`

`ℳ = Functor.F0 FreeM ℳ`

`_⊕_ = Magma.Op (Functor.F0 (ForgetM lzero) ℳ)`

`inj : MagmaHom ℳ (Functor.F0 (ForgetM lzero) ℳ)`

`inj = η unit ℳ`

`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
  ■

```

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 $c : \text{Srcs} \rightarrow \text{Type}$ becomes an argument $(ss : \text{Srcs}) \rightarrow X\ ss \rightarrow X\ (c\ ss)$, 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 $\{ss : \text{Srcs}\} \rightarrow X\ ss \rightarrow X\ (c\ ss)$ then, for one reason or another, the dependent type X needs to be supplies explicitly –yellow Agda! Hence, it behooves us to use explicit in this case. Sometimes, the yellow cannot be avoided.

13.2 Definition

```

record Monoid ℓ : Set (Isuc 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-I = ≡.cong (mor F) (pres-I G) (≡≡) pres-I F
  ; pres-O = ≡.cong (mor F) (pres-O G) (≡≡) pres-O F
  }
; comp-is-∘ = ≡-refl
; Id        = MkHom id ≡.refl ≡.refl
; Id-is-id  = ≡-refl
}

MonoidCat : (ℓ : Level) → Category (Isuc ℓ) ℓ ℓ
MonoidCat ℓ = oneSortedCategory ℓ MonoidAlg

```

13.4 Forgetful Functors ???

```

-- 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  $x \sim y \equiv (\forall z \rightarrow x * z \equiv y * z)$ 
-- the monoid-indistinguishability equivalence relation

```

14 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 Isuc; 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 ( _$_ )
open import DataProperties
open import EqualityCombinators

```

14.1 Definition

```

record Inv {ℓ} : Set (Isuc 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

```

14.2 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
  }

```

14.3 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 ×-o : {ℓ : Level} {X Y Z : Set ℓ} {f : X → Y} {g : Y → Z}
  → 2 × (g ∘ f) ≐ 2 × g ∘ 2 × f
2 ×-o = [ ≐-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) (Involutes ℓ)
Left ℓ = record
  {F0      = λ A → record {A = A ⊔ A; _o = swap+; involutive = swap2}
;F1      = λ f → record {mor = 2 × f; pres = 2 ×-over-swap}
;identity  = 2 ×-id≈id
;homomorphism = 2 ×-o
;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 ×-o: 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: ???
- swap₊: ???
- swap²: ???
- ???

There are actually two left adjoints. It seems the choice of `inj1` / `inj2` is free. But that choice does force the order of `id _o` 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 ⊔ id F _o
    ; 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}
  ;cunit = 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!

• ???

14.4 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
    }
  ;cunit = 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
    }
  ;cunit = record {η = λ _ → proj2; commute = λ _ → ≡.refl}
  ;zig = ≡.refl
  }

```

```
;zag    =  ≐-refl
}
```

Note that we have TWO proofs for `AdjRight` since we can construe $A \times A$ as $\{(a, a^\circ) \mid a \in A\}$ or as $\{(a^\circ, a) \mid a \in A\}$ —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!

• ???

14.5 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? []*

15 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 Relation.Binary.PropositionalEquality using (inspect; [ _ ])
open import Function.Equality using ( $\Pi$ ;  $\_ \longrightarrow \_$ ; id;  $\_ \circ \_$ ;  $\_ \langle \$ \rangle \_$ ; cong)
open import Function using ( $\_ \$ \_$ ) renaming (id to id0;  $\_ \circ \_$  to  $\_ \odot \_$ )
open import Function.Equivalence using (Equivalence)

open import Data.List using (List; [];  $\_ ++ \_$ ;  $\_ :: \_$ ; map)
open import Data.Product using ( $\exists$ )
open import Data.Nat using ( $\mathbb{N}$ ; zero; suc)

open import EqualityCombinators
open import DataProperties
open import SetoidEquiv
open import ParComp

open import TypeEquiv using (swap+)
open import SetoidSetoid
open import Relation.Binary.Sum

infix 4 inSetoidEquiv
inSetoidEquiv : {a ℓ : Level} → (S : Setoid a ℓ) → Setoid.Carrier S → Setoid.Carrier S → Set ℓ
inSetoidEquiv = Setoid. $\_ \approx \_$ 

syntax inSetoidEquiv S x y = x  $\approx$  [ S ] y
```

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* $x \in xs$ is the witness. However, we are in the `Setoid` setting, and in a setting where multiplicity matters (i.e. we may have x occurring 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 $x \approx y$ 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.

15.1 Some₀

Setoid-based variant of `Any`.

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

[WK:] $A \longrightarrow SSetoid _ _$ 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 $A \longrightarrow SSetoid _ _$ is to mesh well with the rest. It is not cast in stone and can potentially be weakened. **[]**

[WK:] S would better be explicit. And this module would be much better named!!! **[]**

```
module _ {a ℓa} {S : Setoid a ℓa} (P0 : Setoid.Carrier S → Set ℓa) where
  open Setoid S renaming (Carrier to A)
  data Some0 : List A → Set (a ⊔ ℓa) 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 `Some0` really are just locations: $Some_0 P \, xs \cong \sum i : \text{Fin } (\text{length } xs) \bullet P (x \, ! \, i)$. 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)
```

```
module _ {a ℓa} {S : Setoid a ℓa} {P0 : Setoid.Carrier S → Set ℓa} where
  open Setoid S renaming (Carrier to A)
  infix 3 _≈_
  data _≈_ : {ys : List A} (pf pf' : Some0 {S = S} P0 ys) → Set (a ⊔ ℓa) 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 P0 xs} {qxs : Some0 P0 xs}
      → _≈_ pxs qxs → _≈_ (there {x = x} pxs) (there {x = x} qxs)
```

Notice that these another from of “natural numbers” whose elements are of the form `thereEqn (hereEq Px Qx _ _)` for some $n : \mathbb{N}$.

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.

```
module _ {a ℓa} {S : Setoid a ℓa} {P0 : Setoid.Carrier S → Set ℓa} where
  open Setoid S renaming (Carrier to A)
  ≈-refl : {xs : List A} {p : Some0 {S = 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 = S} P0 xs} {q : Some0 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 = 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 = S} P0 xs} → p ≡ q → p ≈ q
⇒⇒≈ ≡.refl = ≈-refl
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 ⊔ a)
  Some xs = record
    { Carrier      = Some0 {S = A} P0 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

```

15.2 Membership module

[WK: Please don't waste valuable variable names on levels.] [JC: Good point.]

```

module Membership {ℓS ℓs : Level} (S : Setoid ℓS ℓs) where
  open Setoid S renaming (trans to _{≈}_)
  _{≈} _ : {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
  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 ≈_s y".

```

setoid≈ : Carrier → S → SSetoid ℓs ℓs
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 }
    }
  }
_∈ _ : Carrier → List Carrier → Setoid (ℓS ⊔ ℓs) (ℓS ⊔ ℓs)
x ∈ xs = Some (setoid≈ 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

```

```

 $\epsilon_0\text{-subst}_1 \{x\} \{y\} \{o \_ :: \_ \} x \approx y \text{ (here } a \approx x \text{ px)} = \text{here } a \approx x \text{ (sym } x \approx y \langle \approx \rangle \text{ px)}$ 
 $\epsilon_0\text{-subst}_1 \{x\} \{y\} \{o \_ :: \_ \} x \approx y \text{ (there } x \in xs) = \text{there } (\epsilon_0\text{-subst}_1 x \approx y x \in xs)$ 
 $\epsilon_0\text{-subst}_1\text{-cong} : \{x y : \text{Carrier}\} \{xs : \text{List Carrier}\} (x \approx y : x \approx y)$ 
 $\{i j : x \in_0 xs\} \rightarrow i \approx j \rightarrow \epsilon_0\text{-subst}_1 x \approx y i \approx \epsilon_0\text{-subst}_1 x \approx y j$ 
 $\epsilon_0\text{-subst}_1\text{-cong } x \approx y \text{ (hereEq px qy } x \approx z y \approx z) = \text{hereEq (sym } x \approx y \langle \approx \rangle \text{ px) (sym } x \approx y \langle \approx \rangle \text{ qy) } x \approx z y \approx z$ 
 $\epsilon_0\text{-subst}_1\text{-cong } x \approx y \text{ (thereEq i \approx j) = thereEq } (\epsilon_0\text{-subst}_1\text{-cong } x \approx y i \approx j)$ 
 $\epsilon_0\text{-subst}_1\text{-equiv} : \{x y : \text{Carrier}\} \{xs : \text{List Carrier}\} \rightarrow x \approx y \rightarrow (x \in xs) \cong (y \in xs)$ 
 $\epsilon_0\text{-subst}_1\text{-equiv } \{x\} \{y\} \{xs\} x \approx y = \text{record}$ 
 $\{ \text{to} = \text{record } \{ \_ \$ \} \_ = \epsilon_0\text{-subst}_1 x \approx y; \text{cong} = \epsilon_0\text{-subst}_1\text{-cong } x \approx y \}$ 
 $; \text{from} = \text{record } \{ \_ \$ \} \_ = \epsilon_0\text{-subst}_1 (\text{sym } x \approx y); \text{cong} = \epsilon_0\text{-subst}_1\text{-cong}' \}$ 
 $; \text{inverse-of} = \text{record } \{ \text{left-inverse-of} = \text{left-inv}; \text{right-inverse-of} = \text{right-inv} \}$ 
where
 $\epsilon_0\text{-subst}_1\text{-cong}' : \forall \{ys\} \{i j : y \in_0 ys\} \rightarrow i \approx j \rightarrow \epsilon_0\text{-subst}_1 (\text{sym } x \approx y) i \approx \epsilon_0\text{-subst}_1 (\text{sym } x \approx y) j$ 
 $\epsilon_0\text{-subst}_1\text{-cong}' \text{ (hereEq px qy } x \approx z y \approx z) = \text{hereEq (sym (sym } x \approx y) \langle \approx \rangle \text{ px) (sym (sym } x \approx y) \langle \approx \rangle \text{ qy) } x \approx z y \approx z$ 
 $\epsilon_0\text{-subst}_1\text{-cong}' \text{ (thereEq i \approx j) = thereEq } (\epsilon_0\text{-subst}_1\text{-cong}' i \approx j)$ 
 $\text{left-inv} : \forall \{ys\} (x \in_0 ys : x \in_0 ys) \rightarrow \epsilon_0\text{-subst}_1 (\text{sym } x \approx y) (\epsilon_0\text{-subst}_1 x \approx y x \in_0 ys) \approx x \in_0 ys$ 
 $\text{left-inv (here sm px) = hereEq (sym (sym } x \approx y) \langle \approx \rangle (\text{sym } x \approx y \langle \approx \rangle \text{ px})) px sm sm}$ 
 $\text{left-inv (there } x \in_0 ys) = \text{thereEq (left-inv } x \in_0 ys)$ 
 $\text{right-inv} : \forall \{ys\} (y \in_0 ys : y \in_0 ys) \rightarrow \epsilon_0\text{-subst}_1 x \approx y (\epsilon_0\text{-subst}_1 (\text{sym } x \approx y) y \in_0 ys) \approx y \in_0 ys$ 
 $\text{right-inv (here sm px) = hereEq (sym } x \approx y \langle \approx \rangle (\text{sym (sym } x \approx y) \langle \approx \rangle \text{ px})) px sm sm}$ 
 $\text{right-inv (there } y \in_0 ys) = \text{thereEq (right-inv } y \in_0 ys)$ 
 $\text{BagEq} : (xs ys : \text{List Carrier}) \rightarrow \text{Set } (\ell S \sqcup \ell s)$ 
 $\text{BagEq } xs \text{ ys} = \{x : \text{Carrier}\} \rightarrow (x \in xs) \cong (x \in ys)$ 
 $\epsilon_0\text{-Subst}_2 : \{x : \text{Carrier}\} \{xs ys : \text{List Carrier}\} \rightarrow \text{BagEq } xs \text{ ys} \rightarrow x \in xs \longrightarrow x \in ys$ 
 $\epsilon_0\text{-Subst}_2 \{x\} \{xs\} \{ys\} xs \cong ys = \_ \cong \_ . \text{to } (xs \cong ys \{x\})$ 
 $\epsilon_0\text{-subst}_2 : \{x : \text{Carrier}\} \{xs ys : \text{List Carrier}\} \rightarrow \text{BagEq } xs \text{ ys} \rightarrow x \in_0 xs \rightarrow x \in_0 ys$ 
 $\epsilon_0\text{-subst}_2 xs \cong ys x \in xs = \epsilon_0\text{-Subst}_2 xs \cong ys \$ x \in xs$ 
 $\epsilon_0\text{-subst}_2\text{-cong} : \{x : \text{Carrier}\} \{xs ys : \text{List Carrier}\} (xs \cong ys : \text{BagEq } xs \text{ ys})$ 
 $\rightarrow \{p q : x \in_0 xs\}$ 
 $\rightarrow p \approx q$ 
 $\rightarrow \epsilon_0\text{-subst}_2 xs \cong ys p \approx \epsilon_0\text{-subst}_2 xs \cong ys q$ 
 $\epsilon_0\text{-subst}_2\text{-cong } xs \cong ys = \text{cong } (\epsilon_0\text{-Subst}_2 xs \cong ys)$ 
 $\text{transport} : (Q : S \longrightarrow \text{SSetoid } \ell s \ell s) \rightarrow$ 
 $\text{let } Q_0 = \lambda e \rightarrow \text{Setoid.Carrier } (Q \$) e \text{ in}$ 
 $\{a x : \text{Carrier}\} (p : Q_0 a) (a \approx x : a \approx x) \rightarrow Q_0 x$ 
 $\text{transport } Q p a \approx x = \text{Equivalence.to } (\Pi.\text{cong } Q a \approx x) \$ p$ 
 $\epsilon_0\text{-subst}_1\text{-elim} : \{x : \text{Carrier}\} \{xs : \text{List Carrier}\} (x \in xs : x \in_0 xs) \rightarrow$ 
 $\epsilon_0\text{-subst}_1 \text{ refl } x \in xs \approx x \in xs$ 
 $\epsilon_0\text{-subst}_1\text{-elim (here sm px) = hereEq (sym refl } \langle \approx \rangle \text{ px) px sm sm}$ 
 $\epsilon_0\text{-subst}_1\text{-elim (there } x \in xs) = \text{thereEq } (\epsilon_0\text{-subst}_1\text{-elim } x \in xs)$ 
  -- note how the back-and-forth is clearly apparent below
 $\epsilon_0\text{-subst}_1\text{-sym} : \{a b : \text{Carrier}\} \{xs : \text{List Carrier}\} \{a \approx b : a \approx b\}$ 
 $\{a \in xs : a \in_0 xs\} \{b \in xs : b \in_0 xs\} \rightarrow \epsilon_0\text{-subst}_1 a \approx b a \in xs \approx b \in xs \rightarrow$ 
 $\epsilon_0\text{-subst}_1 (\text{sym } a \approx b) b \in xs \approx a \in xs$ 
 $\epsilon_0\text{-subst}_1\text{-sym } \{a \approx b = a \approx b\} \{ \text{here sm px} \} \{ \text{here sm}_1 \text{ px}_1 \} (\text{hereEq } \_ . \text{px}_1 . \text{sm} . \text{sm}_1) = \text{hereEq (sym (sym } a \approx b) \langle \approx \rangle \text{ px}_1) \text{ px sm}_1 \text{ sm}$ 
 $\epsilon_0\text{-subst}_1\text{-sym } \{a \in xs = \text{there } a \in xs\} \{ \text{here sm px} \} ()$ 
 $\epsilon_0\text{-subst}_1\text{-sym } \{a \in xs = \text{here sm px}\} \{ \text{there } b \in xs \} ()$ 
 $\epsilon_0\text{-subst}_1\text{-sym } \{a \in xs = \text{there } a \in xs\} \{ \text{there } b \in xs \} (\text{thereEq pf}) = \text{thereEq } (\epsilon_0\text{-subst}_1\text{-sym pf})$ 
 $\epsilon_0\text{-subst}_1\text{-trans} : \{a b c : \text{Carrier}\} \{xs : \text{List Carrier}\} \{a \approx b : a \approx b\}$ 
 $\{b \approx c : b \approx c\} \{a \in xs : a \in_0 xs\} \{b \in xs : b \in_0 xs\} \{c \in xs : c \in_0 xs\} \rightarrow$ 
 $\epsilon_0\text{-subst}_1 a \approx b a \in xs \approx b \in xs \rightarrow \epsilon_0\text{-subst}_1 b \approx c b \in xs \approx c \in xs \rightarrow$ 
 $\epsilon_0\text{-subst}_1 (a \approx b \langle \approx \rangle b \approx c) a \in xs \approx c \in xs$ 
 $\epsilon_0\text{-subst}_1\text{-trans } \{a \approx b = a \approx b\} \{b \approx c\} \{ \text{here sm px} \} \{ o (\text{here } y \approx z \text{ qy}) \} \{ o (\text{here } z \approx w \text{ qz}) \} (\text{hereEq } \_ . \text{qy} . \text{sm } y \approx z) (\text{hereEq } \_ . \text{qz} \text{ foo } z \approx w)$ 

```

$\epsilon_0\text{-subst}_1\text{-trans } \{a \approx b = a \approx b\} \{b \approx c\} \{\text{there } a \in xs\} \{\text{there } b \in xs\} \{\circ (\text{there } _)\} (\text{thereEq } pp) (\text{thereEq } qq) = \text{thereEq } (\epsilon_0\text{-subst}_1\text{-trans } pp)$

infix 3 $_ \square$

infixr 2 $_ \approx (_) _$

$_ \approx (_) _ : \{P_0 : \text{Carrier} \rightarrow \text{Set } \ell s\} \{xs : \text{List Carrier}\} (X : \text{Some}_0 \{S = S\} P_0 xs) \{Y Z : \text{Some}_0 P_0 xs\} \rightarrow X \approx Y \rightarrow Y \approx Z \rightarrow X \approx Z$

$X \approx (_ \approx Y) Y \approx Z = \approx\text{-trans } X \approx Y Y \approx Z$

$_ \square : \{P_0 : \text{Carrier} \rightarrow \text{Set } \ell s\} \{xs : \text{List Carrier}\} (X : \text{Some}_0 \{S = S\} P_0 xs) \rightarrow X \approx X$

$X \square = \approx\text{-refl}$

$\text{here} \approx \text{there} : \{P_0 : \text{Carrier} \rightarrow \text{Set } \ell s\} \{xs : \text{List Carrier}\} \{a x : \text{Carrier}\} \{a \approx x : a \approx x\}$

$\{Pa : P_0 a\} \{pf : \text{Some}_0 \{S = S\} P_0 xs\} \rightarrow$

$\text{here } \{x = x\} \{a\} \{xs\} a \approx x Pa \approx \text{there } pf \rightarrow \perp \{ \ell s \}$

$\text{here} \approx \text{there} ()$

private

$\text{Some} [] \rightarrow \perp : \{P_0 : \text{Carrier} \rightarrow \text{Set } \ell s\} \rightarrow \text{Some}_0 \{ \ell s \} \{ \ell s \} \{S\} P_0 [] \rightarrow \perp \{ \ell s \}$

$\text{Some} [] \rightarrow \perp ()$

[WK:] *Trying — unfinished — commented out:*

$\epsilon_0\text{-subst}_1\text{-to} : \{a b : \text{Carrier}\} \{zs ws : \text{List Carrier}\} \{a \approx b : a \approx b\}$

$\rightarrow (zs \cong ws : \text{BagEq } zs ws) (a \in zs : a \in_0 zs)$

$\rightarrow \epsilon_0\text{-subst}_1 a \approx b (\epsilon_0\text{-subst}_2 zs \cong ws a \in zs) \approx \epsilon_0\text{-subst}_2 zs \cong ws (\epsilon_0\text{-subst}_1 a \approx b a \in zs)$

$\epsilon_0\text{-subst}_1\text{-to} \{a\} \{b\} \{\circ (_ :: _)\} \{ws\} \{a \approx b\} zs \cong ws a \in zs =$

$\epsilon_0\text{-subst}_1 a \approx b (\epsilon_0\text{-subst}_2 zs \cong ws a \in zs)$

$\approx (_ \{ !\text{cong } (_ \cong _ . \text{to } zs \cong ws) ! \})$

$\epsilon_0\text{-subst}_2 zs \cong ws (\epsilon_0\text{-subst}_1 a \approx b a \in zs)$

\square

$\epsilon_0\text{-subst}_1\text{-to} \{a\} \{b\} \{\circ (_ :: _)\} \{ws\} \{a \approx b\} zs \cong ws (\text{here } \{x\} \{c\} \{xs\} c \approx x a \approx c)$

with $_ \cong _ . \text{to } zs \cong ws \$ \text{here } c \approx x a \approx c \mid \text{inspect } (_ \$) _ (_ \cong _ . \text{to } zs \cong ws) (\text{here } c \approx x a \approx c)$

$\epsilon_0\text{-subst}_1\text{-to} \{a\} \{b\} \{\circ (x :: xs)\} \{\circ (_ :: _)\} \{a \approx b\} zs \cong ws (\text{here } \{x\} \{z\} \{xs\} z \approx x a \approx z) \mid \text{here } \{y\} \{w\} \{ys\} w \approx y a \approx w \mid [a \in ws\text{-eq}] =$

where

$eq_1 : \text{here } z \approx x a \approx z \approx \text{here } z \approx x (a \approx b (_ \approx _) a \approx z)$

$eq_1 = \text{hereEq } a \approx z (a \approx b (_ \approx _) a \approx z) z \approx x z \approx x$

$\epsilon_0\text{-subst}_1\text{-to} \{a\} \{b\} \{\circ (x :: xs)\} \{\circ (_ :: _)\} \{a \approx b\} zs \cong ws (\text{here } \{x\} \{a_1\} \{xs\} c \approx x a \approx c) \mid \text{there } a \in ws \mid [a \in ws\text{-eq}] = \{!!\}$

$\epsilon_0\text{-subst}_1\text{-to} \{a\} \{b\} \{\circ (_ :: _)\} \{ws\} \{a \approx b\} zs \cong ws (\text{there } a \in zs) = \{!!\}$

[]

postulate

$\epsilon_0\text{-subst}_1\text{-to}' : \{a b : \text{Carrier}\} \{zs ws : \text{List Carrier}\} \{a \approx b : a \approx b\} (\text{eq} : \text{BagEq } zs ws)$

$(a \in zs : a \in_0 zs) \rightarrow \epsilon_0\text{-subst}_1 a \approx b (\epsilon_0\text{-subst}_2 \text{eq } a \in zs) \approx \epsilon_0\text{-subst}_2 \text{eq } (\epsilon_0\text{-subst}_1 a \approx b a \in zs)$

[WK:] *A BagEq that is easily recognised as not just a permutation on indices.*

Commented out since there are still holes.

module NICE where

open import Data.Bool

data Nice $\{ \ell A : \text{Level} \} \{ A : \text{Set } \ell A \} (x : A) : A \rightarrow \text{Set}$ **where**

$\text{nice} : \text{Bool} \rightarrow \text{Nice } x x$

$\text{NiceSetoid} : \{ \ell A : \text{Level} \} \rightarrow \text{Set } \ell A \rightarrow \text{Setoid } \ell A \text{ lzero}$

NiceSetoid A = **record**

```
{ Carrier = A
; _≈_ = Nice
; isEquivalence = record
  { refl = nice true
  ; sym = λ {(nice b) → nice (¬ b)}
  ; trans = λ {(nice b1) (nice b2) → nice (b1 xor b2)}
  }
}
```

data E : Set **where**

E₁ E₂ : E

xs ys : List E

xs = E₁ :: E₂ :: E₂ :: []

ys = E₂ :: E₁ :: E₂ :: []

open Membership (*NiceSetoid* E)

xs⇒ys : (e : E) → Some₀ {S = *NiceSetoid* E} (Nice e) xs
→ Some₀ {S = *NiceSetoid* E} (Nice e) ys

xs⇒ys E₁ (here (nice false) (nice false)) = there (here (nice false) (nice true))

xs⇒ys E₁ (here (nice false) (nice true)) = there (here (nice true) (nice true))

xs⇒ys E₁ (here (nice true) (nice false)) = there (here (nice false) (nice false))

xs⇒ys E₁ (here (nice true) (nice true)) = there (here (nice true) (nice false))

xs⇒ys E₁ (there (here (nice x) ()))

xs⇒ys E₁ (there (there (here (nice x) ())))

xs⇒ys E₁ (there (there (there ())))

xs⇒ys E₂ (here (nice x) ()))

xs⇒ys E₂ (there (here (nice false) (nice false))) = here (nice true) (nice true)

xs⇒ys E₂ (there (here (nice false) (nice true))) = there (there (here (nice true) (nice true)))

xs⇒ys E₂ (there (here (nice true) (nice false))) = here (nice true) (nice false)

xs⇒ys E₂ (there (here (nice true) (nice true))) = there (there (here (nice false) (nice true)))

xs⇒ys E₂ (there (there (here (nice false) (nice false)))) = here (nice false) (nice false)

xs⇒ys E₂ (there (there (here (nice false) (nice true)))) = there (there (here (nice true) (nice false)))

xs⇒ys E₂ (there (there (here (nice true) (nice false)))) = there (there (here (nice false) (nice false)))

xs⇒ys E₂ (there (there (here (nice true) (nice true)))) = here (nice false) (nice true)

xs⇒ys E₂ (there (there (there ())))

xs⇒ys-cong : (e : E) {i j : Some₀ (Nice e) xs} → i ≈ j → xs⇒ys e i ≈ xs⇒ys e j

xs⇒ys-cong E₁ {○ (here (nice false) (nice false))} {○ (here (nice false) (nice false))} (hereEq (nice false) (nice false) (nice false) (nice false))

xs⇒ys-cong E₁ {○ (here (nice false) (nice false))} {○ (here (nice true) (nice false))} (hereEq (nice false) (nice false) (nice false) (nice true))

xs⇒ys-cong E₁ {○ (here (nice true) (nice false))} {○ (here (nice false) (nice false))} (hereEq (nice false) (nice false) (nice true) (nice false))

xs⇒ys-cong E₁ {○ (here (nice true) (nice false))} {○ (here (nice true) (nice false))} (hereEq (nice false) (nice false) (nice true) (nice true))

(hereEq (nice false) (nice false) (nice false) (nice false))

xs⇒ys-cong E₁ {○ (here (nice false) (nice false))} {○ (here (nice false) (nice true))} (hereEq (nice false) (nice true) (nice false) (nice false))

xs⇒ys-cong E₁ {○ (here (nice false) (nice false))} {○ (here (nice true) (nice true))} (hereEq (nice false) (nice true) (nice false) (nice true))

xs⇒ys-cong E₁ {○ (here (nice true) (nice false))} {○ (here (nice false) (nice true))} (hereEq (nice false) (nice true) (nice true) (nice false))

xs⇒ys-cong E₁ {○ (here (nice true) (nice false))} {○ (here (nice true) (nice true))} (hereEq (nice false) (nice true) (nice true) (nice true))

xs⇒ys-cong E₁ {○ (here (nice true) (nice false))} {○ (here (nice false) (nice false))} (hereEq (nice true) (nice false) (nice false) (nice false))

xs⇒ys-cong E₁ {○ (here (nice true) (nice false))} {○ (here (nice true) (nice false))} (hereEq (nice true) (nice false) (nice false) (nice true))

xs⇒ys-cong E₁ {○ (here (nice true) (nice true))} {○ (here (nice false) (nice false))} (hereEq (nice true) (nice false) (nice true) (nice false))

xs⇒ys-cong E₁ {○ (here (nice true) (nice true))} {○ (here (nice true) (nice false))} (hereEq (nice true) (nice false) (nice true) (nice true))

xs⇒ys-cong E₁ {○ (here (nice true) (nice true))} {○ (here (nice false) (nice true))} (hereEq (nice true) (nice true) (nice false) (nice false))

xs⇒ys-cong E₁ {○ (here (nice true) (nice true))} {○ (here (nice true) (nice true))} (hereEq (nice true) (nice true) (nice false) (nice true))

xs⇒ys-cong E₁ {○ (here (nice true) (nice true))} {○ (here (nice false) (nice true))} (hereEq (nice true) (nice true) (nice true) (nice false))

xs⇒ys-cong E₁ {○ (here (nice true) (nice true))} {○ (here (nice true) (nice true))} (hereEq (nice true) (nice true) (nice true) (nice true))

xs⇒ys-cong E₁ {○ (there (here _ (nice x)))} {○ (there (here y≈z (nice x₁)))} (thereEq (hereEq (nice x) (nice x₁) (y≈z))

(thereEq (thereEq (hereEq (nice x) (nice x₁) (y≈z)) (thereEq (thereEq (hereEq (nice x) (nice x₁) (y≈z))

(thereEq (thereEq (thereEq ())))

```

xs⇒ys-cong E2 {◦ (here x≈z px)} {◦ (here y≈z qy)} (hereEq px qy x≈z y≈z) = {!!}
xs⇒ys-cong E2 {◦ (there _)} {◦ (there _)} (thereEq i≈j) = {!!}
ys⇒xs : (e : E) → Some0 {S = NiceSetoid E} (Nice e) ys
        → Some0 {S = NiceSetoid E} (Nice e) xs
ys⇒xs e p = {!!}
xs≈ys : BagEq xs ys
xs≈ys {e} = record
  {to = record {_⟨$⟩_ = xs⇒ys e; cong = xs⇒ys-cong e}
  ;from = record {_⟨$⟩_ = ys⇒xs e; cong = {!!}}
  ;inverse-of = {!!}
  }

```

1

15.3 $++\cong : \dots \rightarrow (\text{Some } P \text{ } xs \uplus \text{Some } P \text{ } ys) \cong \text{Some } P (xs + ys)$

```

module _ {a ℓa : Level} {A : Setoid a ℓa} {P : A → SSetoid ℓa ℓa} where
  ++≅ : {xs ys : List (Setoid.Carrier A)} → (Some P xs  $\uplus$  Some P ys)  $\cong$  Some P (xs + ys)
  ++≅ {xs} {ys} = record
    {to = record {_⟨$⟩_ =  $\uplus \rightarrow ++$ ; cong =  $\uplus \rightarrow ++$ -cong}
    ;from = record {_⟨$⟩_ = ++ $\rightarrow \uplus$  xs; cong = new-cong xs}
    ;inverse-of = record
      {left-inverse-of = lefty xs
      ;right-inverse-of = righty xs
      }
  }
where
  open Setoid A
  private P0 = λ e → Setoid.Carrier (P ⟨$⟩ e)
  _ $\sim$ _ = _≈_;  $\sim$ -refl = ≈-refl {S = A} {P0}
  -- “earlier”
   $\uplus \rightarrow^l$  : ∀ {ws zs} → Some0 {S = A} P0 ws → Some0 {S = A} P0 (ws + zs)
   $\uplus \rightarrow^l$  (here p a≈x) = here p a≈x
   $\uplus \rightarrow^l$  (there p) = there ( $\uplus \rightarrow^l$  p)
  yo : {xs : List Carrier} {x y : Some0 P0 xs} → x  $\sim$  y →  $\uplus \rightarrow^l$  x  $\sim$   $\uplus \rightarrow^l$  y
  yo (hereEq px py _ _) = hereEq px py _ _
  yo (thereEq pf) = thereEq (yo pf)
  -- “later”
   $\uplus \rightarrow^r$  : ∀ xs {ys} → Some0 {S = A} P0 ys → Some0 P0 (xs + ys)
   $\uplus \rightarrow^r$  [] p = p
   $\uplus \rightarrow^r$  (x :: xs) p = there ( $\uplus \rightarrow^r$  xs p)
  oy : (xs : List Carrier) {x y : Some0 P0 ys} → x  $\sim$  y →  $\uplus \rightarrow^r$  xs x  $\sim$   $\uplus \rightarrow^r$  xs y
  oy [] pf = pf
  oy (x :: xs) pf = thereEq (oy xs pf)
  -- Some0 is ++ $\rightarrow \uplus$ -homomorphic, in the second argument.
   $\uplus \rightarrow ++$  : ∀ {zs ws} → (Some0 P0 zs  $\uplus$  Some0 P0 ws) → Some0 P0 (zs + ws)
   $\uplus \rightarrow ++$  (inj1 x) =  $\uplus \rightarrow^l$  x
   $\uplus \rightarrow ++$  {zs} (inj2 y) =  $\uplus \rightarrow^r$  zs y
  ++ $\rightarrow \uplus$  : ∀ xs {ys} → Some0 P0 (xs + ys) → Some0 P0 xs  $\uplus$  Some0 P0 ys
  ++ $\rightarrow \uplus$  [] p = inj2 p
  ++ $\rightarrow \uplus$  (x :: l) (here p _) = inj1 (here p _)
  ++ $\rightarrow \uplus$  (x :: l) (there p) = (there  $\uplus_1$  id0) (++ $\rightarrow \uplus$  l p)

```

```

-- all of the following may need to change
⊞→++-cong : {a b : Some₀ P₀ xs ⊞ Some₀ 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 {S = A} P₀ xs → Some₀ {S = A} P₀ us)
  (F-cong : {p q : Some₀ P₀ xs} → p ~ q → F p ~ F q)
  (G : Some₀ {S = A} P₀ ys → Some₀ {S = A} P₀ vs)
  (G-cong : {p q : Some₀ P₀ ys} → p ~ q → G p ~ G q)
  → {pf pf' : Some₀ P₀ xs ⊞ Some₀ 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 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 P₀ xs ⊞ Some₀ 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 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

```

15.4 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} {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 {} {} }
  }

```

15.5 $\text{map}^\cong : \dots \rightarrow \text{Some } (P \circ f) \text{ xs} \cong \text{Some } P (\text{map } (_ \langle \$ \rangle _ f) \text{ xs})$

```

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 ( _⟨$⟩_ f) xs)
map≅ {a} {ℓa} {A} {B} {P} {f} = record
  {to = record { _⟨$⟩_ = map+; cong = map+-cong }
  ; from = record { _⟨$⟩_ = map-; cong = map--cong }
  ; inverse-of = record { left-inverse-of = map- ∘ map+; right-inverse-of = map+ ∘ map- }
  }
where
open Setoid
open Membership using (transport)
A0 = Carrier A
P0 = λ e → Setoid.Carrier (P ⟨$⟩ e)
_~_ = _≅_ {S = B} {P0 = P0}
map+ : {xs : List A0} → Some0 {S = A} (P0 ∘ _⟨$⟩_ f) xs → Some0 {S = B} P0 (map ( _⟨$⟩_ 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 {S = B} P0 (map ( _⟨$⟩_ f) xs) → Some0 {S = A} (P0 ∘ ( _⟨$⟩_ f)) xs
map- {[]} ()
map- {x :: xs} (here {b} b≈x p) = here (refl A) (Equivalence.to (Π.cong P b≈x) ⟨$⟩ p)
map- {x :: xs} (there p) = there (map- {xs = xs} p)
map+ ∘ map- : {xs : List A0} → (p : Some0 P0 (map ( _⟨$⟩_ 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 (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 (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 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)

```

15.6 FindLose

```

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 {S = 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 $\in$ ys , Pa) = find p
                        in a , there a $\in$ ys , Pa
lose : {ys : List Carrier}  $\rightarrow$  Support ys  $\rightarrow$  Some0 {S = A} P0 ys
lose (y , here b $\approx$ y py , Py) = here b $\approx$ y (Equivalence.to (Π.cong P py) Π.($) Py)
lose (y , there {b} y $\in$ ys , Py) = there (lose (y , y $\in$ ys , Py))

```

15.7 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 _ {a  $\ell$ a : Level} {A : Setoid a  $\ell$ a} {P : A  $\longrightarrow$  SSetoid  $\ell$ a  $\ell$ a} where
  open Membership A
  open Setoid A
  private
    P0 =  $\lambda$  e  $\rightarrow$  Setoid.Carrier (Π. _ ($)_ P e)
    Support =  $\lambda$  ys  $\rightarrow$   $\Sigma$  y : Carrier • y  $\in_0$  ys  $\times$  P0 y
    squish : {x y : Setoid.Carrier A}  $\rightarrow$  P0 x  $\rightarrow$  P0 y  $\rightarrow$  Set  $\ell$ a
    squish _ _ =  $\top$ 

    -- FIXME : this definition is still not right
    _  $\approx$  _ : {ys : List Carrier}  $\rightarrow$  Support ys  $\rightarrow$  Support ys  $\rightarrow$  Set (a  $\sqcup$   $\ell$ a)
    (a , a $\in$ xs , Pa)  $\approx$  (b , b $\in$ xs , Pb) =
       $\Sigma$  (a  $\approx$  b) ( $\lambda$  a $\approx$ b  $\rightarrow$   $\epsilon_0$ -subst1 a $\approx$ b a $\in$ xs  $\approx$  b $\in$ xs  $\times$  squish Pa Pb)

     $\Sigma$ -Setoid : (ys : List Carrier)  $\rightarrow$  Setoid ( $\ell$ a  $\sqcup$  a) ( $\ell$ a  $\sqcup$  a)
     $\Sigma$ -Setoid [] =  $\perp\perp$ 
     $\Sigma$ -Setoid (y :: ys) = record
      { Carrier = Support (y :: ys)
      ; _ $\approx$ _ = _  $\approx$  _
      ; isEquivalence = record
          { refl =  $\lambda$  {s}  $\rightarrow$  Refl {s}
          ; sym =  $\lambda$  {s} {t} eq  $\rightarrow$  Sym {s} {t} eq
          ; trans =  $\lambda$  {s} {t} {u} a b  $\rightarrow$  Trans {s} {t} {u} a b
          }
      }
    where
      Refl : Reflexive _  $\approx$  _
      Refl {a1 , here sm px , Pa} = refl , hereEq (trans (sym refl) px) px sm sm , tt
      Refl {a1 , there a $\in$ xs , Pa} = refl , thereEq ( $\epsilon_0$ -subst1-elim a $\in$ xs) , tt
      Sym : Symmetric _  $\approx$  _
      Sym (a $\approx$ b , a $\in$ xs $\approx$ b $\in$ xs , Pa $\approx$ Pb) = sym a $\approx$ b ,  $\epsilon_0$ -subst1-sym a $\in$ xs $\approx$ b $\in$ xs , tt
      Trans : Transitive _  $\approx$  _
      Trans (a $\approx$ b , a $\in$ xs $\approx$ b $\in$ xs , Pa $\approx$ Pb) (b $\approx$ c , b $\in$ xs $\approx$ c $\in$ xs , Pb $\approx$ Pc) = trans a $\approx$ b b $\approx$ c ,  $\epsilon_0$ -subst1-trans a $\in$ xs $\approx$ b $\in$ xs b $\in$ xs $\approx$ c $\in$ xs , tt

    module  $\approx$  {ys} where open Setoid ( $\Sigma$ -Setoid ys) public
    open FindLose P

    find-cong : {xs : List Carrier} {p q : Some0 P0 xs}  $\rightarrow$  p  $\approx$  q  $\rightarrow$  find p  $\approx$  find q
    find-cong {p =  $\circ$  (here x $\approx$ z px)} {q =  $\circ$  (here y $\approx$ z qy)} (hereEq px qy x $\approx$ z y $\approx$ z) =
      refl , hereEq (trans (sym refl) (sym x $\approx$ z)) (sym y $\approx$ z) x $\approx$ z y $\approx$ z , tt
    find-cong {p =  $\circ$  (there _)} {q =  $\circ$  (there _)} (thereEq p $\approx$ q) =
      proj1 (find-cong p $\approx$ q) , thereEq (proj1 (proj2 (find-cong p $\approx$ q))) , proj2 (proj2 (find-cong p $\approx$ q))
    forget-cong : {xs : List Carrier} {i j : Support xs}  $\rightarrow$  i  $\approx$  j  $\rightarrow$  lose i  $\approx$  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 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 (trans (sym (trans (sym a≈x) (sym px))) (sym a≈x)) px a≈x a≈x, t
right-inv (y, there y∈ys, Py) =
  let (α1, α2, α3) = right-inv (y, y∈ys, Py) in
  (α1, thereEq α2, α3)
  -- (proj1 (right-inv (y, y∈ys, Py))) , (thereEq (proj1 (proj2 (right-inv (y, y∈ys, Py))))), !proj2 (proj2 (right-inv!
Σ-Some : (xs : List Carrier) → Some P xs ≅ Σ-Setoid xs
Σ-Some [] = ≅-sym (⊥≅Some[] {a} {ℓa} {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[] {A = A} {setoid≈ z}) ($) ( _≅_ .from iso ($) here refl refl))
Σ-cong {x :: xs} {} iso = ⊥-elim ( _≅_ .from (⊥≅Some[] {A = A} {setoid≈ x}) ($) ( _≅_ .to iso ($) 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 (≅-sym xs≅ys); cong = λ {i j} → xs→ys-cong (≅-sym xs≅ys) {i} {j} }
  ;inverse-of = record
    {left-inverse-of = λ {(z, z∈xs, Pz) → refl, (≈-trans (ε0-subst1-elim _) (left-inverse-of xs≅ys z∈xs), tt)}
    ;right-inverse-of = λ {(z, z∈ys, Pz) → refl, ≈-trans (ε0-subst1-elim _) (right-inverse-of 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, ε0-subst2 eq a∈xs, Pa)
  -- ε0-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, {!!}, tt

```

15.8 Some-cong

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

[WK:] *It has position preservation neither in the assumption (list-rel), nor in the conclusion. Why did you bother with position preservation for $_ \approx _$?* **[JC:]** *Because $_ \approx _$ is about showing that two positions in the same list are equivalent. And list-rel is a permutation between two lists. I agree that $_ \approx _$ could be “loosened”*

to be up to permutation of elements which are \approx to a given one.

But if our notion of permutation is *BagEq*, which depends on \in , which depends on *Some*, which depends on \approx . If that now depends on *BagEq*, we've got a mutual recursion that seems unnecessary. 1

module $_$ {a ℓ_a : Level} {A : Setoid a ℓ_a } {P : A \longrightarrow SSetoid ℓ_a ℓ_a } **where**

open Membership A

open Setoid A

private $P_0 = \lambda e \rightarrow (\Pi. _ \langle \$ \rangle _ P e)$

Some-cong : {xs₁ xs₂ : List Carrier} \rightarrow

$(\forall \{x\} \rightarrow (x \in xs_1) \cong (x \in xs_2)) \rightarrow$

Some P xs₁ \cong Some P xs₂

Some-cong {xs₁} {xs₂} xs₁ \cong xs₂ =

Some P xs₁ \cong $\langle \Sigma\text{-Some } xs_1 \rangle$

$\Sigma\text{-Setoid } \{P = P\} xs_1 \cong \langle \Sigma\text{-cong } xs_1 \cong xs_2 \rangle$

$\Sigma\text{-Setoid } \{P = P\} xs_2 \cong \langle \cong\text{-sym } (\Sigma\text{-Some } xs_2) \rangle$

Some P xs₂ ■

16 Conclusion and Outlook

???

References