

Theories and Data Structures

Jacques Carette, Musa Al-hassy, Wolfram Kahl

June 15, 2017

Abstract

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

In particular, we answer the following questions:

- Why do lists pop-up more frequently to the average programmer than, say, their duals: bags?
- More simply, why do unit and empty types occur so naturally? What about enumerations/sums and records/products?
- Why is it that dependent sums and products do not pop-up 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 Some	27
11.1 Some_0	27
11.2 Membership module	29
11.3 Parallel Composition	29
11.4 $\uplus\uplus$ -comm	30
11.5 $++\cong : \cdots \rightarrow (\text{Some } P \text{ } xs \uplus\uplus \text{Some } P \text{ } ys) \cong \text{Some } P \text{ } (xs + ys)$	31
11.6 Bottom as a setoid	32
11.7 $\text{map}\cong : \cdots \rightarrow \text{Some } (P \circ f) \text{ } xs \cong \text{Some } P \text{ } (\text{map } (_ \langle \$ \rangle _ f) \text{ } xs)$	33
11.8 Some-cong and holes	35
12 Conclusion and Outlook	37

1 Introduction

???

2 Overview

???

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

<https://github.com/JacquesCarette/TheoriesAndDataStructures>

3 Obtaining Forgetful Functors

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

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

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

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

```

module Forget where
open import Level
open import Categories.Category using (Category)
open import Categories.Functor using (Functor)
open import Categories.Agda using (Sets)
open import Function2
open import Function
open import EqualityCombinators

```

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

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

```

record OneSortedAlg (ℓ : Level) : Set (suc (suc ℓ)) where
  field
    Alg      : Set (suc ℓ)
    Carrier  : Alg → Set ℓ
    Hom      : Alg → Alg → Set ℓ
    mor      : {A B : Alg} → Hom A B → (Carrier A → Carrier B)
    comp     : {A B C : Alg} → Hom B C → Hom A B → Hom A C
    .comp-is-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∘l to o-≐-cong2; cong∘r 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 *_⊔_*, which is rather odd since infix operators ought to have higher precedence than equality combinators, yet the standard library assigns *_≈⟨_⟩_* a precedence level of 2. The usage of these two —e.g. in *CommMonoid.lagda*— causes an annoying number of parentheses and so we reassign the level of the infix operator to avoid such a situation.

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

5.1 Generalised Bot and Top

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

```
open import Level
data ⊥ {ℓ : Level} : Set ℓ where
  ⊥-elim : {a ℓ : Level} {A : Set a} → ⊥ {ℓ} → A
```

\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 DataProperties using (T; tt)
open import SetoidEquiv

```

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

```

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

```

Given two elements of a given Setoid A, define a Setoid of equivalences of those elements. We consider all such equivalences to be equivalent. In other words, for $e_1 e_2 : \text{Setoid.Carrier } A$, 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; _∘_; const)
open import Function2              using (_$i)
open import Forget
open import EqualityCombinators
open import DataProperties

```

7.1 Definitions

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

```
record TwoSorted ℓ : Set (lsuc ℓ) where
```

```
  constructor MkTwo
```

```
  field
```

```
    One : Set ℓ
```

```
    Two : Set ℓ
```

```
open TwoSorted
```

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

```
record Hom {ℓ} (Src Tgt : TwoSorted ℓ) : Set ℓ where
```

```
  constructor MkHom
```

```
  field
```

```
    one : One Src → One Tgt
```

```
    two : Two Src → Two Tgt
```

```
open Hom
```

7.2 Category and Forgetful Functors

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

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

```
Twos ℓ = record
```

```
  {Obj      = TwoSorted ℓ
```

```
  ;_⇒_      = Hom
```

```
  ;_≡_      = λ F G → one F ≐ one G × two F ≐ two G
```

```
  ;id       = MkHom id id
```

```
  ;_∘_      = λ F G → MkHom (one F ∘ one G) (two F ∘ two G)
```

```
  ;assoc    = ≐-refl , ≐-refl
```

```
  ;identityl = ≐-refl , ≐-refl
```

```
  ;identityr = ≐-refl , ≐-refl
```

```
  ;equiv    = record
```

```
    {refl    = ≐-refl , ≐-refl
```

```
    ;sym     = λ {(oneEq , twoEq) → ≐-sym oneEq , ≐-sym twoEq}
```

```
    ;trans   = λ {(oneEq1 , twoEq1) (oneEq2 , twoEq2) → ≐-trans oneEq1 oneEq2 , ≐-trans twoEq1 twoEq2}
```

```
    }
```

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

```
  }
```

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

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

$\text{Forget} : (\ell : \text{Level}) \rightarrow \text{Functor} (\text{Twos } \ell) (\text{Sets } \ell)$

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

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

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

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

```
{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 \top is the “largest” type we adjoin to obtain a **Twos** object. This is one way that the unit and empty types naturally arise.

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

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

```
{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
}
```

$\text{Cofree} : (\ell : \text{Level}) \rightarrow \text{Functor} (\text{Sets } \ell) (\text{Twos } \ell)$

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

```
{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)

$\text{Free}^2 : (\ell : \text{Level}) \rightarrow \text{Functor} (\text{Sets } \ell) (\text{Twos } \ell)$

$\text{Free}^2 \ell = \text{record}$

```
{F0          = MkTwo ⊥
;F1          = MkHom id
;identity     = ≡-refl , ≡-refl
;homomorphism = ≡-refl , ≡-refl
;F-resp-≡    = λ f≈g → ≡-refl , λ x → f≈g {x}
}
```

$\text{Cofree}^2 : (\ell : \text{Level}) \rightarrow \text{Functor} (\text{Sets } \ell) (\text{Twos } \ell)$

```

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
}

```

$\text{Left}_2 : (\ell : \text{Level}) \rightarrow \text{Adjunction } (\text{Choice } \ell) (\text{Dup } \ell)$

$\text{Left}_2 \ell = \text{record}$

```
{unit      = record {η = λ _ → MkHom inj1 inj2; commute = λ _ → ≐-refl , ≐-refl}
; counit   = record {η = λ _ → from $\Psi$ ; commute = λ _ {x} → (≡.sym ∘ from $\Psi$ -nat) x}
; zig      = λ { {- } } {x} → from $\Psi$ -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; $_ \sqcup _$ to $_ \sqcup _$)

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

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 $\ell \ell' : \text{Set } (\text{lsuc } (\ell \sqcup \ell'))$ **where**

constructor MkHRel

field

One : Set ℓ

Two : Set ℓ

Rel : One \rightarrow Two \rightarrow Set ℓ'

open HetroRel

relOp = HetroRel.Rel

syntax relOp A x y = x \langle A \rangle y

Then define the strcture-preserving operations,

record Hom $\{\ell \ell'\}$ (Src Tgt : HetroRel $\ell \ell'$) : Set $(\ell \sqcup \ell')$ **where**

constructor MkHom

field

one : One Src \rightarrow One Tgt

two : Two Src \rightarrow Two Tgt

shift : $\{x : \text{One Src}\} \{y : \text{Two Src}\} \rightarrow x \langle \text{Src} \rangle y \rightarrow \text{one } x \langle \text{Tgt} \rangle \text{two } y$

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

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

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

```
{F0           = λ A → MkHRel A ⊥ (λ { _ } ())
;F1           = λ f → MkHom f id (λ { {y = ()} })
;identity      = ≐-refl , ≐-refl
;homomorphism  = ≐-refl , ≐-refl
;F-resp≡       = λ f≈g → (λ x → f≈g {x}) , ≐-refl
}
```

-- (MkRel X ⊥ ⊥ → Alg) ≅ (X → One Alg)

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

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

```
{unit          = record
  {η = λ _ → id
  ; commute = λ _ → ≡.refl
  }
; counit = record
  {η = λ A → MkHom (λ z → z) (λ { () } ) (λ {x} { })
  ; commute = λ f → ≐-refl , (λ ())
  }
; zig      = ≐-refl , (λ ())
; zag      = ≡.refl
}
```

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

$\text{CoFree}^1 \ell = \text{record}$

```
{F0           = λ A → MkHRel A ⊤ (λ _ _ → A)
;F1           = λ f → MkHom f id f
;identity      = ≐-refl , ≐-refl
;homomorphism  = ≐-refl , ≐-refl
;F-resp≡       = λ f≈g → (λ x → f≈g {x}) , ≐-refl
}
```

-- (One Alg → X) ≅ (Alg → MkRel X ⊤ (λ _ _ → X))

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

$\text{Right}^1 \ell = \text{record}$

```
{unit          = record
  {η = λ _ → MkHom id (λ _ → tt) (λ {x} {y} _ → x)
  ; commute = λ _ → ≐-refl , (λ x → ≡.refl)
  }
; counit = record {η = λ _ → id; commute = λ _ → ≡.refl}
; zig     = ≡.refl
; zag     = ≐-refl , λ { tt → ≡.refl }
}
```

-- Another cofree functor:

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

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

```
{F0           = λ A → MkHRel A ⊤ (λ _ _ → ⊤)
;F1           = λ f → MkHom f id id
;identity      = ≐-refl , ≐-refl
;homomorphism  = ≐-refl , ≐-refl
```



```

;F-resp-≡ = λ f≈g → (λ x → f≈g {x}) , ≡-refl
}
-- (One Alg → X) ≅ (Alg → MkRel X ⊤ (λ _ → ⊤))
Right1' : (ℓ : Level) → Adjunction (Forget1 ℓ ℓ) (CoFree1' ℓ)
Right1' ℓ = record
{unit = record
  {η = λ _ → MkHom id (λ _ → tt) (λ {x} {y} _ → tt)
  ; commute = λ _ → ≡-refl , (λ x → ≡.refl)
  }
; counit = record {η = λ _ → id; commute = λ _ → ≡.refl}
; zig = ≡.refl
; zag = ≡-refl , λ {tt → ≡.refl}
}

```

But wait, adjoints are necessarily unique, up to isomorphism, whence $\text{CoFree}^1 \cong \text{Cofree}^{1'}$. 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, B) 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 B .

```

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

```

Notice that: The number of B steps is preserved, $[[B, B]] \circ \text{step}^n \doteq B^n \circ [[B, B]]$. Essentially, $[[B, B]] (\text{step}^n \text{base } x) \approx B^n B x$. 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' B' : A → A} {B B : B → B} {f : A → B}
  → (basis : B ∘ f ≐i f ∘ B')
  → (next : B ∘ f ≐i f ∘ B')
  → [ B , B ] ∘ map f ≐ f ∘ [ B' , B' ]
[ ]-naturality {B = B} basis next = elim basis (λ ind → ≡.cong B 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 Some

module Some **where**

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

open import Relation.Binary **using** (Setoid; IsEquivalence; Rel;
Reflexive; Symmetric; Transitive)

open import Function.Equality **using** (Π ; $_ \longrightarrow _$; id; $_ \circ _$; $_ \langle \$ \rangle _$)

open import Function **using** ($_ \$ _$) **renaming** (id to id₀; $_ \circ _$ to $_ \odot _$)

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 TypeEquiv **using** (swap₊)

open import SetoidSetoid

open import Relation.Binary.Sum

open import Relation.Binary.PropositionalEquality **using** (inspect)

[WK: Goal?]

11.1 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. **[]**

```

module  $\_ \{a \ell a\} \{A : Setoid\ a \ell a\} (P : A \longrightarrow SSetoid\ \ell a\ \ell a) \textbf{where}$ 
  open Setoid A
  private  $P_0 = \lambda e \rightarrow Setoid.Carrier\ (\Pi.\_ \$) \_ P\ e$ 
  data  $Some_0 : List\ Carrier \rightarrow Set\ (a \sqcup \ell a) \textbf{where}$ 
    here :  $\{x : Carrier\} \{xs : List\ Carrier\} (px : P_0\ x) \rightarrow Some_0\ (x :: xs)$ 
    there :  $\{x : Carrier\} \{xs : List\ Carrier\} (pxs : Some_0\ xs) \rightarrow Some_0\ (x :: xs)$ 

```

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

```

 $toN : \{xs : List\ Carrier\} \rightarrow Some_0\ xs \rightarrow \mathbb{N}$ 
 $toN\ (here\ \_) = 0$ 
 $toN\ (there\ pf) = suc\ (toN\ pf)$ 
 $\_ \sim S \_ : \{xs : List\ Carrier\} \rightarrow Some_0\ xs \rightarrow Some_0\ xs \rightarrow Set$ 
 $s_1 \sim S\ s_2 = toN\ s_1 \equiv toN\ s_2$ 

```

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

```

module  $\_ \{a \ell a\} \{A : Setoid\ a \ell a\} \{P : A \longrightarrow SSetoid\ \ell a\ \ell a\} \{Q : A \longrightarrow SSetoid\ \ell a\ \ell a\} \textbf{where}$ 
  open Setoid A
  private  $P_0 = \lambda e \rightarrow Setoid.Carrier\ (\Pi.\_ \$) \_ P\ e$ 
  private  $Q_0 = \lambda e \rightarrow Setoid.Carrier\ (\Pi.\_ \$) \_ Q\ e$ 
  infix 3  $\_ \approx \_$ 
  data  $\_ \approx \_ : \{xs\ ys : List\ Carrier\} (pf : Some_0\ P\ xs) (pf' : Some_0\ Q\ ys) \rightarrow Set\ \ell a \textbf{where}$ 
    hereEq :  $\{xs\ ys : List\ Carrier\} \{x\ y : Carrier\} (px : P_0\ x) (qy : Q_0\ y)$ 
       $\rightarrow \_ \approx \_ (here\ \{x = x\} \{xs\} px) (here\ \{x = y\} \{ys\} qy)$ 
    thereEq :  $\{xs\ ys : List\ Carrier\} \{x\ y : Carrier\} \{pxs : Some_0\ P\ xs\} \{qys : Some_0\ Q\ ys\}$ 
       $\rightarrow \_ \approx \_ pxs\ qys \rightarrow \_ \approx \_ (there\ \{x = x\} pxs) (there\ \{x = y\} qys)$ 

```

Notice that these another from of “natural numbers” whose elements are of the form $thereEq^n\ (hereEq\ P\ x\ Q\ x)$ for some $n : \mathbb{N}$.

```

module  $\_ \{a \ell a\} \{A : Setoid\ a \ell a\} \{P : A \longrightarrow SSetoid\ \ell a\ \ell a\} \textbf{where}$ 
  open Setoid A
   $\approx\text{-refl} : \{xs : List\ Carrier\} \{p : Some_0\ P\ xs\} \rightarrow p \approx p$ 
   $\approx\text{-refl}\ \{p = here\ px\} = hereEq\ px\ px$ 
   $\approx\text{-refl}\ \{p = there\ p\} = thereEq\ \approx\text{-refl}$ 
module  $\_ \{a \ell a\} \{A : Setoid\ a \ell a\} \{P : A \longrightarrow SSetoid\ \ell a\ \ell a\} \{Q : A \longrightarrow SSetoid\ \ell a\ \ell a\} \textbf{where}$ 
  open Setoid A
   $\approx\text{-sym} : \{xs : List\ Carrier\} \{p : Some_0\ P\ xs\} \{q : Some_0\ Q\ xs\} \rightarrow p \approx q \rightarrow q \approx p$ 
   $\approx\text{-sym}\ (hereEq\ px\ py) = hereEq\ py\ px$ 
   $\approx\text{-sym}\ (thereEq\ eq) = thereEq\ (\approx\text{-sym}\ eq)$ 
module  $\_ \{a \ell a\} \{A : Setoid\ a \ell a\} \{P : A \longrightarrow SSetoid\ \ell a\ \ell a\} \{Q : A \longrightarrow SSetoid\ \ell a\ \ell a\} \{R : A \longrightarrow SSetoid\ \ell a\ \ell a\} \textbf{where}$ 
  open Setoid A
   $\approx\text{-trans} : \{xs : List\ Carrier\} \{p : Some_0\ P\ xs\} \{q : Some_0\ Q\ xs\} \{r : Some_0\ R\ xs\}$ 

```

```

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

```

11.2 Membership module

```

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

```

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

```

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

```

11.3 Parallel Composition

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

```

data _||_ {a1 b1 c1 a2 b2 c2 : Level}
  {A1 : Set a1} {B1 : Set b1} (_~1_ : A1 → B1 → Set c1)
  {A2 : Set a2} {B2 : Set b2} (_~2_ : A2 → B2 → Set c2)

```

```

: A1 ⊔ A2 → B1 ⊔ B2 → Set (a1 ⊔ b1 ⊔ c1 ⊔ a2 ⊔ b2 ⊔ c2) where
left : {x : A1} {y : B1} (x ~1 y : x ~1 y) → ( _ ~1 _ || _ ~2 _ ) (inj1 x) (inj1 y)
right : {x : A2} {y : B2} (x ~2 y : x ~2 y) → ( _ ~1 _ || _ ~2 _ ) (inj2 x) (inj2 y)
-- Non-working “eliminator” for this type.
[ _ || _ ] : {a1 b1 c1 a2 b2 c2 ℓ : Level}
  {A1 : Set a1} {B1 : Set b1} { _ ~1 _ : A1 → B1 → Set c1 }
  {A2 : Set a2} {B2 : Set b2} { _ ~2 _ : A2 → B2 → Set c2 }
  →
    {Z : {a : A1 ⊔ A2} {b : B1 ⊔ B2} → ( _ ~1 _ || _ ~2 _ ) a b → Set ℓ}
    (F : {a : A1} {b : B1} (a ~1 b : a ~1 b) → Z (left a ~1 b))
    (G : {a : A2} {b : B2} (a ~2 b : a ~2 b) → Z (right a ~2 b))
  →
    {x : A1 ⊔ A2} {y : B1 ⊔ B2}
    → (x || y : ( _ ~1 _ || _ ~2 _ ) x y) → Z x || y
[ F || G ] (left x ~1 y) = F x ~1 y
[ F || G ] (right x ~2 y) = G x ~2 y
-- If the argument relations are symmetric then so is their parallel composition.
||-sym : {a a' c c' : Level} {A : Set a} { _ ~ _ : A → A → Set c }
  {A' : Set a'} { _ ~' _ : A' → A' → Set c' }
  (sym1 : {x y : A} → x ~ y → y ~ x) (sym2 : {x y : A'} → x ~' y → y ~' x)
  {x y : A ⊔ A'}
  →
    ( _ ~ _ || _ ~' _ ) x y → ( _ ~ _ || _ ~' _ ) y x
||-sym sym1 sym2 (left x ~1 y) = left (sym1 x ~1 y)
||-sym sym1 sym2 (right x ~2 y) = right (sym2 x ~2 y)
--
-- ought to be just: [ left ∘ sym1 || right ∘ sym2 ]
--
-- Instead, I can use, with much distasteful yellow,
-- ||-sym sym1 sym2 = [ (λ pf → left (sym1 pf)) || (λ pf → right (sym2 pf)) ]
infix 999 _ ⊔ ⊔ _
_ ⊔ ⊔ _ : {i1 i2 k1 k2 : Level} → Setoid i1 k1 → Setoid i2 k2 → Setoid (i1 ⊔ i2) (i1 ⊔ i2 ⊔ k1 ⊔ k2)
A ⊔ ⊔ B = record
  {Carrier = A0 ⊔ B0
  ; _ ≈ _ = ≈1 || ≈2
  ; isEquivalence = record
    { refl = λ { {inj1 x} → left refl1; {inj2 x} → right refl2 }
    ; sym = λ { (left eq) → left (sym1 eq); (right eq) → right (sym2 eq) }
      -- ought to be writable as [ left ∘ sym1 || right ∘ sym2 ]
    ; trans = λ { (left eq) (left eqq) → left (trans1 eq eqq)
      ; (right eq) (right eqq) → right (trans2 eq eqq)
      }
    }
  }
}
where
  open Setoid A renaming (Carrier to A0; _ ≈ _ to ≈1; refl to refl1; sym to sym1; trans to trans1)
  open Setoid B renaming (Carrier to B0; _ ≈ _ to ≈2; refl to refl2; sym to sym2; trans to trans2)

```

11.4 ⊔ ⊔-comm

```

⊔ ⊔-comm : {a b aℓ bℓ : Level} {A : Setoid a aℓ} {B : Setoid b bℓ} → A ⊔ ⊔ B ≅ B ⊔ ⊔ A
⊔ ⊔-comm {A = A} {B} = record
  {to = record { _ ($) _ = swap+; cong = swap-on-|| }
  ; from = record { _ ($) _ = swap+; cong = swap-on-||' }

```

```
; inverse-of = record {left-inverse-of = swap2≈||≈id; right-inverse-of = swap2≈||≈id'}
```

where

```
open Setoid A renaming (Carrier to A0; _~_ to ≈1; refl to refl1)
open Setoid B renaming (Carrier to B0; _~_ to ≈2; refl to refl2)
swap-on-|| : {i j : A0 ⊔ B0} → (≈1 || ≈2) i j → (≈2 || ≈1) (swap+ i) (swap+ j)
swap-on-|| (left x~1y) = right x~1y
swap-on-|| (right x~2y) = left x~2y
swap2≈||≈id : (z : A0 ⊔ B0) → (≈1 || ≈2) (swap+ (swap+ z)) z
swap2≈||≈id (inj1 _) = left refl1
swap2≈||≈id (inj2 _) = right refl2
{-Tried to obtain the following via ||-sym ... -}
swap-on-||' : {i j : B0 ⊔ A0} → (≈2 || ≈1) i j → (≈1 || ≈2) (swap+ i) (swap+ j)
swap-on-||' (left x~y) = right x~y
swap-on-||' (right x~y) = left x~y
swap2≈||≈id' : (z : B0 ⊔ A0) → (≈2 || ≈1) (swap+ (swap+ z)) z
swap2≈||≈id' (inj1 _) = left refl2
swap2≈||≈id' (inj2 _) = right refl1
```

11.5 $++\cong : \dots \rightarrow (\text{Some } P \text{ xs } \uplus \text{Some } P \text{ ys}) \cong \text{Some } P (xs + ys)$

```
module _ {a ℓa : Level} {A : Setoid a ℓa} {P : A → SSetoid ℓa ℓa} where
++≅ : {xs ys : List (Setoid.Carrier A)} → (Some P xs ⊔ Some P ys) ≅ Some P (xs + ys)
++≅ {xs} {ys} = record
{to = record {_⟨$⟩_ = ⊔→+++; cong = ⊔→+++-cong}
;from = record {_⟨$⟩_ = +++→⊔; cong = new-cong xs}
;inverse-of = record
{left-inverse-of = lefty xs
;right-inverse-of = righty xs
}
}
where
open Setoid A
_~_ = _~S_ P
_≈_ = _≈_ ; ~-refl = ≈-refl {P = P}
-- “ealier”
⊔→! : ∀ {ws zs} → Some0 P ws → Some0 P (ws + zs)
⊔→! (here p) = here p
⊔→! (there p) = there (⊔→! p)
```

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

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

1

```
yo : {xs : List Carrier} {x y : Some0 P xs} → x ~ y → ⊔→! x ~ ⊔→! y
yo (hereEq px py) = hereEq px py
```

```

yo (thereEq pf) = thereEq (yo pf)
  -- “later”
 $\mathfrak{u} \rightarrow^r : \forall \text{xs} \{ \text{ys} \} \rightarrow \text{Some}_0 \text{P} \text{ys} \rightarrow \text{Some}_0 \text{P} (\text{xs} + \text{ys})$ 
 $\mathfrak{u} \rightarrow^r [] \text{p} = \text{p}$ 
 $\mathfrak{u} \rightarrow^r (\text{x} :: \text{xs}) \text{p} = \text{there} (\mathfrak{u} \rightarrow^r \text{xs} \text{p})$ 
oy : (xs : List Carrier) {x y : Some0 P ys} → x ~ y →  $\mathfrak{u} \rightarrow^r \text{xs} \text{x} \sim \mathfrak{u} \rightarrow^r \text{xs} \text{y}$ 
oy [] pf = pf
oy (x :: xs) pf = thereEq (oy xs pf)
  -- Some0 is ++→ $\mathfrak{u}$ -homomorphic, in the second argument.
 $\mathfrak{u} \rightarrow ++ : \forall \{ \text{zs ws} \} \rightarrow (\text{Some}_0 \text{P} \text{zs} \mathfrak{u} \text{Some}_0 \text{P} \text{ws}) \rightarrow \text{Some}_0 \text{P} (\text{zs} + \text{ws})$ 
 $\mathfrak{u} \rightarrow ++ (\text{inj}_1 \text{x}) = \mathfrak{u} \rightarrow^1 \text{x}$ 
 $\mathfrak{u} \rightarrow ++ \{ \text{zs} \} (\text{inj}_2 \text{y}) = \mathfrak{u} \rightarrow^r \text{zs} \text{y}$ 
++→ $\mathfrak{u} : \forall \text{xs} \{ \text{ys} \} \rightarrow \text{Some}_0 \text{P} (\text{xs} + \text{ys}) \rightarrow \text{Some}_0 \text{P} \text{xs} \mathfrak{u} \text{Some}_0 \text{P} \text{ys}$ 
++→ $\mathfrak{u} [] \text{p} = \text{inj}_2 \text{p}$ 
++→ $\mathfrak{u} (\text{x} :: \text{l}) (\text{here} \text{p}) = \text{inj}_1 (\text{here} \text{p})$ 
++→ $\mathfrak{u} (\text{x} :: \text{l}) (\text{there} \text{p}) = (\text{there} \mathfrak{u}_1 \text{id}_0) (++\rightarrow \mathfrak{u} \text{l} \text{p})$ 
  -- all of the following may need to change
 $\mathfrak{u} \rightarrow ++\text{-cong} : \{ \text{a b} : \text{Some}_0 \text{P} \text{xs} \mathfrak{u} \text{Some}_0 \text{P} \text{ys} \} \rightarrow (\_ \sim \_ \parallel \_ \sim \_) \text{a} \text{b} \rightarrow \mathfrak{u} \rightarrow ++ \text{a} \sim \mathfrak{u} \rightarrow ++ \text{b}$ 
 $\mathfrak{u} \rightarrow ++\text{-cong} (\text{left } \text{x}_1 \sim \text{x}_2) = \text{yo } \text{x}_1 \sim \text{x}_2$ 
 $\mathfrak{u} \rightarrow ++\text{-cong} (\text{right } \text{y}_1 \sim \text{y}_2) = \text{oy } \text{xs} \text{y}_1 \sim \text{y}_2$ 
 $\sim \parallel \sim\text{-cong} : \{ \text{xs ys us vs} : \text{List Carrier} \}$ 
  → (F : Some0 P xs → Some0 P us) (F-cong : {p q : Some0 P xs} → p ~ q → F p ~ F q)
  → (G : Some0 P ys → Some0 P vs) (G-cong : {p q : Some0 P ys} → p ~ q → G p ~ G q)
  → {pf pf' : Some0 P xs  $\mathfrak{u}$  Some0 P ys}
  → ( $\_ \sim \_ \parallel \_ \sim \_$ ) pf pf' → ( $\_ \sim \_ \parallel \_ \sim \_$ ) ((F  $\mathfrak{u}_1$  G) pf) ((F  $\mathfrak{u}_1$  G) pf')
 $\sim \parallel \sim\text{-cong} \text{F} \text{F-cong} \text{G} \text{G-cong} (\text{left } \text{x} \sim_1 \text{y}) = \text{left} (\text{F-cong } \text{x} \sim_1 \text{y})$ 
 $\sim \parallel \sim\text{-cong} \text{F} \text{F-cong} \text{G} \text{G-cong} (\text{right } \text{x} \sim_2 \text{y}) = \text{right} (\text{G-cong } \text{x} \sim_2 \text{y})$ 
new-cong : (xs : List Carrier) {i j : Some0 P (xs + ys)} → i ~ j → ( $\_ \sim \_ \parallel \_ \sim \_$ ) (++→ $\mathfrak{u}$  xs i) (++→ $\mathfrak{u}$  xs j)
new-cong [] pf = right pf
new-cong (x :: xs) (hereEq px py) = left (hereEq px py)
new-cong (x :: xs) (thereEq pf) =  $\sim \parallel \sim\text{-cong}$  there thereEq id0 id0 (new-cong xs pf)
lefty : (xs {ys} : List Carrier) (p : Some0 P xs  $\mathfrak{u}$  Some0 P ys) → ( $\_ \sim \_ \parallel \_ \sim \_$ ) (++→ $\mathfrak{u}$  xs ( $\mathfrak{u} \rightarrow ++ \text{p}$ )) p
lefty [] (inj1 ())
lefty [] (inj2 p) = right  $\approx$ -refl
lefty (x :: xs) (inj1 (here px)) = left  $\sim$ -refl
lefty (x :: xs) {ys} (inj1 (there p)) with ++→ $\mathfrak{u}$  xs {ys} ( $\mathfrak{u} \rightarrow ++$  (inj1 p)) | lefty xs {ys} (inj1 p)
... | inj1 _ | (left x1 y) = left (thereEq x1 y)
... | inj2 _ | ()
lefty (z :: zs) {ws} (inj2 p) with ++→ $\mathfrak{u}$  zs {ws} ( $\mathfrak{u} \rightarrow ++$  {zs} (inj2 p)) | lefty zs (inj2 p)
... | inj1 x | ()
... | inj2 y | (right x2 y) = right x2 y
righty : (zs {ws} : List Carrier) (p : Some0 P (zs + ws)) → ( $\mathfrak{u} \rightarrow ++$  (++→ $\mathfrak{u}$  zs p)) ~ p
righty [] {ws} p =  $\sim$ -refl
righty (x :: zs) {ws} (here px) =  $\sim$ -refl
righty (x :: zs) {ws} (there p) with ++→ $\mathfrak{u}$  zs p | righty zs p
... | inj1 _ | res = thereEq res
... | inj2 _ | res = thereEq res

```

11.6 Bottom as a setoid

```

 $\perp\perp : \forall \{ \text{a} \ell \text{a} \} \rightarrow \text{Setoid} \text{a} \ell \text{a}$ 
 $\perp\perp \{ \text{a} \} \{ \ell \text{a} \} = \text{record}$ 

```



```

{Carrier =  $\perp$ 
;  $\_ \approx \_ = \lambda \_ \_ \rightarrow \top$ 
; isEquivalence = record {refl = tt; sym =  $\lambda \_ \rightarrow \text{tt}$ ; trans =  $\lambda \_ \_ \rightarrow \text{tt}$ }
}

```

module $_ \{a \ell a : \text{Level}\} \{A : \text{Setoid } a \ell a\} \{P : A \rightarrow \text{SSetoid } \ell a \ell a\}$ **where**

```

 $\perp \cong \text{Some} [] : \perp \perp \{a\} \{\ell a\} \cong \text{Some } P []$ 
 $\perp \cong \text{Some} [] = \text{record}$ 
  {to      = record  $\{ \_ \langle \$ \rangle \_ = \lambda \{()\}; \text{cong} = \lambda \{\{()\}\}$ 
; from    = record  $\{ \_ \langle \$ \rangle \_ = \lambda \{()\}; \text{cong} = \lambda \{\{()\}\}$ 
; inverse-of = record {left-inverse-of =  $\lambda \_ \rightarrow \text{tt}$ ; right-inverse-of =  $\lambda \{()\}$ }
}

```

11.7 $\text{map} \cong : \dots \rightarrow \text{Some } (P \circ f) \text{ xs} \cong \text{Some } P (\text{map } (_ \langle \$ \rangle _ f) \text{ xs})$

$\text{map} \cong : \forall \{a \ell a\} \{A B : \text{Setoid } a \ell a\} \{P : B \rightarrow \text{SSetoid } \ell a \ell a\} \{f : A \rightarrow B\} \{xs : \text{List } (\text{Setoid.Carrier } A)\} \rightarrow$

$\text{Some } (P \circ f) \text{ xs} \cong \text{Some } P (\text{map } (_ \langle \$ \rangle _ f) \text{ xs})$

$\text{map} \cong \{A = A\} \{B\} \{P\} \{f\} = \text{record}$

```

{to = record  $\{ \_ \langle \$ \rangle \_ = \text{map}^+; \text{cong} = \text{map}^+ \text{-cong}$ 
; from = record  $\{ \_ \langle \$ \rangle \_ = \text{map}^-; \text{cong} = \text{map}^- \text{-cong}$ 
; inverse-of = record {left-inverse-of =  $\text{map}^- \circ \text{map}^+$ ; right-inverse-of =  $\text{map}^+ \circ \text{map}^-$ }
}

```

where

$g = _ \langle \$ \rangle _ f$

$A_0 = \text{Setoid.Carrier } A$

$_ \sim _ = _ \approx _ \{P = P\}$

$\text{map}^+ : \{xs : \text{List } A_0\} \rightarrow \text{Some}_0 (P \circ f) \text{ xs} \rightarrow \text{Some}_0 P (\text{map } g \text{ xs})$

$\text{map}^+ (\text{here } p) = \text{here } p$

$\text{map}^+ (\text{there } p) = \text{there } \$ \text{ map}^+ p$

$\text{map}^- : \{xs : \text{List } A_0\} \rightarrow \text{Some}_0 P (\text{map } g \text{ xs}) \rightarrow \text{Some}_0 (P \circ f) \text{ xs}$

$\text{map}^- \{\} ()$

$\text{map}^- \{x :: xs\} (\text{here } p) = \text{here } p$

$\text{map}^- \{x :: xs\} (\text{there } p) = \text{there } (\text{map}^- \{xs = xs\} p)$

$\text{map}^+ \circ \text{map}^- : \{xs : \text{List } A_0\} \rightarrow (p : \text{Some}_0 P (\text{map } g \text{ xs})) \rightarrow \text{map}^+ (\text{map}^- p) \sim p$

$\text{map}^+ \circ \text{map}^- \{\} ()$

$\text{map}^+ \circ \text{map}^- \{x :: xs\} (\text{here } p) = \text{hereEq } p p$

$\text{map}^+ \circ \text{map}^- \{x :: xs\} (\text{there } p) = \text{thereEq } (\text{map}^+ \circ \text{map}^- p)$

$\text{map}^- \circ \text{map}^+ : \{xs : \text{List } A_0\} \rightarrow (p : \text{Some}_0 (P \circ f) \text{ xs})$

$\rightarrow \text{let } _ \sim_2 _ = _ \approx _ \{P = P \circ f\} \text{ in } \text{map}^- (\text{map}^+ p) \sim_2 p$

$\text{map}^- \circ \text{map}^+ \{\} ()$

$\text{map}^- \circ \text{map}^+ \{x :: xs\} (\text{here } p) = \text{hereEq } p p$

$\text{map}^- \circ \text{map}^+ \{x :: xs\} (\text{there } p) = \text{thereEq } (\text{map}^- \circ \text{map}^+ p)$

$\text{map}^+ \text{-cong} : \{ys : \text{List } A_0\} \{i j : \text{Some}_0 (P \circ f) \text{ ys}\} \rightarrow _ \approx _ \{P = P \circ f\} i j \rightarrow \text{map}^+ i \sim \text{map}^+ j$

$\text{map}^+ \text{-cong} (\text{hereEq } px py) = \text{hereEq } px py$

$\text{map}^+ \text{-cong} (\text{thereEq } i \sim j) = \text{thereEq } (\text{map}^+ \text{-cong } i \sim j)$

$\text{map}^- \text{-cong} : \{ys : \text{List } A_0\} \{i j : \text{Some}_0 P (\text{map } g \text{ ys})\} \rightarrow i \sim j \rightarrow _ \approx _ \{P = P \circ f\} (\text{map}^- i) (\text{map}^- j)$

$\text{map}^- \text{-cong} \{\} ()$

$\text{map}^- \text{-cong} \{x :: ys\} (\text{hereEq } px py) = \text{hereEq } px py$

$\text{map}^- \text{-cong} \{x :: ys\} (\text{thereEq } i \sim j) = \text{thereEq } (\text{map}^- \text{-cong } i \sim j)$

module FindLose $\{a \ell a : \text{Level}\} \{A : \text{Setoid } a \ell a\} (P : A \rightarrow \text{SSetoid } \ell a \ell a)$ **where**

open Membership A

open Setoid A

open Π

open \cong

private

$P_0 = \lambda e \rightarrow \text{Setoid.Carrier } (\Pi. _ \$) _ P e$

$\text{Support} = \lambda ys \rightarrow \Sigma y : \text{Carrier} \bullet y \in_0 ys \times P_0 y$

$\text{find} : \{ys : \text{List Carrier}\} \rightarrow \text{Some}_0 P ys \rightarrow \text{Support } ys$

$\text{find } \{y :: ys\} (\text{here } p) = y, \text{here refl}, p$

$\text{find } \{y :: ys\} (\text{there } p) = \text{let } (a, a \in ys, Pa) = \text{find } p$
 $\text{in } a, \text{there } a \in ys, Pa$

$\text{lose} : \{ys : \text{List Carrier}\} \rightarrow \text{Support } ys \rightarrow \text{Some}_0 P ys$

$\text{lose } (y, \text{here } py, Py) = \text{here } (_ \cong _. \text{to } (\Pi. \text{cong } P py) \Pi. \$) Py$

$\text{lose } (y, \text{there } y \in ys, Py) = \text{there } (\text{lose } (y, y \in ys, Py))$

-- "If an element of ys has a property P, then some element of ys has property P"

-- cf copy below

$\text{Some-Intro} : \{y : \text{Carrier}\} \{ys : \text{List Carrier}\}$

$\rightarrow y \in_0 ys \rightarrow P_0 y \rightarrow \text{Some}_0 P ys$

$\text{Some-Intro } \{y\} y \in ys Qy = \text{lose } (y, y \in ys, Qy)$

$\text{bag-as} \Rightarrow : \{xs ys : \text{List Carrier}\} \rightarrow \text{BagEq } xs ys \rightarrow \text{Some}_0 P xs \rightarrow \text{Some}_0 P ys$

$\text{bag-as} \Rightarrow xs \cong ys Pxs = \text{let } (x, x \in xs, Px) = \text{find } Pxs \text{ in}$

$\text{let } x \in ys = \text{to } xs \cong ys \$ x \in xs$

$\text{in } \text{lose } (x, x \in ys, Px)$

module FindLoseCong $\{a \ell a : \text{Level}\} \{A : \text{Setoid } a \ell a\} \{P : A \longrightarrow \text{SSetoid } \ell a \ell a\} \{Q : A \longrightarrow \text{SSetoid } \ell a \ell a\} \text{ where}$

open Membership A

open Setoid A

private

$P_0 = \lambda e \rightarrow \text{Setoid.Carrier } (\Pi. _ \$) _ P e$

$Q_0 = \lambda e \rightarrow \text{Setoid.Carrier } (\Pi. _ \$) _ Q e$

$\text{PSupport} = \lambda ys \rightarrow \Sigma y : \text{Carrier} \bullet y \in_0 ys \times P_0 y$

$\text{QSupport} = \lambda ys \rightarrow \Sigma y : \text{Carrier} \bullet y \in_0 ys \times Q_0 y$

$_ \approx _ : \{xs ys : \text{List Carrier}\} \rightarrow \text{PSupport } xs \rightarrow \text{QSupport } ys \rightarrow \text{Set } \ell a$

$(a, a \in xs, Pa) \approx (b, b \in ys, Qb) = a \approx b \times a \in xs \approx b \in ys$

open FindLose

$\text{find-cong} : \{ys : \text{List Carrier}\} \{p : \text{Some}_0 P ys\} \{q : \text{Some}_0 Q ys\} \rightarrow p \approx q \rightarrow \text{find } P p \approx \text{find } Q q$

$\text{find-cong } (\text{hereEq } px qy) = \text{refl}, \approx\text{-refl}$

$\text{find-cong } (\text{thereEq } eq) = \text{let } (\text{fst}, \text{snd}) = \text{find-cong } eq \text{ in } \text{fst}, \text{thereEq } \text{snd}$

private

$P^+ : \{x y : \text{Carrier}\} \rightarrow x \approx y \rightarrow P_0 x \rightarrow P_0 y$

$P^+ x \approx y = \Pi. _ \$ _ (_ \cong _. \text{to } (\Pi. \text{cong } P x \approx y))$

$Q^+ : \{x y : \text{Carrier}\} \rightarrow x \approx y \rightarrow Q_0 x \rightarrow Q_0 y$

$Q^+ x \approx y = \Pi. _ \$ _ (_ \cong _. \text{to } (\Pi. \text{cong } Q x \approx y))$

$\text{lose-cong} : \{xs ys : \text{List Carrier}\} \{p : \text{PSupport } xs\} \{q : \text{QSupport } ys\} \rightarrow p \approx q \rightarrow \text{lose } P p \approx \text{lose } Q q$

$\text{lose-cong } \{p = a, \text{here } a \approx x, Pa\} \{b, \text{here } b \approx x, Qb\} (\text{fst}, \text{hereEq } .a \approx x .b \approx x) = \text{hereEq } (P^+ a \approx x Pa) (Q^+ b \approx x Qb)$

$\text{lose-cong } \{p = a, \text{here } a \approx x, Pa\} \{b, \text{there } b \in ys, Qb\} (\text{fst}, ())$

$\text{lose-cong } \{p = a, \text{there } a \in xs, Pa\} \{b, \text{here } px, Qb\} (\text{fst}, ())$

$\text{lose-cong } \{p = a, \text{there } a \in xs, Pa\} \{b, \text{there } b \in ys, Qb\} (a \approx b, \text{thereEq } a \in xs \approx b \in ys) = \text{thereEq } (\text{lose-cong } (a \approx b, a \in xs \approx b \in ys))$

$\text{cong-fwd} : \{xs ys : \text{List Carrier}\} \{xs \cong ys : \text{BagEq } xs ys\} \{p : \text{Some}_0 P xs\} \{q : \text{Some}_0 Q xs\}$

$\rightarrow p \approx q \rightarrow \text{bag-as} \Rightarrow P xs \cong ys p \approx \text{bag-as} \Rightarrow Q xs \cong ys q$

$\text{cong-fwd } \{xs\} \{ys\} \{xs \cong ys\} \{p\} \{q\} p \approx q \text{ with } \text{find } P p \mid \text{find } Q q \mid \text{find-cong } p \approx q$

$\dots \mid (x, x \in xs, px) \mid (y, y \in xs, py) \mid (x \approx y, x \in xs \approx y \in xs) = \text{lose-cong } (x \approx y, \text{goal})$

where

open \cong $(xs \cong ys \{x\})$ **using** $()$ **renaming** (to to F)

open \cong $(xs \cong ys \{y\})$ **using** $()$ **renaming** (to to G)

```

F-cong : {a b : x ∈0 xs} → a ≈ b → F ⟨$⟩ a ≈ F ⟨$⟩ b
F-cong = Π.cong F
G-cong : {a b : y ∈0 xs} → a ≈ b → G ⟨$⟩ a ≈ G ⟨$⟩ b
G-cong = Π.cong G
To = λ {i} → Π. _⟨$⟩_ ( _≅_ .to (xs≅ys {i}))
postulate helper : {i j : Carrier} → i ≈ j → {!To {i} ≡ To {j} !}
  -- switch to john major equality in the defn of ≡ ?
goal : F ⟨$⟩ x∈xs ≈ G ⟨$⟩ y∈xs
goal = {!Π.cong F!}
y∈ysT : y ∈0 xs
y∈ysT = y∈xs

```

[Somebody: *Commented out:*

```

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

```

]

11.8 Some-cong and holes

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

```

module _ {a ℓa : Level} {A : Setoid a ℓa} {P : A → SSetoid ℓa ℓa} where
open Membership A
open Setoid A
private
  P0 = λ e → Setoid.Carrier (Π. _⟨$⟩_ P e)
  Support = λ ys → Σ y : Carrier • y ∈0 ys × P0 y
  _ ≈ _ : {ys : List Carrier} → Support ys → Support ys → Set ℓa
  (a, a∈xs, Pa) ≈ (b, b∈xs, Pb) = a ≈ b × a∈xs ≈ b∈xs
  Σ-Setoid : (ys : List Carrier) → Setoid (ℓa ⊔ a) ℓa
  Σ-Setoid ys = record
    {Carrier = Support ys
    ; _ ≈ _ = _ ≈ _
    ; isEquivalence = record
      {refl = λ {s} → Refl {s}
      ; sym = λ {s} {t} eq → Sym {s} {t} eq
      ; trans = λ {s} {t} {u} a b → Trans {s} {t} {u} a b
      }
    }
where
  Refl : Reflexive _ ≈ _
  Refl {a, a∈xs, Pa} = refl, ≈-refl
  Sym : Symmetric _ ≈ _
  Sym (a≈b, a∈xs≈b∈xs) = sym a≈b, ≈-sym a∈xs≈b∈xs
  Trans : Transitive _ ≈ _
  Trans (a≈b, a∈xs≈b∈xs) (b≈c, b∈xs≈c∈xs) = trans a≈b b≈c, ≈-trans a∈xs≈b∈xs b∈xs≈c∈xs
module ≈ {ys} where open Setoid (Σ-Setoid ys) public
open FindLose P
open FindLoseCong hiding ( _ ≈ _ )

```

```

left-inv : {ys : List Carrier} (xεys : Some0 P ys) → lose (find xεys) ≈ xεys
left-inv (here px) = hereEq _ px
left-inv (there xεys) = thereEq (left-inv xεys)

right-inv : {ys : List Carrier} (pf : Σ y : Carrier • y ∈0 ys × P0 y) → find (lose pf) ≈ pf
right-inv (y, here px, Py) = (sym px), (hereEq refl px)
right-inv (y, there yεys, Py) = (proj1 (right-inv (y, yεys, Py))), (thereEq (proj2 (right-inv (y, yεys, Py))))

Σ-Some : (xs : List Carrier) → Some P xs ≅ Σ-Setoid xs
Σ-Some xs = record
  {to = record {_$}_ = find {xs}; cong = find-cong}
  ; from = record {_$}_ = lose; cong = lose-cong}
  ; inverse-of = record
    {left-inverse-of = left-inv
     ; right-inverse-of = right-inv
    }
  }

```

module _ {a ℓa : Level} {A : Setoid a ℓa } {P : A → SSetoid ℓa ℓa } **where**

open Membership A

open Setoid A

private P₀ = λ e → Setoid.Carrier (Π. _(\$)_ P e)

Some-cong : {xs₁ xs₂ : List Carrier} →

(∀ {x} → (x ∈ xs₁) ≅ (x ∈ xs₂)) →

Some P xs₁ ≅ Some P xs₂

Some-cong {xs₁} {xs₂} list-rel = **record**

{to = **record** {_\$}_ = bag-as⇒ list-rel; cong = FindLoseCong.cong-fwd {P = P} {Q = P}} -- Yellow! ???

; from = **record** {_\$}_ = xs₁→xs₂ (≅-sym list-rel); cong = {! {- ??? -} !}}

; inverse-of = **record** {left-inverse-of = {! {- ??? -} !}; right-inverse-of = {! {- ??? -} !}}

}

where

open FindLose P **using** (bag-as⇒; find)

-- this is probably a specialized version of Respects.

-- is also related to an uncurried version of 'lose'.

copy : ∀ {x} {ys} {Q : A → SSetoid ℓa ℓa } → x ∈₀ ys → (Setoid.Carrier (Π. _(\$)_ Q x)) → Some₀ Q ys

copy {Q = Q} (here p) pf = here (_≅_.to (Π.cong Q p) (\$) pf)

copy (there p) pf = there (copy p pf)

-- this should be generalized to qy coming from Q₀ x.

copy-cong : {x y : Carrier} {xs ys : List Carrier} {Q : A → SSetoid ℓa ℓa }

(px : P₀ x) (qy : Setoid.Carrier (Π. _(\$)_ Q y)) (x_{εxs} : x ∈₀ xs) (y_{εys} : y ∈₀ ys) →

(x_{εxs} ≈ y_{εys}) → copy {Q = P} x_{εxs} px ≈ copy {Q = Q} y_{εys} qy

copy-cong px qy₁ (here px₁) ∘ (here qy) (hereEq .px₁ qy) = hereEq _ _

copy-cong px qy (there i) ∘ (there _) (thereEq i≈j) = thereEq (copy-cong px qy _ _ i≈j)

xs₁→xs₂ : ∀ {xs ys} → (∀ {x} → (x ∈ xs) ≅ (x ∈ ys)) → Some₀ P xs → Some₀ P ys

xs₁→xs₂ {xs} rel p =

let pos = find {ys = xs} p in

copy (_≅_.to rel (\$) proj₁ (proj₂ pos)) (proj₂ (proj₂ pos))

cong-fwd : {i j : Some₀ P xs₁} →

i ≈ j → xs₁→xs₂ list-rel i ≈ xs₁→xs₂ list-rel j

cong-fwd {i} {j} i≈j = copy-cong _ _ _ _ {! {- ??? -} !}

12 Conclusion and Outlook

???