

# Theories and Data Structures

Jacques Carette, Musa Al-hassy, Wolfram Kahl

June 12, 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

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Overview</b>	<b>3</b>
<b>3</b>	<b>Obtaining Forgetful Functors</b>	<b>3</b>
<b>4</b>	<b>Equality Combinators</b>	<b>4</b>
4.1	Propositional Equality . . . . .	4
4.2	Function Extensionality . . . . .	5
4.3	Equiv . . . . .	5
4.4	Making symmetry calls less intrusive . . . . .	5
<b>5</b>	<b>Properties of Sums and Products</b>	<b>6</b>
5.1	Generalised Bot and Top . . . . .	6
5.2	Sums . . . . .	7
5.3	Products . . . . .	7
<b>6</b>	<b>Two Sorted Structures</b>	<b>7</b>
6.1	Definitions . . . . .	8
6.2	Category and Forgetful Functors . . . . .	8
6.3	Free and CoFree . . . . .	9
6.4	Adjunction Proofs . . . . .	10
6.5	Merging is adjoint to duplication . . . . .	11
6.6	Duplication also has a left adjoint . . . . .	11
<b>7</b>	<b>Binary Heterogeneous Relations</b> — <span style="border: 1px solid black; padding: 0 2px;">[ MA:</span> <i>What named data structure do these correspond to in programming?</i> <span style="border: 1px solid black; padding: 0 2px;">]</span>	<b>12</b>
7.1	Definitions . . . . .	12
7.2	Category and Forgetful Functors . . . . .	13
7.3	Free and CoFree Functors . . . . .	14
7.4	<span style="border: 1px solid black; padding: 0 2px;">???</span> . . . . .	17
<b>8</b>	<b>Pointed Algebras: Nullable Types</b>	<b>18</b>
8.1	Definition . . . . .	19
8.2	Category and Forgetful Functors . . . . .	19
8.3	A Free Construction . . . . .	20
<b>9</b>	<b>Conclusion and Outlook</b>	<b>21</b>

# 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
  ; _∘_      = 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) ∘ mor F   ≈⟨ o-≐-cong1 _ comp-is-o ⟩
    mor H ∘ mor G ∘ mor F     ≈⟨ o-≐-cong2 (mor H) comp-is-o ⟩
    mor H ∘ mor (comp G F)   ≈⟨ comp-is-o ⟩
    mor (comp H (comp G F)) ■
  ; identityl = λ {f = f} → comp-is-o ⟨ ≐ ⟩ Id-is-id ∘ mor f
  ; identityr = λ {f = f} → comp-is-o ⟨ ≐ ⟩ ≡.cong (mor f) ∘ 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 ( cong1 to o-≐-cong2; cong2 to o-≐-cong1 )
infix 5 _⟨≐≐⟩_
_⟨≐≐⟩_ = ≐-trans

```

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

## 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 _⊔_)

```

### 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
  ⊥-elim ()
record ⊤ {ℓ : Level} : Set ℓ where
  constructor tt

```

## 5.2 Sums

Just as  $\_ \uplus \_$  takes types to types, its “map” variant  $\_ \uplus_1 \_$  takes functions to functions and is a functorial congruence: It preserves identity, distributes over composition, and preserves extensional equality.

```

 $\uplus\text{-id} : \{a\ b : \text{Level}\} \{A : \text{Set } a\} \{B : \text{Set } b\} \rightarrow \text{id} \uplus_1 \text{id} \doteq \text{id} \{A = A \uplus B\}$ 
 $\uplus\text{-id} = [ \doteq\text{-refl} , \doteq\text{-refl} ]$ 

 $\uplus\text{-o} : \{a\ b\ c\ a' \ b' \ c' : \text{Level}\}$ 
   $\{A : \text{Set } a\} \{A' : \text{Set } a'\} \{B : \text{Set } b\} \{B' : \text{Set } b'\} \{C : \text{Set } c\} \{C' : \text{Set } c'\}$ 
   $\{f : A \rightarrow A'\} \{g : B \rightarrow B'\} \{f' : A' \rightarrow C\} \{g' : B' \rightarrow C'\}$ 
   $\rightarrow (f' \circ f) \uplus_1 (g' \circ g) \doteq (f' \uplus_1 g') \circ (f \uplus_1 g) \quad \text{-- aka “the exchange rule for sums”}$ 
 $\uplus\text{-o} = [ \doteq\text{-refl} , \doteq\text{-refl} ]$ 

 $\uplus\text{-cong} : \{a\ b\ c\ d : \text{Level}\} \{A : \text{Set } a\} \{B : \text{Set } b\} \{C : \text{Set } c\} \{D : \text{Set } d\} \{f\ f' : A \rightarrow C\} \{g\ g' : B \rightarrow D\}$ 
   $\rightarrow f \doteq f' \rightarrow g \doteq g' \rightarrow f \uplus_1 g \doteq f' \uplus_1 g'$ 
 $\uplus\text{-cong } f \approx f' \ g \approx g' = [ \text{o-}\doteq\text{-cong}_2 \text{ inj}_1 \ f \approx f' , \text{o-}\doteq\text{-cong}_2 \text{ inj}_2 \ g \approx g' ]$ 

```

Composition post-distributes into casing,

```

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

```

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  $D\ A = 2 \times A = A \uplus A$ :

```

 $\text{from}\uplus : \{\ell : \text{Level}\} \{A : \text{Set } \ell\} \rightarrow A \uplus A \rightarrow A$ 
 $\text{from}\uplus = [ \text{id} , \text{id} ]$ 
  -- from $\uplus$  is a natural transformation
  --
 $\text{from}\uplus\text{-nat} : \{a\ b : \text{Level}\} \{A : \text{Set } a\} \{B : \text{Set } b\} \{f : A \rightarrow B\} \rightarrow f \circ \text{from}\uplus \doteq \text{from}\uplus \circ (f \uplus_1 f)$ 
 $\text{from}\uplus\text{-nat} = [ \doteq\text{-refl} , \doteq\text{-refl} ]$ 
  -- from $\uplus$  is injective and so is pre-invertible,
  --
 $\text{from}\uplus\text{-preInverse} : \{a\ b : \text{Level}\} \{A : \text{Set } a\} \{B : \text{Set } b\} \rightarrow \text{id} \doteq \text{from}\uplus \{A = A \uplus B\} \circ (\text{inj}_1 \uplus_1 \text{inj}_2)$ 
 $\text{from}\uplus\text{-preInverse} = [ \doteq\text{-refl} , \doteq\text{-refl} ]$ 

```

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

## 5.3 Products

Dual to  $\text{from}\uplus$ , a natural transformation  $2 \times \_ \rightarrow \mathbf{I}$ , is  $\text{diag}$ , 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?

] ]

## 6 Two Sorted Structures

So far we have been considering algebraic structures with only one underlying carrier set, however programmers are faced with a variety of different types at the same time, and the graph structure between them, and so we

consider briefly consider two sorted structures by starting the simplest possible case: Two type and no required interaction whatsoever between them.

```

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

```

## 6.1 Definitions

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

```

record TwoSorted  $\ell$  : 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

```

## 6.2 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    = 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.

**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}
}

```

**Forget**<sup>2</sup> : (ℓ : Level) → Functor (Twos ℓ) (Sets ℓ)

**Forget**<sup>2</sup> ℓ = **record**

```

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

```

### 6.3 Free and CoFree

Given a type, we can pair it with the empty type or the 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 )

**Free**<sup>2</sup> : (ℓ : Level) → Functor (Sets ℓ) (Twos ℓ)

```

Free2 ℓ = record
  {F0          = MkTwo ⊥
  ;F1          = MkHom id
  ;identity     = ≐-refl , ≐-refl
  ;homomorphism = ≐-refl , ≐-refl
  ;F-resp≡     = λ f≈g → ≐-refl , λ x → f≈g {x}
  }

Cofree2 : (ℓ : Level) → Functor (Sets ℓ) (Twos ℓ)
Cofree2 ℓ = record
  {F0          = MkTwo ⊤
  ;F1          = MkHom id
  ;identity     = ≐-refl , ≐-refl
  ;homomorphism = ≐-refl , ≐-refl
  ;F-resp≡     = λ f≈g → ≐-refl , λ x → f≈g {x}
  }

```

## 6.4 Adjunction Proofs

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

Left : (ℓ : Level) → Adjunction (Free ℓ) (Forget ℓ)

```

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

```

Right : (ℓ : Level) → Adjunction (Forget ℓ) (Cofree ℓ)

```

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

```

-- Dually,

Left<sup>2</sup> : (ℓ : Level) → Adjunction (Free<sup>2</sup> ℓ) (Forget<sup>2</sup> ℓ)

```

Left2 ℓ = record
  {unit = record
    {η = λ _ → id
    ; commute = λ _ → ≡.refl
    }
  ; counit = record
    {η = λ _ → MkHom (λ {()}) id
    ; commute = λ f → (λ {()}) , ≐-refl
    }
  }

```

```

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

```

## 6.5 Merging is adjoint to duplication

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

-- The category of Sets has products and so the **TwoSorted** type can be reified there.

```

Merge : (ℓ : Level) → Functor (Twos ℓ) (Sets ℓ)
Merge ℓ = record
{ F0      = λ S → One S × Two S
; F1      = λ F → one F ×1 two F
; identity = ≡.refl
; homomorphism = ≡.refl
; F-resp≡ = λ {(F≈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
}

```

## 6.6 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 : ( $\ell$  : Level)  $\rightarrow$  Functor (Twos  $\ell$ ) (Sets  $\ell$ )

Choice  $\ell$  = **record**

```
{F0          =  $\lambda$  S  $\rightarrow$  One S  $\uplus$  Two S
;F1          =  $\lambda$  F  $\rightarrow$  one F  $\uplus_1$  two F
;identity     =  $\uplus$ -id $i
;homomorphism =  $\lambda$  {x = x}  $\rightarrow$   $\uplus$ -o x}
;F-resp- $\equiv$    =  $\lambda$  F  $\approx$  G {x}  $\rightarrow$  uncurry  $\uplus$ -cong F  $\approx$  G x
}
```

Left<sub>2</sub> : ( $\ell$  : Level)  $\rightarrow$  Adjunction (Choice  $\ell$ ) (Dup  $\ell$ )

Left<sub>2</sub>  $\ell$  = **record**

```
{unit        = record { $\eta$  =  $\lambda$  _  $\rightarrow$  MkHom inj1 inj2; commute =  $\lambda$  _  $\rightarrow$   $\dot{=}$ -refl ,  $\dot{=}$ -refl}
;counit      = record { $\eta$  =  $\lambda$  _  $\rightarrow$  from $\uplus$ ; commute =  $\lambda$  _ {x}  $\rightarrow$  ( $\equiv$ .sym  $\circ$  from $\uplus$ -nat) x}
;zig         =  $\lambda$  { {-} } {x}  $\rightarrow$  from $\uplus$ -preInverse x}
;zag         =  $\dot{=}$ -refl ,  $\dot{=}$ -refl
}
```

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

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

**module** Structures.Rel **where**

**open import** Level **renaming** (suc to lsuc; zero to lzero;  $\_ \sqcup \_$  to  $\_ \uplus \_$ )

**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** ( $\_ \$ \_$ )

**open import** Forget

**open import** EqualityCombinators

**open import** DataProperties

**open import** Structures.TwoSorted **using** (TwoSorted; Twos; MkTwo) **renaming** (Hom to TwoHom; MkHom to MkTwoHom)

### 7.1 Definitions

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

**record** HetroRel  $\ell \ell'$  : Set (lsuc ( $\ell \uplus \ell'$ )) **where**

constructor MkHRel

**field**

One : Set  $\ell$

Two : Set  $\ell$

Rel : One  $\rightarrow$  Two  $\rightarrow$  Set  $\ell'$

**open** HetroRel

relOp = HetroRel.Rel

syntax relOp A  $\times$  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 → One Tgt
    two : Two Src → Two Tgt
    shift : {x : One Src} {y : Two Src} → x < Src > y → one x < Tgt > two y
open Hom

```

## 7.2 Category and Forgetful Functors

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

```

Rels : ( $\ell \ell'$  : Level) → Category (Isuc ( $\ell \sqcup \ell'$ )) ( $\ell \sqcup \ell'$ )  $\ell$ 
Rels  $\ell \ell'$  = record
  {Obj      = HetroRel  $\ell \ell'$ 
  ;  $\Rightarrow$  = Hom
  ;  $\equiv$     =  $\lambda F G \rightarrow$  one F  $\doteq$  one G  $\times$  two F  $\doteq$  two G
  ; id      = MkHom id id id
  ;  $\circ$      =  $\lambda F G \rightarrow$  MkHom (one F  $\circ$  one G) (two F  $\circ$  two G) (shift F  $\circ$  shift 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 =  $\lambda \{(oneEq_1, twoEq_1) (oneEq_2, twoEq_2) \rightarrow \doteq$ -trans oneEq1 oneEq2 ,  $\doteq$ -trans twoEq1 twoEq2}
    }
  ; o-resp $\equiv$  =  $\lambda \{(g_{\approx_1} k, g_{\approx_2} k) (f_{\approx_1} h, f_{\approx_2} h) \rightarrow$  o-resp $\doteq$   $g_{\approx_1} k f_{\approx_1} h$  , o-resp $\doteq$   $g_{\approx_2} k f_{\approx_2} h\}$ 
  }

```

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

```

Forget1 : ( $\ell \ell'$  : Level) → Functor (Rels  $\ell \ell'$ ) (Sets  $\ell$ )

```

```

Forget1  $\ell \ell'$  = record
  {F0      = HetroRel.One
  ; F1      = Hom.one
  ; identity =  $\equiv$ .refl
  ; homomorphism =  $\equiv$ .refl
  ; F-resp $\equiv$  =  $\lambda \{(F_{\approx_1} G, F_{\approx_2} G) \{x\} \rightarrow F_{\approx_1} G x\}$ 
  }

```

```

Forget2 : ( $\ell \ell'$  : Level) → Functor (Rels  $\ell \ell'$ ) (Sets  $\ell$ )

```

```

Forget2  $\ell \ell'$  = record
  {F0      = HetroRel.Two
  ; F1      = Hom.two
  ; identity =  $\equiv$ .refl
  ; homomorphism =  $\equiv$ .refl
  ; F-resp $\equiv$  =  $\lambda \{(F_{\approx_1} G, F_{\approx_2} G) \{x\} \rightarrow F_{\approx_2} G x\}$ 
  }

```

-- Whence, Rels is a subcategory of Twos

```

Forget3 : ( $\ell \ell'$  : Level) → Functor (Rels  $\ell \ell'$ ) (Twos  $\ell$ )

```

```

Forget3  $\ell \ell'$  = record
  {F0      =  $\lambda S \rightarrow$  MkTwo (One S) (Two S)
  ; F1      =  $\lambda F \rightarrow$  MkTwoHom (one F) (two F)

```

```

; identity      = ≐-refl , ≐-refl
; homomorphism = ≐-refl , ≐-refl
; F-resp-≡ = id
}

```

### 7.3 Free and CoFree Functors

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

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

```
Free1 : (ℓ ℓ' : Level) → Functor (Sets ℓ) (Rels ℓ ℓ')
```

```
Free1 ℓ ℓ' = record
  { F0      = λ A → MkHRel A ⊥ (λ { _ } ())
  ; F1      = λ f → MkHom f id (λ { { y = () } })
  ; identity = ≐-refl , ≐-refl
  ; homomorphism = ≐-refl , ≐-refl
  ; F-resp-≡ = λ f ≈ g → (λ x → f ≈ g { x }) , ≐-refl
  }

```

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

```
Left1 : (ℓ ℓ' : Level) → Adjunction (Free1 ℓ ℓ') (Forget1 ℓ ℓ')
```

```
Left1 ℓ ℓ' = record
  { unit = record
    { η = λ _ → id
    ; commute = λ _ → ≡.refl
    }
  ; counit = record
    { η = λ A → MkHom (λ z → z) (λ { { () } }) (λ { { x } } { })
    ; commute = λ f → ≐-refl , (λ { })
    }
  ; zig = ≐-refl , (λ { })
  ; zag = ≡.refl
  }

```

```
CoFree1 : (ℓ : Level) → Functor (Sets ℓ) (Rels ℓ ℓ)
```

```
CoFree1 ℓ = record
  { F0      = λ A → MkHRel A ⊤ (λ _ _ → A)
  ; F1      = λ f → MkHom f id f
  ; identity = ≐-refl , ≐-refl
  ; homomorphism = ≐-refl , ≐-refl
  ; F-resp-≡ = λ f ≈ g → (λ x → f ≈ g { x }) , ≐-refl
  }

```

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

```
Right1 : (ℓ : Level) → Adjunction (Forget1 ℓ ℓ) (CoFree1 ℓ)
```

```
Right1 ℓ = record
  { unit = record
    { η = λ _ → MkHom id (λ _ → tt) (λ { { x } } { { y } } _ → x)
    ; commute = λ _ → ≐-refl , (λ x → ≡.refl)
    }
  ; counit = record { η = λ _ → id; commute = λ _ → ≡.refl }
  }

```

```

; zig    = ≡.refl
; zag    = ≡-refl , λ {tt → ≡.refl}
}

-- Another cofree functor:
CoFree1' : (ℓ : Level) → Functor (Sets ℓ) (Rels ℓ ℓ)
CoFree1' ℓ = record
  {F0      = λ A → MkHRel A ⊤ (λ _ _ → ⊤)
; F1      = λ f → MkHom f id id
; identity  = ≡-refl , ≡-refl
; homomorphism = ≡-refl , ≡-refl
; F-resp≡ = λ f≈g → (λ x → f≈g {x}) , ≡-refl
}

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

```

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

```

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

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

```
{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 ⊤

$\text{Right}^2 : (\ell : \text{Level}) \rightarrow \text{Adjunction} (\text{Forget}^2 \ell \ell) (\text{CoFree}^2 \ell)$

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

```
{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

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

$\text{Free}^3 \ell = \text{record}$

```
{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)

$\text{Left}^3 : (\ell : \text{Level}) \rightarrow \text{Adjunction} (\text{Free}^3 \ell) (\text{Forget}^3 \ell \ell)$

$\text{Left}^3 \ell = \text{record}$

```
{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
}
```

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

$\text{CoFree}^3 \ell = \text{record}$

```
{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.

## 7.4 ???

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

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

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

-- The category of Sets has products and so the **TwoSorted** type can be reified there.

Merge : (ℓ : Level) → Functor (Twos ℓ) (Sets ℓ)

```

Merge  $\ell$  = record
  {F0          =  $\lambda S \rightarrow \text{One } S \times \text{Two } S$ 
  ;F1          =  $\lambda F \rightarrow \text{one } F \times_1 \text{two } F$ 
  ;identity     =  $\equiv.\text{refl}$ 
  ;homomorphism =  $\equiv.\text{refl}$ 
  ;F-resp $\equiv$  =  $\lambda \{(F \approx_1 G, F \approx_2 G) \{x, y\} \rightarrow \equiv.\text{cong}_2 \_ , \_ (F \approx_1 G \ x) (F \approx_2 G \ y)\}$ 
  }

```

-- Every set gives rise to its square as a TwoSorted type.

```

Dup : ( $\ell$  : Level)  $\rightarrow$  Functor (Sets  $\ell$ ) (Twos  $\ell$ )

```

```

Dup  $\ell$  = record
  {F0          =  $\lambda A \rightarrow \text{MkTwo } A \ A$ 
  ;F1          =  $\lambda f \rightarrow \text{MkHom } f \ f$ 
  ;identity     =  $\doteq\text{-refl}, \doteq\text{-refl}$ 
  ;homomorphism =  $\doteq\text{-refl}, \doteq\text{-refl}$ 
  ;F-resp $\equiv$  =  $\lambda F \approx G \rightarrow \text{diag } (\lambda \_ \rightarrow F \approx G)$ 
  }

```

Then the proof that these two form the desired adjunction

```

Right2 : ( $\ell$  : Level)  $\rightarrow$  Adjunction (Dup  $\ell$ ) (Merge  $\ell$ )

```

```

Right2  $\ell$  = record
  {unit      = record { $\eta$  =  $\lambda \_ \rightarrow \text{diag}$ ; commute =  $\lambda \_ \rightarrow \equiv.\text{refl}$ }
  ;counit    = record { $\eta$  =  $\lambda \_ \rightarrow \text{MkHom proj}_1 \ \text{proj}_2$ ; commute =  $\lambda \_ \rightarrow \doteq\text{-refl}, \doteq\text{-refl}$ }
  ;zig       =  $\doteq\text{-refl}, \doteq\text{-refl}$ 
  ;zag       =  $\equiv.\text{refl}$ 
  }

```

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 : ( $\ell$  : Level)  $\rightarrow$  Functor (Twos  $\ell$ ) (Sets  $\ell$ )

```

```

Choice  $\ell$  = record
  {F0          =  $\lambda S \rightarrow \text{One } S \uplus \text{Two } S$ 
  ;F1          =  $\lambda F \rightarrow \text{one } F \uplus_1 \text{two } F$ 
  ;identity     =  $\uplus\text{-id } \$_i$ 
  ;homomorphism =  $\lambda \{\{x = x\} \rightarrow \uplus\text{-o } x\}$ 
  ;F-resp $\equiv$  =  $\lambda F \approx G \{x\} \rightarrow \text{uncurry } \uplus\text{-cong } F \approx G \ x$ 
  }

```

```

Left2 : ( $\ell$  : Level)  $\rightarrow$  Adjunction (Choice  $\ell$ ) (Dup  $\ell$ )

```

```

Left2  $\ell$  = record
  {unit      = record { $\eta$  =  $\lambda \_ \rightarrow \text{MkHom inj}_1 \ \text{inj}_2$ ; commute =  $\lambda \_ \rightarrow \doteq\text{-refl}, \doteq\text{-refl}$ }
  ;counit    = record { $\eta$  =  $\lambda \_ \rightarrow \text{from}\uplus$ ; commute =  $\lambda \_ \{x\} \rightarrow (\equiv.\text{sym} \circ \text{from}\uplus\text{-nat}) \ x\}$ 
  ;zig       =  $\lambda \{\{-\} \{x\} \rightarrow \text{from}\uplus\text{-preInverse } x\}$ 
  ;zag       =  $\doteq\text{-refl}, \doteq\text{-refl}$ 
  }

```

## 8 Pointed Algebras: Nullable Types

We consider the theory of *pointed algebras* which consist of a type along with an elected value of that type.<sup>1</sup> 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.

<sup>1</sup>Note that this definition is phrased as a “dependent product”!

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

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

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

**module** Structures.Pointed **where**

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

**open import** Categories.Category **using** (Category; **module** Category)

**open import** Categories.Functor **using** (Functor)

**open import** Categories.Adjunction **using** (Adjunction)

**open import** Categories.NaturalTransformation **using** (NaturalTransformation)

**open import** Categories.Agda **using** (Sets)

**open import** Function **using** (id;  $\_ \circ \_$ )

**open import** Data.Maybe **using** (Maybe; just; nothing; maybe; maybe')

**open import** Forget

**open import** Data.Empty

**open import** Relation.Nullary

**open import** EqualityCombinators

## 8.1 Definition

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

**record** Pointed {a} : Set (lsuc a) **where**

  constructor MkPointed

**field**

    Carrier : Set a

    point : Carrier

**open** Pointed

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

**record** Hom {ℓ} (X Y : Pointed {ℓ}) : Set ℓ **where**

  constructor MkHom

**field**

    mor : Carrier X → Carrier Y

    preservation : mor (point X) ≡ point Y

**open** Hom

## 8.2 Category and Forgetful Functors

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

oneSortedAlg :  $\forall \{ \ell \} \rightarrow \text{OneSortedAlg } \ell$

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? **[ ]**

### 8.3 A Free Construction

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

```

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

```

Which is indeed deserving of its name:

```

MaybeLeft : (ℓ : Level) → Adjunction (Free ℓ) (Forget ℓ)
MaybeLeft ℓ = record
{unit      = record {η = λ _ → just; commute = λ _ → ≡.refl}
}

```

```

; counit      = record
  {  $\eta$         =  $\lambda X \rightarrow \text{MkHom } (\text{maybe id } (\text{point } X)) \equiv \text{refl}$ 
    ; commute = maybe  $\dot{=}$ -refl  $\circ \equiv \text{sym} \circ \text{preservation}$ 
  }
; zig         = maybe  $\dot{=}$ -refl  $\equiv \text{refl}$ 
; zag         =  $\equiv \text{refl}$ 
}

```

**[ 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 : {  $\ell$  : Level }  $\rightarrow$  (CoFree : Functor (Sets  $\ell$ ) (Pointeds  $\ell$ ))  $\rightarrow \neg$  (Adjunction (Forget  $\ell$ ) CoFree)
NoRight (record {  $F_0 = f$  }) Adjunct = lower ( $\eta$  (counit Adjunct) (Lift  $\perp$ ) (point (f (Lift  $\perp$ ))))
where open Adjunction
open NaturalTransformation

```

## 9 Conclusion and Outlook

???