# Theories and Data Structures

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

Jacques Carette, Musa Al-hassy, Wolfram Kahl

Spring 2018

**Abstract**

We aim to show how common data-structures naturally arise from elementary mathematical theories.
In particular, we answer the following questions:

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

unfinished ...
The Agda source code for this development is available on-line at the following URL:
`https://github.com/JacquesCarette/TheoriesAndDataStructures`

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

## Contents

```
module report where

open import Data.Nat
```

# Part I
# Variations on Sets

## 1 Two Sorted Structures ; HELLO WORLD

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.

%{{{ Imports

```
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
```

%}}}

%{{{ TwoSorted ; Hom

### 1.1 Definitions

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

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

open TwoSorted
```

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

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

```
    one : One Src → One Tgt
    two : Two Src → Two Tgt

  open Hom
```

%}}}
%{{{ TwoCat ; Forget

## 1.2 Category and Forgetful Functors

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

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

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

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

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


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

%}}}
%{{{ Free and CoFree

### 1.3 Free and CoFree

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

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

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

-- Dually, ( also shorter due to eta reduction )

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

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

%}}}
%{{{ Left and Right adjunctions

### 1.4 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
  }
```

%}}}

%{{{ Merge and Dup functors ; $\text{Right}_2$ adjunction

## 1.5 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
  }
```

%}}}
%{{{ Choice ; from⊎ ; Left₂ adjunction

## 1.6 Duplication also has a left adjoint

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

```
Choice : (ℓ : Level) → Functor (Twos ℓ) (Sets ℓ)
Choice ℓ = record
  { F₀             = λ S → One S ⊎ Two S
  ; F₁             = λ F → one F ⊎₁ two F
  ; identity       = ⊎-id $ᵢ
  ; homomorphism = λ{ {x = x} → ⊎-∘ x }
  ; F-resp-≡ = λ F≈G {x} → uncurry ⊎-cong F≈G x
  }

Left₂ : (ℓ : Level) → Adjunction (Choice ℓ) (Dup ℓ)
Left₂ ℓ = record
  { unit    = record { η = λ _ → MkHom inj₁ inj₂ ; commute = λ _ → ≐-refl , ≐-refl }
```

```
; counit  = record { η = λ _ → from⊎ ; commute = λ _ {x} → (≡.sym ∘ from⊎-nat) x }
; zig     = λ{ {_} {x} → from⊎-preInverse x }
; zag     = ≐-refl , ≐-refl
}
```

%}}}

% Quick Folding Instructions: % C-c C-s :: show/unfold region % C-c C-h :: hide/fold region % C-c C-w :: whole file fold % C-c C-o :: whole file unfold % % Local Variables: % folded-file: t % eval: (fold-set-marks "%{{{ " "%}}}") % eval: (fold-whole-buffer) % fold-internal-margins: 0 % end:

7472}

## 2 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]Note that this definition is phrased as a "dependent product"!} 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 defa`

edinsert{MA}{Haskell's typeclass analogue of |default|?}

edcomm{MA}{Perhaps discuss "types as values" and the subtle issue of how pointed algebras are completely different than classes in an imperative setting. }

%{{{ Imports

```
{-# OPTIONS --allow-unsolved-metas #-}

module Structures.Pointed where

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

open import Forget

open import Data.Empty
open import Relation.Nullary

open import EqualityCombinators
```

%}}}

%{{{ Pointed ; Hom

### 2.1 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
```

[1]{

```
    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
```

%}}}

%{{{ PointedAlg ; PointedCat ; Forget

## 2.2 Category and Forgetful Functors

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

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

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

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

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

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

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

```
  open import Data.Product
  Forget{ : (ℓ : Level) → Functor (Pointeds ℓ) (Sets ℓ)
  Forget{ 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) (≡.proc
; F-resp-≡ = λ {P} {Q} {F} {G} F≈G → λ{ {val , val≡ptP} → {!≡.cong₂ _,_ (F≈G val) ?!} }
}
```

That is, we "only remember the point".
edinsert{MA}{An adjoint to this functor?}
%}}}
%{{{ Free ; MaybeLeft ; NoRight

### 2.3 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
  }
```

edcomm{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
```

%}}}
% Quick Folding Instructions: % C-c C-s :: show/unfold region % C-c C-h :: hide/fold region % C-c C-w :: whole file fold % C-c C-o :: whole file unfold % % Local Variables: % folded-file: t % eval: (fold-set-marks "%{{{ " "%}}}") % eval: (fold-whole-buffer) % fold-internal-margins: 0 % end:

# Part II
# Helpers

8718}

## 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.

%{{{ Imports begin{ModuleHead}

```
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
```

end{ModuleHead} edcomm{MA}{For one reason or another, the module head is not making the imports smaller.}
%}}}

%{{{ OneSortedAlg

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

```
record OneSortedAlg (ℓ : Level) : Set (suc (suc ℓ)) where
  field
    Alg         : Set (suc ℓ)
    Carrier     : Alg → Set ℓ
    Hom         : Alg → Alg → Set ℓ
    mor         : {A B : Alg} → Hom A B → (Carrier A → Carrier B)
    comp        : {A B C : Alg} → Hom B C → Hom A B → Hom A C
    .comp-is-∘ : {A B C : Alg} {g : Hom B C} {f : Hom A B} → mor (comp g f) ≐ mor g ∘ mor f
    Id          : {A : Alg} → Hom A A
    .Id-is-id   : {A : Alg} → mor (Id {A}) ≐ id
```

%}}}

%{{{ oneSortedCategory

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

```
open import Relation.Binary.SetoidReasoning
oneSortedCategory : (ℓ : Level) → OneSortedAlg ℓ → Category (suc ℓ) ℓ ℓ
oneSortedCategory ℓ A = record
  { Obj   = Alg
  ; _⇒_  = Hom
  ; _≡_  = λ F G → mor F ≐ mor G
```

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

%}}}

%{{{ mkForgetful

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

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

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

% Quick Folding Instructions: % C-c C-s :: show/unfold region % C-c C-h :: hide/fold region % C-c C-w :: whole file fold % C-c C-o :: whole file unfold % % Local Variables: % folded-file: t % eval: (fold-set-marks "%{{{ " "%}}}") % eval: (fold-whole-buffer) % fold-internal-margins: 0 % end:

## 4 To Do

- include EqualityCombinators.lagda
- include DataProperties.lagda
- include RATH.lagda %% ! This module is not being called from anywhere ! June 9, 2017.

# Part III
# Unary Algebras

- include Structures/UnaryAlgebra.lagda
- include Structures/InvolutiveAlgebra.lagda
- include Structures/IndexedUnaryAlgebra.lagda

# Part IV
# Boom Hierarchy

- include Structures/Magma.lagda
- include Structures/Semigroup.lagda
- include Structures/Monoid.lagda
- include Structures/CommMonoid.lagda
- include Structures/CommMonoidTerm.lagda
- include Structures/AbelianGroup.lagda
- include Structures/Multiset.lagda

# Part V
# Setoids

- include SetoidEquiv.lagda
- include SetoidOfIsos.lagda
- include SetoidSetoid.lagda
- include SetoidFamilyUnion.lagda

# Part VI
# Equiv

- include Equiv.lagda
- include ISEquiv.lagda
- include TypeEquiv.lagda

# Part VII
# Misc

- include Function2.lagda
- include ParComp.lagda
- include Belongs.lagda
- include Some.lagda
- include CounterExample.lagda

## 5 Conclusion and Outlook

7472}

# 6 Pointed Algebras: Nullable Types

We consider the theory of *pointed algebras* which consist of a type along with an elected value of that type.[2]Note that this definition is phrased as a "dependent product"!} 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 defa`

edinsert{MA}{Haskell's typeclass analogue of |default|?}

edcomm{MA}{Perhaps discuss "types as values" and the subtle issue of how pointed algebras are completely different than classes in an imperative setting. }

%{{{ Imports

```
{-# OPTIONS --allow-unsolved-metas #-}

module Structures.Pointed where

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

open import Forget

open import Data.Empty
open import Relation.Nullary

open import EqualityCombinators
```

%}}}

%{{{ Pointed ; Hom

## 6.1 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
```

[2]{

```
    field
      mor          : Carrier X → Carrier Y
      preservation : mor (point X) ≡ point Y

  open Hom
```

%}}}
%{{{ PointedAlg ; PointedCat ; Forget

## 6.2 Category and Forgetful Functors

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

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

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

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

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

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

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

```
open import Data.Product
Forget{ : (ℓ : Level) → Functor (Pointeds ℓ) (Sets ℓ)
Forget{ 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) (≡.proo
  ; F-resp-≡ = λ {P} {Q} {F} {G} F≈G → λ{ {val , val≡ptP} → {!≡.cong₂ _,_ (F≈G val) ?!} }
  }
```

That is, we "only remember the point".

edinsert{MA}{An adjoint to this functor?}

%}}}

%{{{ Free ; MaybeLeft ; NoRight

### 6.3 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
  }
```

edcomm{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
```

%}}}

% Quick Folding Instructions: % C-c C-s :: show/unfold region % C-c C-h :: hide/fold region % C-c C-w :: whole file fold % C-c C-o :: whole file unfold % % Local Variables: % folded-file: t % eval: (fold-set-marks "%{{{ " "%}}}") % eval: (fold-whole-buffer) % fold-internal-margins: 0 % end: