Generic Programming

Release 1.0

Pierre-Évariste Dagand

Jan 22, 2019

CONTENTS

1 Extensional Generic Programming **3**
- 1.1 Functor . . . 4
- 1.2 Applicative . . . 5
- 1.3 Naperian . . . 7
- 1.4 Interlude: Monoid . . . 8
- 1.5 Foldable . . . 9
- 1.6 Traversable . . . 10
- 1.7 Dimension . . . 11
- 1.8 Multi-dimensional matrices . . . 13

2 Intensional Generic Programming **17**
- 2.1 Descriptions . . . 17
- 2.2 Fixpoint . . . 18
- 2.3 Induction . . . 19
- 2.4 Generic free monad . . . 20
- 2.5 Bootstrap . . . 21

3 Conclusion **23**

Extensional vs. intensional generic programming:

- extensional: meta-level support for accessing structures (intuition: type-classes)
- intensional: internalized code & interpretation (intuition: universe)

Extensional side of this lecture:

- (Lecture 1: Monad, MonadFail)
- Monoid
- Functor
- Applicative
- Foldable
- Traversable
- Want more? Typeclassopedia!

Intensional side of this lecture:

- reflecting inductive types
- internalized generic programming (over inductive types)

Vision: Generic programming is just programming

- Program with structure, one way (extensional) or another (intensional)
- Reflect the programming language in itself, one way (type-classes) or another (universe)

Takeaways:

- you will be *able* to spot the following structures: Monoid, Functor, Applicative, Monad, Foldable, Traversable
- you will be *able* to generalize a program to exploit any of the above structures
- you will be *able* to program in/with a universe of descriptions
- you will be *familiar* with Naperian functors
- you will be *familiar* with the notion of universe
- you will be *familiar* with "instance arguments"/"type classes"

CHAPTER

ONE

EXTENSIONAL GENERIC PROGRAMMING

Following Gibbons' APLicative Programming Naperian Functors, we are going to study the algebraic structure of "array-oriented programming language", à la APL (or any of its descendant, such as J). In effect, we shall build a *deep embedding* of a (small) subset of APL in Agda.

To do so, we will need a datastructure to represent multi-dimensional arrays, keeping track of (all of) their dimensions. We ought to be able to define:

```
-- 3-elements vectors:
v123 = C (S (1 :: 2 :: 3 :: []))
v456 = C (S (4 :: 5 :: 6 :: []))

-- 2x2 matrices:
v12-34 = C (C (S (((1 :: 2 :: []) ::
                   ((3 :: 4 :: []) :: [])))))

v56-78 = C (C (S (((5 :: 6 :: []) ::
                   ((7 :: 8 :: []) :: [])))))

-- 3x3 matrix:
v123-456-789 = C (C (S ((1 :: 2 :: 3 :: []) ::
                        (4 :: 5 :: 6 :: []) ::
                        (7 :: 8 :: 9 :: []) :: [])))

-- 3x2 matrix:
v12-45-78 = C (C (S ((1 :: 2 :: []) ::
                     (4 :: 5 :: []) ::
                     (7 :: 8 :: []) :: [])))

-- 2x3 matrix:
v123-456 = C (C (S ((1 :: 2 :: 3 :: []) ::
                    ((4 :: 5 :: 6 :: []) :: []))))

-- 2x2x2 matrix:
v1234-5678 = C (C (C (S (((1 :: 2 :: []) ::
                          ((3 :: 4 :: []) :: [])) ::
                         (((5 :: 6 :: []) ::
                          ((7 :: 8 :: []) :: [])) :: [])))))
```

Exploiting add-hoc polymorphism, we want to be able to apply unary operations pointwise to every element of a matrix, whatever its size:

```
square (S 3) ≡ S 9
square v123 ≡ C (S (1 :: 4 :: 9 :: []))
square v123-456-789
```

(continues on next page)

(continued from previous page)

```
  ≡ C (C (S ((1 :: 4 :: 9 :: []) ::
             (16 :: 25 :: 36 :: []) ::
             (49 :: 64 :: 81 :: []) :: [])))
square v12-45-78
  ≡ C (C (S(( 1 :: 4 :: []) ::
             (16 :: 25 :: []) ::
             (49 :: 64 :: []) :: [])))
square v1234-5678
  ≡ C (C (C (S (((1  ::  4 :: []) ::
                 ((9  :: 16 :: []) :: [])) ::
                (((25 :: 36 :: []) ::
                 ((49 :: 64 :: []) :: [])) :: [])))))
```

And similarly for n-ary operations, when the arguments are "compatible" (we will define and refine the notion of compatibility later):

```
(_+_ <$> v123 ⊛ v456)
  ≡ C (S (5 :: 7 :: 9 :: []))

(_+_ <$> v12-34 ⊛ v56-78)
  ≡ C (C (S (( 6 ::  8 :: []) ::
             ((10 :: 12 :: []) :: []))))
```

We should be able to process a matrix "per row", perhaps in a stateful manner:

```
sum v123 ≡ S 6
sum v123-456 ≡ C (S (6 :: 15 :: []))

sums v123 ≡ C (S (1 :: 3 :: 6 :: []))

sums v123-456 ≡ C (C (S ((1 :: 3 ::  6 :: []) ::
                         (4 :: 9 :: 15 :: []) :: [])))
```

Or "per column", using the *reranking* operator `¹, which amounts to pre- and post-composing the desired operation with a transposition:

```
sums `¹ v123-456 ≡ C (C (S ((1 :: 2 :: 3 :: []) ::
                            (5 :: 7 :: 9 :: []) :: [])))
```

1.1 Functor

To represent vectors, we need a notion of arrays of a given size:

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)

vec : ℕ → Set → Set
vec n A = Vec A n
```

Applying an operation pointwise to every elements of a vector is exactly what `map` does:

```
map-Vec : ∀ {n}{A B : Set} → (A → B) → vec n A → vec n B
map-Vec f [] = []
map-Vec f (x :: xs) = f x :: map-Vec f xs
```

This would allow us to lift the operation `square` on numbers to apply on vectors of numbers.

A function of type `Set → Set` having a `map` is called a functor:

```
record Functor (F : Set → Set) : Set₁ where
  infixl 4 _<$>_ _<$_

  field
    _<$>_ : ∀ {A B} → (A → B) → F A → F B

  _<$_ : ∀ {A B} → A → F B → F A
  x <$ y = const x <$> y

open Functor ⦃...⦄
instance
  VecFunctor : ∀ {n} → Functor (vec n)
  _<$>_ {{ VecFunctor {n} }} = map-Vec
```

It ought to abide by the functorial laws:

```
record IsFunctor (F : Set → Set){{_ : Functor F}} : Set₁ where
  field
    id-<$> : ∀ {A} (x : F A) →
                (id <$> x) ≡ x
    ∘-<$> : ∀ {A B C} (x : F A)(f : A → B)(g : B → C) →
                ((g ∘ f) <$> x) ≡ (g <$> (f <$> x))
```

Exercise (difficulty: 1) Prove the functor law for `vec`.

Another (arbitrary) example of functor is the following:

```
data Pair (A : Set) : Set where
  P : A → A → Pair A

instance
  PairFunctor : Functor Pair
  _<$>_ {{PairFunctor}} f (P x y) = P (f x) (f y)
```

Exercise (difficulty: 1) Prove the functor law for `Pair`.

Exercise (difficulty: 1) Show that lists define a functor.

1.2 Applicative

To lift n-ary operation `f` over two vectors of same size, we merely need a (total!) `zipWith`:

```
zipWith-Vec : ∀ {n} {A B C : Set} →
          (A → B → C) → vec n A → vec n B → vec n C
zipWith-Vec f [] [] = []
zipWith-Vec f (x :: xs) (y :: ys) = f x y :: zipWith-Vec f xs ys
```

However, `zipWith` can be obtained from two more primitive operations and the functoriality of vectors:

```
replicate-Vec : ∀ {n} {A : Set} → A → vec n A
replicate-Vec {n = zero}  x = []
replicate-Vec {n = suc n} x = x :: replicate-Vec x

_<*>-Vec_ : ∀ {n} {A B : Set} → vec n (A → B) → vec n A → vec n B
```

(continues on next page)

(continued from previous page)

```
[]       <*>-Vec []       = []
(f :: fs) <*>-Vec (x :: xs) = f x :: (fs <*>-Vec xs)

zipWith-Vec' : ∀ {n} {A B C : Set} →
          (A → B → C) → vec n A → vec n B → vec n C
zipWith-Vec' f xs ys = f <$> xs <*>-Vec ys
```

A functor equipped with these two operations is an applicative functor:

```
record Applicative (F : Set → Set) : Set₁ where
  infixl 4 _⊛_ _<⊛_ _⊛>_
  infix  4 _⊛_

  field
    pure : ∀ {A} → A → F A
    _⊛_  : ∀ {A B} → F (A → B) → F A → F B
    overlap {{super}} : Functor F

  zipWith : ∀ {A B C} → (A → B → C) → F A → F B → F C
  zipWith f x y = f <$> x ⊛ y

  _<⊛_ : ∀ {A B} → F A → F B → F A
  a <⊛ b = const <$> a ⊛ b

  _⊛>_ : ∀ {A B} → F A → F B → F B
  a ⊛> b = (const id) <$> a ⊛ b

  _⊗_ : ∀ {A B} → F A → F B → F (A × B)
  x ⊗ y = (_,_) <$> x ⊛ y

  replicate : ∀ {A} → A → F A
  replicate = pure

open Applicative ⦃...⦄
instance
  VecApplicative : ∀ {n} → Applicative (vec n)
  pure {{VecApplicative}} = replicate-Vec
  _⊛_ {{VecApplicative}} = _<*>-Vec_
```

It ought to abide by the applicative laws:

```
record IsApplicative (F : Set → Set){{_ : Applicative F}} : Set₁ where
  field
    pure-id : ∀ {A} (v : F A) →
                 (pure id ⊛ v) ≡ v
    pure-pure : ∀ {A B} (f : A → B)(a : A) →
                 (pure f ⊛ pure a) ≡ pure {F} (f a)
    applicative : ∀ {A B}{u : F (A → B)}{a : A} →
                 (u ⊛ pure a) ≡ (pure (λ f → f a) ⊛ u)
    composition : ∀ {A B C}{u : F (B → C)}{v : F (A → B)}{w : F A} →
                 (u ⊛ (v ⊛ w)) ≡ ((pure (λ g f a → g (f a))) ⊛ u ⊛ v ⊛ w)
```

Exercise (difficulty: 1) Prove the applicative laws for `vec`.

Pairs are also applicative:

```
instance
  PairApplicative : Applicative Pair
  pure {{PairApplicative}} a = P a a
  _⊛_  {{PairApplicative}} (P f g) (P x y) = P (f x) (g y)
```

Exercise (difficulty: 1) Prove the applicative laws for `Pair`.

Remark: Every monad is an applicative functor (but not conversely!). So, for example, the `State` monad (encountered in Lecture 1) is an applicative:

```
instance
  StateFunctor : ∀ {A : Set} → Functor (State A)
  _<$>_ {{StateFunctor}} f m s = let (x , s') = m s in
                                 f x , s'
  StateApplicative : ∀ {A : Set} → Applicative (State A)
  pure {{StateApplicative}} x s = x , s
  _⊛_  {{StateApplicative}} fs xs s = let (f , s') = fs s in
                                      let (x , s'') = xs s' in
                                      f x , s''
```

Exercise (difficulty: 1) Write a program that takes a monad (specified with `return` and `>>=`) and produces its underlying applicative.

1.3 Naperian

Let us (temporarily) model an m-by-n matrix as an m-elements vector of n-elements vectors:

```
matrix : ℕ → ℕ → Set → Set
matrix m n A = vec m (vec n A)
```

To implement transposition (and, therefore, reranking), we need to be able to *index* into a vector (say, "get the value on row `i` and column `j`") as well as to be able to *create* a vector as a function from its indices (say, "define the matrix of value `f(i, j)` at row `i` and column `j`). The first corresponds to a lookup while the second corresponds to a tabulation:

```
lookup-Vec : ∀ {n} {A : Set} → vec n A → Fin n → A
lookup-Vec (x ∷ xs)  zero = x
lookup-Vec (x ∷ xs) (suc i) = lookup-Vec xs i

tabulate-Vec : ∀ {n} {A : Set} → (Fin n → A) → vec n A
tabulate-Vec {zero}  f = []
tabulate-Vec {suc n} f = f zero ∷ tabulate-Vec (f ∘ suc)

transpose-Matrix : ∀ {m n} {A : Set} → matrix m n A → matrix n m A
transpose-Matrix m = tabulate-Vec (λ i →
                     tabulate-Vec (λ j →
                     lookup-Vec (lookup-Vec m j) i))
```

An applicative functor such that there exists a set `Log` supporting `lookup` and `tabulate` is called a Naperian functor or a representable functor:

```
record Naperian (F : Set → Set) : Set₁ where
  field
    Log : Set
    lookup : ∀ {A} → F A → (Log → A)
    tabulate : ∀ {A} → (Log → A) → F A
```

(continues on next page)

(continued from previous page)

```
    overlap {{super}} : Functor F

  positions : F Log
  positions = tabulate λ ix → ix

open Naperian {{...}}

instance
  VecNaperian : ∀ {n} → Naperian (vec n)
  Log {{VecNaperian {n}}} = Fin n
  lookup {{VecNaperian}} = lookup-Vec
  tabulate {{VecNaperian}} = tabulate-Vec
```

Exercise (difficulty: 2) State the Naperian laws and prove them for vectors.

Pairs are Naperian too:

```
instance
  PairNaperian : Naperian Pair
  Log {{PairNaperian}} = Bool
  lookup {{PairNaperian}} (P x y) true = x
  lookup {{PairNaperian}} (P x y) false = y
  tabulate {{PairNaperian}} f = P (f true) (f false)
```

Given any pair of Naperian functors, transposition is expressed as swapping the composition of structures:

```
transpose : ∀ {F G : Set → Set}
              {{_ : Naperian F}}{{_ : Naperian G}} →
            ∀ {A} → F (G A) → G (F A)
transpose fga = tabulate <$> (tabulate (λ gx fx → lookup (lookup fga fx) gx))
```

1.4 Interlude: Monoid

So far, we have focused our attention onto type constructors (functions of type `Set → Set`). But sets can be interesting too. For example, we may be interested in exhibiting the monoidal structure of a given set:

```
record Monoid (A : Set) : Set₁ where
  infixr 6 _<>_
  field
    mempty : A
    _<>_ : A → A → A

open Monoid ⦃...⦄
```

Famous monoids include `(ℕ, 0, _+_)` and `(List A, [], _++_)` (also called the free monoid):

```
instance
  NatMonoid : Monoid ℕ
  mempty {{NatMonoid}} = 0
  _<>_ {{NatMonoid}} = _+_

  ListMonoid : ∀ {A} → Monoid (List A)
  mempty {{ListMonoid}} = []
  _<>_ {{ListMonoid}} xs ys = xs ++ ys
```

Perhaps less obviously (or, perhaps, too obviously to be noted), endomorphisms form a monoid `(A → A, id, _∘_)`:

```
EndoMonoid : ∀ {A} → Monoid (A → A)
mempty {{EndoMonoid}} = id
_<>_ {{EndoMonoid}} f g = f ∘ g
```

Exercise (difficulty: 2) State the monoid laws and prove them for the above examples.

1.5 Foldable

To compute the running `sum` over a vector of numbers, we need a notion of iteration over vectors. In all generality, the left-to-right iteration over a vector can be implemented as the interpretation into a given monoid:

```
foldMap-Vec : ∀ {n}{A}{W : Set} {{MonW : Monoid W}} → (A → W) → vec n A → W
foldMap-Vec f [] = mempty
foldMap-Vec f (x ∷ xs) = f x <> foldMap-Vec f xs

sumAll-Vec : ∀ {n} → vec n ℕ → ℕ
sumAll-Vec = foldMap-Vec id
```

Note that we recover the 70's embodiment of iteration, the `foldr`, by exploiting the fact that endomorphisms form a monoid:

```
foldr-Vec : ∀ {n}{A B : Set} → (A → B → B) → B → vec n A → B
foldr-Vec su ze fs = foldMap-Vec su fs ze
```

Conversely, we can interpret it into the initial model of foldability, namely lists:

```
toList-Vec : ∀ {n A} → vec n A → List A
toList-Vec = foldMap-Vec (λ a → a ∷ [])
```

A functor offering such an iterator is said to be foldable:

```
record Foldable (F : Set → Set) : Set₁ where
  field
    foldMap : ∀ {A}{W : Set} {{MonW : Monoid W}} → (A → W) → F A → W
    overlap {{super}} : Functor F

  foldr : ∀ {A B : Set} → (A → B → B) → B → F A → B
  foldr su ze fs = foldMap su fs ze

  toList : ∀ {A} → F A → List A
  toList = foldMap (λ a → a ∷ [])

open Foldable {{...}}

sumAll : ∀ {F} → {{ _ : Foldable F}} → F ℕ → ℕ
sumAll = foldMap id

instance
  VecFoldable : ∀ {n} → Foldable (λ A → Vec A n)
  foldMap {{VecFoldable}} = foldMap-Vec
```

Pairs are foldable too:

```
instance
  PairFoldable : Foldable Pair
  foldMap {{PairFoldable}} f (P a b) = f a <> f b
```

Exercise (difficulty: 1) Show that lists are foldable.

Exercise (difficulty: 2) State the foldable laws and prove them for the above examples.

1.6 Traversable

Being foldable enables us to write pure iterators. To compute the running sum of a vector, we need to perform a stateful iteration:

```
traverse-Vec : ∀ {n F A B} {{_ : Applicative F}} → (A → F B) → vec n A → F (vec n B)
traverse-Vec f [] = pure []
traverse-Vec f (x :: v) = _::_ <$> f x ⊛ traverse-Vec f v

increase : ℕ → State ℕ ℕ
increase n = λ m → let n' = m + n in n' , n'

sumsAll-Vec : ∀ {n} → vec n ℕ → vec n ℕ
sumsAll-Vec xs = proj₁ (traverse-Vec increase xs 0)
```

Remark: Rather than an applicative, we could have asked for a monad. However, this is needlessly restrictive (remember, every monad is an applicative): if the side-effects are commutative (and we like those for performance reasons), we get more freedom with a purely applicative implementation rather than a monadic one (for the same reason that OCaml is applicative, compiler writers like under-specifications!).

A functor offering such an iterator is said to be traversable:

```
record Traversable (T : Set → Set) : Set₁ where
  field
    traverse : ∀ {F : Set → Set} {A B} {{_ : Applicative F}} → (A → F B) → T A → F (T B)
    overlap {{super}} : Foldable T

  sequence :  ∀ {F : Set → Set} {A} {{_ : Applicative F}} → T (F A) -> F (T A)
  sequence = traverse id

open Traversable ⦃...⦄
instance
  VectorTraversable : ∀ {n} → Traversable (λ A → Vec A n)
  traverse {{VectorTraversable}} f [] = pure []
  traverse {{VectorTraversable}} f (x :: v) = _::_ <$> f x ⊛ traverse f v
```

Surprise, pairs are traversable too:

```
instance
  PairTraversable : Traversable Pair
  traverse {{PairTraversable}} f (P x y) = P <$> f x ⊛ f y
```

Exercise (difficulty: 2) State the foldable laws and prove them for the above examples.

The running sum example can then be implemented for any traversable structure:

```
sumsAll : ∀ {T} {{_ : Traversable T}} → T ℕ → T ℕ
sumsAll xs = proj₁ (traverse increase xs 0)
```

1.7 Dimension

To serve as a data container, we thus require for our type constructor to be both traversable (ie. support effectful iteration) and naperian (ie. suppport indexing):

```
record Dimension (F : Set → Set) : Set₁ where
  field
    overlap {{super₁}} : Applicative F
    overlap {{super₂}} : Naperian F
    overlap {{super₃}} : Traversable F


  size : ∀ {α} → F α → ℕ
  size as = length (toList as)

open Dimension ⦃...⦄
```

As a result of our hard work, pairs and vectors are straightforward instances:

```
instance
  PairDimension : Dimension Pair
  PairDimension = record {}

  VectorDimension : ∀ {n} → Dimension (vec n)
  VectorDimension = record {}
```

Remark: Any dimensionable functor admits a `size` operation, which counts the number of elements stored in the structure. For vectors, a direct implementation of `size` would simply return the index of the vector (without conversion to list) and for pairs, it is constantly equal to 2.

Example: binary vectors rather than indexing vectors by Peano numbers, we can index them by binary numbers:

```
data Binary : Set where
  zero : Binary
  2× : Binary → Binary
  1+2× : Binary → Binary

data BVector (A : Set) : Binary → Set where
  single : A → BVector A zero
  join : ∀ {n} → BVector A n → BVector A n → BVector A (2× n)
  1+join : ∀ {n} → A → BVector A n → BVector A n → BVector A (1+2× n)

bvector : Binary → Set → Set
bvector b A = BVector A b
```

Exercise (difficulty: 2) Show that binary vectors can be used as a dimension:

```
instance
  BVectorFunctor : ∀ {n} → Functor (bvector n)
  BVectorFunctor = {!!}

  BVectorFoldable : ∀ {n} → Foldable (bvector n)
  BVectorFoldable = {!!}

  BVectorApplicative : ∀ {n} → Applicative (bvector n)
  BVectorApplicative = {!!}
```

(continues on next page)

(continued from previous page)

```
  BVectorNaperian : ∀ {n} → Naperian (bvector n)
  BVectorNaperian = {!!}

  BVectorTraversable : ∀ {n} → Traversable (bvector n)
  BVectorTraversable = {!!}

  BVectorDimension : ∀ {n} → Dimension (bvector n)
  BVectorDimension = record {}
```

Remark: as for vectors, the `size` of a binary vector can be statically deduced from the index:

```
bin2nat : Binary → ℕ
bin2nat zero = 0
bin2nat (2× b) = 2 * (bin2nat b)
bin2nat (1+2× b) = 1 + 2 * bin2nat b
```

Example: block matrices This example is taken from An agda formalisation of the transitive closure of block matrices, in which block matrices are defined as follows:

```
data Shape : Set where
  L  : Shape
  B  : Shape → Shape → Shape

data M (a : Set) : (rows cols : Shape) → Set where
    One  :  a → M a L L
    Row  :  {c₁ c₂ : Shape} →
            M a L c₁ → M a L c₂ →  M a L (B c₁ c₂)
    Col  :  {r₁ r₂ : Shape} →
            M a r₁ L → M a r₂ L →  M a (B r₁ r₂) L
    Q    :  {r₁ r₂ c₁ c₂ : Shape} →
            M a r₁ c₁ →  M a r₁ c₂ →
            M a r₂ c₁ →  M a r₂ c₂ →
            M a (B r₁ r₂) (B c₁ c₂)
```

Exercise (difficulty: 2) Show that block matrices can be used as a dimension:

```
instance
    MFunctor : ∀ {r c} → Functor (λ A → M A r c)
    MFunctor = {!!}

    MFoldable : ∀ {r c} → Foldable (λ A → M A r c)
    MFoldable = {!!}

    MApplicative : ∀ {r c} → Applicative (λ A → M A r c)
    MApplicative = {!!}

    MNaperian : ∀ {r c} → Naperian (λ A → M A r c)
    MNaperian = {!!}

    MTraversable : ∀ {r c} → Traversable (λ A → M A r c)
    MTraversable = {!!}

    MDimension : ∀ {r c} → Dimension (λ A → M A r c)
    MDimension = record {}
```

Exercise (difficulty: 2) Show that the generic `size` operator defined by `MDimension` is equivalent to the

following function:

```
toNat : Shape  →  ℕ
toNat L        =  1
toNat (B l r)  = toNat l + toNat r
```

Programming solely with the structure offered by dimensions, we can implement a generic inner product and matrix product:

```
inner-product : ∀ {F} → {{_ : Dimension F}} →
                F ℕ → F ℕ → ℕ
inner-product xs ys = sumAll (zipWith _*_ xs ys)

matrix-product : ∀ {F G H} →
                 {{_ : Dimension F}}{{_ : Dimension G}}{{_ : Dimension H}} →
                 F (G ℕ) → G (H ℕ) → F (H ℕ)
matrix-product {F}{G}{H} {{dimF}} xss yss =
    zipWith (zipWith inner-product) (replicate <$> xss) (replicate (transpose yss))
```

1.8 Multi-dimensional matrices

So far, we have mostly equipped vectors with structure (and pretended that we cared about `Pair`). To talk about m-by-n matrices, we ended up defining a custom datatype built from vectors of vectors. In this section, we are going to generalize matrices both in terms of dimension (the number of functors composed) and Dimension (the type of functors that are composed).

This is also were all the unification hell breaks loose. This means that we are going to introduce apparently useless definitions to guide the unifier and some manual instanciations here and there.

A high-dimensional matrix is essentially a composition of multiple Dimension functors. To help the unifier, we are going to reify the composition (and identity) through custom datatype definitions:

```
data Id (A : Set) : Set where
  I : A → Id A

data Seq (G : Set → Set)(F : Set → Set)(A : Set) : Set where
  S : F (G A) → Seq G F A
```

Unsurprisingly, the structures we have seen so far are verified by the identity functor and closed under composition, so we get the expected instances.

An hyper-matrix is essentially a list of functors:

```
hyper : Set₁
hyper = List (Set → Set)
```

which is interpreted as-is in the monoid of endofunctor on `Set`:

```
Hyper : hyper → Set → Set
Hyper [] A = Id A
Hyper (F :: Fs) A = Seq F (Hyper Fs) A
```

that is (but this would not play nice with unification):

```
Hyper : hyper → Set → Set
Hyper Fs A = foldMap {{_}}{{FunctorMonoid}} id Fs A
  where FunctorMonoid : Monoid (Set → Set)
```

(continues on next page)

(continued from previous page)

```
      mempty {{FunctorMonoid}} = Id
      _<>_ {{FunctorMonoid}} = Seq
```

`Hyper` thus provides a uniform way to type high-dimension matrices:

```
v123 : Hyper (vec 3 :: []) ℕ
v123 = S (I (1 :: 2 :: 3 :: []))

v456 : Hyper (vec 3 :: []) ℕ
v456 = S (I (4 :: 5 :: 6 :: []))

v123-456-789 : Hyper (vec 3 :: vec 3 :: []) ℕ
v123-456-789 = S (S (I ((1 :: 2 :: 3 :: []) ::
                        (4 :: 5 :: 6 :: []) ::
                        (7 :: 8 :: 9 :: []) :: [])))

v12-45-78 : Hyper (vec 2 :: vec 3 :: []) ℕ
v12-45-78 = S (S (I ((1 :: 2 :: []) ::
                     (4 :: 5 :: []) ::
                     (7 :: 8 :: []) :: [])))

m1234 : Hyper (vec 2 :: vec 2 :: []) ℕ
m1234 = S (S (I (((1 :: 2 :: []) ::
                 ((3 :: 4 :: []) :: []))))))

m5678 : Hyper (vec 2 :: vec 2 :: []) ℕ
m5678 = S (S (I (((5 :: 6 :: []) ::
                 ((7 :: 8 :: []) :: []))))))

v1234-5678 : Hyper (vec 2 :: vec 2 :: vec 2 :: []) ℕ
v1234-5678 = S (S (S (I (((1 :: 2 :: []) ::
                          ((3 :: 4 :: []) :: [])) ::
                         (((5 :: 6 :: []) ::
                          ((7 :: 8 :: []) :: [])) :: []))))))

v123-456 : Hyper (vec 3 :: vec 2 :: []) ℕ
v123-456 = S (S (I ((1 :: 2 :: 3 :: []) ::
                    ((4 :: 5 :: 6 :: []) :: []))))
```

While we can try to *inhabit* an hyper-matrix for **any** list of functors, we will only be able to *compute* with those when each of these functors are Dimensions:

```
Shapely : List (Set → Set) → Set₁
Shapely [] = ⊤
Shapely (F :: Fs) = Dimension F × Shapely Fs
```

As a result, a shapely list of functors is itself a dimension.

Exercise (difficulty: 3) Show that a shapely hyper-matrix has a dimension:

```
HyperFunctor : ∀ {Fs} → Shapely Fs → Functor (Hyper Fs)
HyperFunctor shapes = {!!}

HyperApplicative : ∀ {Fs} → Shapely Fs → Applicative (Hyper Fs)
HyperApplicative shapes = {!!}

HyperNaperian : ∀ {Fs} → Shapely Fs → Naperian (Hyper Fs)
```

(continues on next page)

(continued from previous page)

```
HyperNaperian shapes = {!!}

HyperFoldable : ∀ {Fs} → Shapely Fs → Foldable (Hyper Fs)
HyperFoldable shapes = {!!}

HyperTraversable : ∀ {Fs} → Shapely Fs → Traversable (Hyper Fs)
HyperTraversable shapes = {!!}

HyperDimension : ∀ {Fs} → Shapely Fs → Dimension (Hyper Fs)
HyperDimension shapes = {!!}
```

As a result, we can define:

```
square : ∀ {T} → {{_ : Traversable T}} → T ℕ → T ℕ
square x = (λ x → x * x) <$> x
```

and seamlessly apply it to any hyper-matrix.

We can also define the generalized running sum:

```
sums : ∀ {F Fs}
         {{_ : Shapely Fs}}{{_ : Dimension F}} →
         Hyper (F :: Fs) ℕ → Hyper (F :: Fs) ℕ
sums {{shapeFs}} (S xs) = S (sumsAll <$>H xs)
    where open Functor (HyperFunctor shapeFs) renaming (_<$>_ to _<$>H_)
```

and apply it to any matrix of dimension at least `F`.

We can also iterate over all "rows" of an hyper-matrix, bringing the dimension down by `F`:

```
reduceBy : ∀ {F Fs A M} →
             {{_ : Shapely Fs}}{{_ : Monoid M}}{{_ : Dimension F}} →
             (A → M) → Hyper (F :: Fs) A → Hyper Fs M
reduceBy {{shapeFs}} f (S fga) = (foldMap f) <$>H fga
    where open Functor (HyperFunctor shapeFs) renaming (_<$>_ to _<$>H_)

sum : ∀ {F Fs} →
        {{_ : Shapely Fs}}{{_ : Dimension F}} →
        Hyper (F :: Fs) ℕ → Hyper Fs ℕ
sum = reduceBy id
```

And, finally, we can generalize `transpose` to any hyper-matrix and obtain the reranking operator:

```
transpose' : ∀ {A F G Fs} →
             {{_ : Shapely Fs}}{{_ : Dimension F}}{{_ : Dimension G}} →
             Hyper (F :: G :: Fs) A → Hyper (G :: F :: Fs) A
transpose' {{shapeFs}} (S (S x)) = S (S (transpose <$>H x))
    where open Functor (HyperFunctor shapeFs) renaming (_<$>_ to _<$>H_)

_`¹_ : ∀ {A F₁ F₂ Fs G₁ G₂ Gs} →
         {{_ : Shapely Fs}}{{_ : Shapely Gs}} →
         {{_ : Dimension F₁}}{{_ : Dimension F₂}}
         {{_ : Dimension G₁}}{{_ : Dimension G₂}} →
         (Hyper (F₁ :: F₂ :: Fs) A → Hyper (G₁ :: G₂ :: Gs) A) →
         Hyper (F₂ :: F₁ :: Fs) A → Hyper (G₂ :: G₁ :: Gs) A
f `¹ m = transpose' (f (transpose' m))
```

At this stage, we are merely touching upon what Gibbons' talks about in APLicative Programming Naperian Functors. For instance, when applying a binary operation, we (that is, applicative) currently ask for the two argument matrices to be exactly the same. J, on the other hand, would automatically lift values to match up dimensions. For example, we would like to able to sum a scalar to a matrix:

```
I 3 + S (I (4 :: 5 :: 6 :: []))
≡ S (I (3 :: 3 :: 3 :: [])) + S (I (4 :: 5 :: 6 :: []))
≡ S (I (7 :: 8 :: 9 :: []))

S (I (1 :: 2 :: 3 :: [])) + S (S (I ((4 :: 5 :: 6 :: []) ::
                                     (7 :: 8 :: 9 :: []) :: [])))
≡ S (S (I (1 :: 2 :: 3 :: []) ::
          (1 :: 2 :: 3 :: []) :: []))
+ S (S (I ((4 :: 5 :: 6 :: []) ::
           (7 :: 8 :: 9 :: []) :: [])))
≡ S (S (I ((5 :: 7 :: 9 :: []) ::
           (8 :: 10 :: 12 :: []) :: [])))
```

However, this is also at this point that the extensional style starts to break. To feel that pain, try to translate Gibbons' `Max` type-class. As we will see in the last lecture, manipulating an object of type `List (Set → Set)` is a red-herring, it is already quite surprising that we came this far.

CHAPTER

TWO

INTENSIONAL GENERIC PROGRAMMING

In this second part, we apply a type-theoretic concept, a *universe*, to manipulate some structure of interest. Here, we shall look at inductive types.

Universes were born around the same time as type theory: they were introduced by Martin-Löf in Intuitionistic Type Theory. Their application to generic programming came later with Universes for Generic Programs and Proofs in Dependent Type Theory.

Following The Gentle Art of Levitation, we shall:

- code a universe for describing inductive types
- show that the resulting types admit an induction principle
- implement a generic datatype construction: the free monad
- reflect the universe in itself

Vision: "Whereof one cannot speak, thereof one must be silent."

2.1 Descriptions

In lecture 1, we asked whether we could give a "grammar" for the functors used to encode the signatures of algebraic effects. As mentioned then, signatures are essentially the same as datatype definitions. We shall thus decompose our model of inductive types in, first, the underlying functor encoding a signature and, second, a fixpoint of signatures.

The grammar can be understood as taking the closure of all the operations offering/preserving a functorial structure. Namely, the identity and constant functors are functors. Then, the pointwise product of functors is itself a functor while the indexed sum and product of functors is itself a functor. The *code* of the universe translates this intuition by describing the *smallest* set closed under those operations:

```
data Desc : Set₁ where
  `X   : Desc
  `K   : Set → Desc
  _`×_ : (D₁ D₂ : Desc) → Desc
  _`+_ : (D₁ D₂ : Desc) → Desc
  `Σ   : (S : Set)(D : S → Desc) → Desc
  `Π   : (S : Set)(D : S → Desc) → Desc
```

The *interpretation* gives the desired semantics:

```
⟦_⟧ : Desc → Set → Set
⟦ `X ⟧ X       = X
⟦ `K S ⟧ X     = S
⟦ D₁ `× D₂ ⟧ X = ⟦ D₁ ⟧ X × ⟦ D₂ ⟧ X
```

(continues on next page)

(continued from previous page)

```
⟦ D₁ `+ D₂ ⟧ X = ⟦ D₁ ⟧ X ⊎ ⟦ D₂ ⟧ X
⟦ `Σ S T ⟧ X   = Σ[ s ∈ S ] ⟦ T s ⟧ X
⟦ `Π S T ⟧ X   = (s : S) → ⟦ T s ⟧ X
```

Exercise (difficulty: 2) Note that we would expect the composition of two functors to be a functor. Implement composition of descriptions:

```
_∘D_ : Desc → Desc → Desc
D₁ ∘D D₂ = {!!}

correctness-∘ : ∀ {X D₁ D₂} → ⟦ D₁ ∘D D₂ ⟧ X ≡ ⟦ D₁ ⟧ (⟦ D₂ ⟧ X)
correctness-∘ = {!!}
  where postulate ext : Extensionality Level.zero Level.zero
```

Exercise (difficulty: 2) We claim that our description interpret to functors. We ought to be able to equip *any* description with a functorial action:

```
map : ∀ {X Y} → (D : Desc)(f : X → Y)(v : ⟦ D ⟧ X) → ⟦ D ⟧ Y
map = {!!}
```

and (generically) prove the functor laws:

```
proof-map-id : ∀ {X} → (D : Desc)(v : ⟦ D ⟧ X) → map D id v ≡ v
proof-map-id = {!!}
  where postulate ext : Extensionality Level.zero Level.zero

proof-map-compos : ∀ {X Y Z}{f : X → Y}{g : Y → Z} →
                   (D : Desc)(v : ⟦ D ⟧ X) →
                   map D (λ x → g (f x)) v ≡ map D g (map D f v)
proof-map-compos = {!!}
  where postulate ext : Extensionality Level.zero Level.zero
```

2.2 Fixpoint

The functors captured by our grammar have also the property of being "strictly-positive". We are therefore allowed to take their fixpoint:

```
data μ (D : Desc) : Set where
  ⟨_⟩ : ⟦ D ⟧ (μ D) → μ D
```

Over this (standard) inductive type, we can implement the traditional `fold` operator:

```
{-# TERMINATING #-}
fold : (D : Desc){T : Set} →
       (⟦ D ⟧ T → T) → μ D → T
fold D α ⟨ x ⟩ = α (map D (fold D α) x)
```

Exercise (difficulty: 3) Convince the termination checker that `fold` is indeed terminating. Hint: manually specialize the partially applied function `map D (fold D α)` in a definition mutually-recursive with `fold`.

Example: natural numbers:: Natural numbers are thus described as follows:

```
data NatTag : Set where
  `Ze `Su : NatTag
```

(continues on next page)

(continued from previous page)

```
NatD : Desc
NatD = `Σ NatTag (λ { `Ze → `K ⊤
                    ; `Su → `X })

Nat : Set
Nat = μ NatD

pattern ze = ⟨ `Ze , tt ⟩
pattern su n = ⟨ `Su , n ⟩
```

Using the `fold`, we can implement addition over these numbers:

```
plus : Nat → Nat → Nat
plus x = fold NatD (λ { (`Ze , tt) → x
                      ; (`Su , rec) → su rec })

test : plus (su (su ze)) (su (su (su ze)))
       ≡ su (su (su (su (su ze))))
test = refl
```

Example: lists:: Similarly, here are lists:

```
data ListTag : Set where
  `Nil `Cons : ListTag

ListD : Set → Desc
ListD X = `Σ ListTag (λ { `Nil → `K ⊤
                        ; CCons → `Σ X λ _ → `X })

List : Set → Set
List X = μ (ListD X)

nil : ∀ {X} → List X
nil = ⟨ `Nil , tt ⟩

cons : ∀ {X} → X → List X → List X
cons x t = ⟨ `Cons , x , t ⟩
```

Exercise (difficulty: 1):: Implement binary trees using descriptions.

2.3 Induction

Introducing a `fold` to enable recursion over μ `D` is simple(-type)-minded. Being in type theory, we actually want a recursion principle. We obtain it by instantiating the usual framework for induction:

```
All : ∀{X} → (D : Desc)(P : X → Set) → ⟦ D ⟧ X → Set
All `X         P x         = P x
All (`K Z)     P x         = ⊤
All (D₁ `× D₂) P (d₁ , d₂) = All D₁ P d₁ × All D₂ P d₂
All (D₁ `+ D₂) P (inj₁ d₁) = All D₁ P d₁
All (D₁ `+ D₂) P (inj₂ d₂) = All D₂ P d₂
All (`Σ S T)   P (s , xs)  = All (T s) P xs
All (`Π S T)   P k         = ∀ s → All (T s) P (k s)

Rec-μ : ∀ D → RecStruct (μ D) _ _
```

(continues on next page)

(continued from previous page)

```
Rec-μ D P ⟨ xs ⟩ = All D P xs

all : ∀ {X P} → (D : Desc) → (rec : (x : X) → P x)(x : ⟦ D ⟧ X) → All D P x
all `X rec x = rec x
all (`K S) rec z = tt
all (D₁ `× D₂) rec (d₁ , d₂) = all D₁ rec d₁ , all D₂ rec d₂
all (D₁ `+ D₂) rec (inj₁ d₁) = all D₁ rec d₁
all (D₁ `+ D₂) rec (inj₂ d₂) = all D₂ rec d₂
all (`Σ S T) rec (s , xs) = all (T s) rec xs
all (`Π S T) rec k = λ s → all (T s) rec (k s)

{-# TERMINATING #-}
rec-μ-builder : ∀{D} → RecursorBuilder (Rec-μ D)
rec-μ-builder {D} P rec ⟨ xs ⟩ = all D (λ x → rec x (rec-μ-builder P rec x)) xs

induction : (D : Desc)(P : μ D → Set) →
            ((x : ⟦ D ⟧ (μ D)) → All D P x → P ⟨ x ⟩) →
            (x : μ D) → P x
induction D P ms xs = build rec-μ-builder P (λ { ⟨ x ⟩ x₁ → ms x x₁ }) xs
```

Exercise (difficulty: 3):: Convince the termination checker that induction is terminating, either by implementing `rec-μ-builder` in an obviously terminating manner, or by writing `induction` directly in terms of `all`.

Using induction, we can write any dependently-typed programs or proofs over described inductive types: they have become (mostly, modulo the fact that we have to go trough the fold/induction principle, which is not idiomatic Agda) first-class objects.

But we can also take this as a opportunity to understand what we did earlier, in a simply-typed setting:

```
plus[_:_] : Nat → Nat → Set
plus[ m : n ] = Nat

plus : (m n : Nat) → plus[ m : n ]
plus m = induction NatD (λ n → plus[ m : n ])
         (λ { (`Ze , tt) tt → m
            ; (`Su , n) rec → su rec } )
```

2.4 Generic free monad

Thinking about it, we now have first-class inductive types (modulo encoding, again). This means that we can craft new datatypes from existing datatypes. We exercise this possibility by implementing a generic free monad construction.

In its most brutal form, the free monad construction consists in grafting an extra constructor containing a value of a provided type, the elements of the earlier signature being integrated as operations `op`:

```
_*D_ : Desc → Set → Desc
D *D X = `Σ Bool λ { true → `K X ; false → D }

Free : Desc → Set → Set
Free D X = μ (D *D X)

return : ∀ {D X} → X → Free D X
return x = ⟨ true , x ⟩
```

(continues on next page)

(continued from previous page)

```
op : ∀ {D X} → ⟦ D ⟧ (Free D X) → Free D X
op xs = ⟨ false , xs ⟩
```

Doing so, the resulting description has a monadic structure, which we can realize generically:

```
subst[_:_:_] : ∀ {X Y} → (D : Desc) → Free D X → (X → Free D Y) → Set
subst[_:_:_] {X}{Y} D _ _ = Free D Y

subst : ∀ {X Y} → (D : Desc) →
        Free D X → (X → Free D Y) → Free D Y
subst {X}{Y} D mx k =
  induction (D *D X) (λ mx₁ → subst[ D : mx : k ])
    (λ { (true , x) tt → k x
       ; (false , xs) as → ⟨ false , help D xs as ⟩ })
    mx
  where help : ∀ {X Y} D → (ds : ⟦ D ⟧ X) → All D (λ _ → Y) ds → ⟦ D ⟧ Y
        help `X ds as = as
        help (`K x) ds as = ds
        help (D₁ `× D₂) (ds₁ , ds₂) (as₁ , as₂) = help D₁ ds₁ as₁ , help D₂ ds₂ as₂
        help (D₁ `+ D₂) (inj₁ ds₁) as₁ = inj₁ (help D₁ ds₁ as₁)
        help (D₁ `+ D₂) (inj₂ ds₂) as₂ = inj₂ (help D₂ ds₂ as₂)
        help (`Σ S D₁) (s , ds) as = s , help (D₁ s) ds as
        help (`Π S D₁) ds as = λ s → help (D₁ s) (ds s) (as s)
```

Exercise (difficulty: 4):: Prove the monad laws.

2.5 Bootstrap

So far, we have been doing "generic programming" on the one hand (computing over Desc) and "programming" on the other hand (computing over anything else, including inhabitants of μ `D`, for `D : Desc`). This may have gone unnoticed (probably because doing anything with these encodings is blindly painful) but, in a standalone language, this would mean having two "programming languages" in the programming language, one for generic programming and the other for programming.

There may be two solutions to this problem: either we (pragmatically) make the generic programming language to borrow as much as possible from the programming language, or we (brutally) collapse the programming language into the generic programming language. For obvious reasons (having to do with full employment), we chose the latter. The key idea consists in noticing that `Desc` itself is an inductive type. As such, it can be described:

```
DescD : Desc
DescD =  `K ⊤
      `+ `K Set
      `+ (`X `× `X)
      `+ (`X `× `X)
      `+ (`Σ Set λ S → `Π S (λ _ → `X))
      `+ (`Σ Set λ S → `Π S (λ _ → `X))

Desc' : Set₁
Desc' = μ DescD

`X' : Desc'
`X' = ⟨ inj₁ tt ⟩
```

(continues on next page)

(continued from previous page)

```
`K' : Set → Desc'
`K' S = ⟨ inj₂ (inj₁ S) ⟩

_`×'_ : Desc' → Desc' → Desc'
D₁ `×' D₂ = ⟨ inj₂ (inj₂ (inj₁ (D₁ , D₂) )) ⟩

_`+'_ : Desc' → Desc' → Desc'
D₁ `+' D₂ = ⟨ inj₂ (inj₂ (inj₂ (inj₁ (D₁ , D₂) ))) ⟩

`Σ' : (S : Set)(T : S → Desc') → Desc'
`Σ' S T = ⟨ inj₂ (inj₂ (inj₂ (inj₂ (inj₁ (S , T))))) ⟩

`Π' : (S : Set)(T : S → Desc') → Desc'
`Π' S T = ⟨ inj₂ (inj₂ (inj₂ (inj₂ (inj₂ (S , T))))) ⟩
```

Note that, aside from the constructor `⟨_⟩` of μ, the constructor of `Desc'` only depend on constructors pre-existing in the type theory (unit, cartesian product, injections into sum): in an implementation, we can simply take these codes as the *definition*. We then only need to implement the fixpoint operator and its induction principle: this provides us with the ability to compute over inductive types on one hand (programming) but also, in particular, to compute over Desc since it is described in itself (generic programming).

CHAPTER

THREE

CONCLUSION

We have seen two complementary approaches to generic programming. In both cases, we have exploited (type-class) or built (universe) a mechanism that allows us to reify a subset of the programming language in itself.

Whichever mechanism we chose depends highly on the functionalities offered by the programming language. For instance, Coq type-classes are extremely powerful whereas its strict-positive criteria is extremely obtuse: as a result, the extensional approach works well whereas the intensional one is nearly impossible.

Exercises (difficulty: open ended):

- Implement Section 6 to 8 of Gibbons' paper (in Coq, probably)
- Extend `Desc` to encode inductive families
- Extend `Desc` to support internal fixpoints (such as `data Rose A = rose : List (Rose A)` $\rightarrow$ `Rose A`)

Going further, extensionally:

- Other examples of functor-oriented programming: unification-fd, lenses
- Structures in idiomatic Agda: Agda Prelude