Programación funcional avanzada

Primeras estructuras algebraicas

Federico Sawady

Universidad Nacional de Quilmes

Abril, 2017

Outline

1. Estructuras algebraicas
   - Abstracciones
   - Definición
   - Monoides
   - Duality and the De Morgan Principle
   - Functors
   - Foldable

Outline

Operaciones

```
sum    = foldr (+) 0
prod   = foldr (*) 1
concat = foldr (++) []
and    = foldr (&&) True
or     = foldr (||) False
```

Operaciones

```
(+)  :: Int  -> Int  -> Int
(*)  :: Int  -> Int  -> Int
(++) :: [a]  -> [a]  -> [a]
(&&) :: Bool -> Bool -> Bool
(||) :: Bool -> Bool -> Bool
```

Operaciones

```
sum    :: [Int]  -> Int  -> Int
prod   :: [Int]  -> Int  -> Int
concat :: [[a]]  -> [a]  -> [a]
and    :: [Bool] -> Bool -> Bool
```

Operaciones

Un compose más específico

```
(.) :: (a -> a)  -> (a -> a)  -> (a -> a)
```

Operaciones

```
composeL :: [(a -> a)] -> (a -> a) -> (a -> a)
composeL = foldr (.) id
```

Operaciones

Operaciones binarias, internas, asociativas, con elemento neutro (identidad de la operación). Se cumple:

```
x <> e = e <> x = x

x <> (y <> z) = (x <> y) <> z

x :: T, y :: T, entonces x <> y :: T
```

Operaciones

Si pudieramos abstraer todas las operaciones específicas podríamos escribir:

```
... = foldr <> e
```

Outline

1. Estructuras algebraicas
   - Abstracciones
   - Definición
   - Monoides
   - Duality and the De Morgan Principle
   - Functors
   - Foldable

Estructura algebarica

Estructura algebraica

En álgebra abstracta, una estructura algebraica, también conocida como sistema algebraico, es una n-tupla $(a1, a2, \ldots, an)$, donde $a1$ es un conjunto dado no vacío, y $\{a2, \ldots, an\}$ un conjunto de operaciones aplicables a los elementos de dicho conjunto.

Estructura algebarica

Estructuras algebraicas más utilizadas en matemática:

1. Magma
2. Semigrupo
3. Monoide
4. Grupo
5. Anillo
6. Retículo

Outline

1. Estructuras algebraicas
   - Abstracciones
   - Definición
   - Monoides
   - Duality and the De Morgan Principle
   - Functors
   - Foldable

Introducing Monoids

Monoide

A monoid is an algebraic structure with a single associative closed binary operation and an identity element

Monoids

The functional programming world is full of monoids:

`List: ++ and []`

`Sum: + and 0`

`Prod: * and 1`

`Bool: && and True`

`Bool: || and False`

And many others

Monoides

```haskell
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a

-- Laws:
-- mappend must be associative.
-- mempty is the identity of mappend
```

Monoides

```haskell
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a

-- sinonimo
(<>) = mappend
```

Sum Monoid

```haskell
instance Monoid Int where
    mempty = 0
    mappend x y = x + y
```

Sum Monoid

```haskell
instance Monoid Int where
    mempty = 0
    mappend = (+)
```

Sum Monoid

```haskell
instance Monoid Int where
    mempty = 1
    mappend = (*)
```

Sum Monoid

```haskell
newtype Sum = Sum Int
-- zero-cost abstraction

getSum (Sum x) = x

instance Monoid Sum where
    mempty = Sum 0
    mappend (Sum x) (Sum y) = Sum (x+y)
```

Prod Monoid

```haskell
newtype Prod = Prod Int

getProd (Prod x) = x

instance Monoid Prod where
    mempty = Prod 1
    mappend (Prod x) (Prod y) = Prod (x*y)
```

List Monoid

```haskell
instance Monoid [a] where
    mempty = []
    mappend = (++)
```

All Monoid

```haskell
newtype All = All Bool

getAll (All b) = b

instance Monoid All where
    mempty = All True
    mappend (All x) (All y) = All (x && y)
```

All Monoid

```haskell
newtype Any = Any Bool

getAny (Any b) = b

instance Monoid Any where
    mempty = Any False
    mappend (Any x) (Any y) = Any (x || y)
```

Endo Monoid

```haskell
newtype Endo a = Endo (a -> a)

appEndo (Endo f) = f

instance Monoid Endo where
    mempty = Endo id
    mappend (Endo f) (Endo g) = Endo (f . g)
```

Pair of monoids

```haskell
instance (Monoid a, Monoid b) => Monoid (a, b) where
    mempty = (mempty, mempty)
    mappend (a1, b1) (a2, b2) =
        (mappend a1 a2, mappend b1 b2)
```

Maybe Monoid

```haskell
instance Monoid a => Monoid (Maybe a) where
  mempty = Nothing
  mappend Nothing m = m
  mappend m Nothing = m
  mappend (Just m1) (Just m2) = Just (m1 `mappend` m2)
```

Maybe (first) Monoid

```haskell
newtype First a = First (Maybe a)

getFirst (First m) = m

instance Monoid (First a) where
    mempty = First Nothing
    mappend (First Nothing) r = r
    mappend l               _ = l
```

Maybe (last) Monoid

```haskell
newtype Last a = Last (Maybe a)

getLast (Last m) = m

instance Monoid (Last a) where
    mempty = Last Nothing
    mappend l (Last Nothing) = l
    mappend _ r              = r
```

Monoid Concat

```haskell
mconcat :: Monoid o => [o] -> o
```

Monoid Concat

```haskell
mconcat :: Monoid o => [o] -> o
mconcat []     = mempty
mconcat (m:ms) = m <> mconcat ms
```

Monoid Concat

```haskell
mconcat :: Monoid o => [o] -> o
mconcat = foldr (<>) mempty
```

Monoid Concat

```haskell
sum :: [Int] -> Int
sum  = getSum  . mconcat . map Sum

prod :: [Int] -> Int
prod = getProd . mconcat . map Prod

concat :: [[a]] -> [a]
concat = mconcat

all, any :: (a -> Bool) -> [a] -> Bool
all p = getAll . mconcat . map (All . p)
any p = getAny . mconcat . map (Any . p)
```

Semigrupos

Los monoides son semigrupos

```
class Semigroup a => Monoid a where
  mempty :: a
  mappend :: a -> a -> a
```

Semigrupos

```haskell
class Semigroup a where
  (<>) :: a -> a -> a

-- Laws:
-- <> is associative
```

Semigrupos

```haskell
sconcat :: Semigroup a => [a] -> a
sconcat [x]      = x
sconcat (x : xs) = x <> sconcat xs
```

Semigrupos

```haskell
sconcat :: Semigroup a => [a] -> a
sconcat = foldr1 (<>)
```

Semigrupos

```haskell
stimes :: Int -> a -> a
stimes 1 x = x
stimes n x = x <> stimes (n-1) x
```

Semigrupos

```haskell
stimes :: Int -> a -> a
stimes n x = sconcat $ replicate n x
```

Semigrupos

```
mult x y = stimes x (Sum y)
```

Monoids

```haskell
mtimesDefault :: Monoid a => Int -> a -> a
mtimesDefault 0 x = mempty
mtimesDefault n x = x <> mtimesDefault (n-1) x
```

Semigrupos

```haskell
newtype Min a = Min a

getMin (Min x) = x

instance Ord a => Semigroup (Min a) where
  (<>) = min

instance (Ord a, Bounded a) => Monoid (Min a) where
  mempty = maxBound
  mappend = (<>)
```

Semigrupos

```haskell
newtype Max a = Max a

getMax (Max x) = x

instance Ord a => Semigroup (Min a) where
  (<>) = Max

instance (Ord a, Bounded a) => Monoid (Min a) where
  mempty = minBound
  mappend = (<>)
```

Semigrupos

```haskell
instance Semigroup () where
  _ <> _ = ()

instance Semigroup b => Semigroup (a -> b) where
  f <> g = \a -> f a <> g a
```

Outline

Duality and the De Morgan Principle

```haskell
class Dual a where
   opuesto :: a -> a

-- laws
-- opuesto . opuesto = id
```

Duality

```
instance Dual Bool where
   opuesto = not
```

Duality

```haskell
instance Dual Int where
    opuesto = negate
    -- opuesto = (*) (-1)
```

Duality

```haskell
instance Dual a => Dual [a] where
    opuesto = reverse . map opuesto
```

Duality

```
instance (Dual a, Dual b) => Dual (a -> b) where
   opuesto f = opuesto . f . opuesto
```

Demostración de que las funciones implementan Dual correctamente:

```
opuesto (opuesto f)
  = -- { definición de opuesto, dos veces }
    opuesto . opuesto . f . opuesto . opuesto
  = -- { opuesto . opuesto = id for Dual a, Dual b }
    id . f . id
  = -- { ((.),id) es un monoide }
    f
```

Distributes over application and composition

```
opuesto (f x) = (opuesto f) (opuesto x)

opuesto (f . g) = opuesto f . opuesto g
```

Duality

```
opuesto max 3 5
= { LEMA }
opuesto (max (opuesto 3) (opuesto 5))
= { def opuesto }
opuesto (max (-3) (-5))
= { def max }
opuesto (-3)
= { def opuesto }
3
```

Duality

```
min = opuesto max

max = opuesto min
```

Duality

```
opuesto head [1..10]

last = opuesto head
```

Duality

```
opuesto tail [1..10]

init = opuesto tail

opuesto (++) [1,2] [3,4] -- opuesto (++) = flip (++)
```

Duality

```
opuesto (++) [1,2] [3,4]

opuesto (++) = flip (++)
```

Duality

```
True && x = x
False && x = False

opuesto (&&) True x
=
opuesto (&&) (opuesto True) (opuesto x)
=
opuesto (&&) False (not x)
=
opuesto (False && not x)
=
opuesto False
=
True
```

Duality

```
True && x = x
False && x = False

opuesto (&&) False x
=
opuesto (&&) (opuesto False) (opuesto x)
=
opuesto (&&) True (not x)
=
opuesto (True && not x)
=
opuesto (not x)
=
x
```

Duality

```
(||) = opuesto (&&)

(&&) = opuesto (||)
```

Duality

Si la operacion es conmutativa:

```
opuesto (+) 3 4

opuesto (+) = (+)
```

Duality

```
foldr' :: (Dual a, Dual b) => (a -> b -> b) -> b -> [a] -> b
foldr' f z [] = z
foldr' f z (x:xs) = f x (foldr f z xs)

foldl' :: (Dual a, Dual b) => (a -> b -> b) -> b -> [a] -> b
foldl' = opuesto foldr'
```

Duality

```haskell
foldl' :: (Dual a, Dual b) => (b -> a -> b) -> b -> [a] -> b
foldl' = opuesto foldr' . flip
```

Outline

Map function

Consider the familiar map function:

```
map :: (a -> b) -> List a -> List b
map f  []     = []
map f (x:xs) = f x : map f xs
```

MapM function

We can do the same with Maybe:

```
mapM :: (a -> b) -> Maybe a -> Maybe b
mapM f Nothing = Nothing
mapM f (Just x) = Just (f x)
```

MapT Function

We can do the same with Tree:

```
mapT :: (a -> b) -> Tree a -> Tree b
mapT f EmptyT = EmptyT
mapT f (Node x t1 t2) =
    Node (f x) (mapT f t1)
               (mapT f t2)
```

Looking for an abstraction

We can think two questions:

- What are those functions doing?
- What abstractions can we infer?

Looking for an abstraction

```
map  :: (a -> b) -> List  a -> List  b
mapM :: (a -> b) -> Maybe a -> Maybe b
mapT :: (a -> b) -> Tree  a -> Tree  b
```

Looking for an abstraction

```haskell
map  :: (a -> b) -> List  a -> List  b
mapM :: (a -> b) -> Maybe a -> Maybe b
mapT :: (a -> b) -> Tree  a -> Tree  b

-- Abstraigo el tipo del contenedor
fmap :: (a -> b) -> f a -> f b
```

Functor Class

```haskell
class Functor f where
   fmap :: (a -> b) -> f a -> f b
```

Functor

Definition

Functors are structures that *can be mapped over*

Functor

Int y Bool no son Functor porque no son contenedores de datos

Functor

`fmap` takes a function from one type to another and a **functor** applied with one type and returns a **functor** applied with another type

Laws

```haskell
fmap id = id
fmap (f . g) = fmap f . fmap g
```

Other Functors

```haskell
instance Functor (r ->) where
--   fmap :: (a -> b) -> (r -> a) -> (r -> b)
    fmap f g = f . g

    -- fmap f g = (.)
```

Other Functors

Siempre mapeo el último parámetro:

```
instance Functor (Either b) where
--  fmap :: (a -> b) -> Either b a -> Either b a
    fmap f (Left x) = Left x
    fmap f (Right y) = Right (f y)
```

Deriving

Siempre es mecánico:

```
data Tree a = EmptyT | NodeT a (Tree a) (Tree a)
                 deriving Functor
```

Outline

Foldable

```haskell
class Foldable t where
    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap f = foldr (mappend . f) mempty

    foldr :: (a -> b -> b) -> b -> t a -> b
    foldr f z t = appEndo (foldMap (Endo . f) t) z
```

Foldable

```haskell
class Foldable t where
    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap f = foldr (mappend . f) mempty

    foldr :: (a -> b -> b) -> b -> t a -> b
    foldr f z t = appEndo (foldMap (Endo . f) t) z

    fold :: Monoid m => t m -> m
    fold = foldMap id
```

Foldable

```haskell
class Foldable t where
    ... -- todas también definidas
    foldr' :: (a -> b -> b) -> b -> t a -> b
    foldl' :: (b -> a -> b) -> b -> t a -> b
    foldr1 :: (a -> a -> a) -> t a -> a
    foldl1 :: (a -> a -> a) -> t a -> a
    toList :: t a -> [a]
    null :: t a -> Bool
    length :: t a -> Int
    elem :: Eq a => a -> t a -> Bool
    maximum :: forall a . Ord a => t a -> a
    minimum :: forall a . Ord a => t a -> a
    product :: Num a => t a -> a
```

Foldable

```haskell
and :: Foldable t => t Bool -> Bool
or :: Foldable t => t Bool -> Bool
any :: Foldable t => (a -> Bool) -> t a -> Bool
all :: Foldable t => (a -> Bool) -> t a -> Bool
find :: Foldable t => (a -> Bool) -> t a -> Maybe a
maximumBy :: Foldable t => (a -> a -> Ordering) -> t a -> a
minimumBy :: Foldable t => (a -> a -> Ordering) -> t a -> a
```

Foldable

```haskell
sum :: (Foldable t, Num a) => t a -> a
sum = getSum . foldMap Sum
```

Monoid Concat

```haskell
mconcat :: (Foldable f, Monoid o) => f o -> o
mconcat = foldr (<>) mempty
```

Foldable

```haskell
data Tree a = Empty | Node a (Tree a) (Tree a)

-- Hay que implementar foldMap
foldMap :: (Monoid m, Foldable t) => (a -> m) -> t a -> m
```

Foldable

```haskell
instance Foldable Tree where
    foldMap f Empty = mempty
    foldMap f (Node x l r) =
       foldMap f l `mappend` f x `mappend` foldMap f r
```

Foldable

```
data Tree a = Empty | Node a (Tree a) (Tree a)
                deriving Foldable
```

Foldable

Foldable instances are expected to satisfy the following laws:

```
foldr f z t = appEndo (foldMap (Endo . f) t ) z

foldl f z t =
    appEndo (getDual
        (foldMap (Dual . Endo . flip f) t)) z

fold = foldMap id
```

Foldable

If the type is also a Functor instance, it should satisfy

```
foldMap f = fold . fmap f

foldMap f . fmap g = foldMap (f . g)
```

For Further Reading I

Miran LipovačaA.
Learn You a Haskell for Great Good!
April 2011.

Mark P. Jones.
Functional programming with overloading and higher-order polymorphism