Composition in FP

Vladimir Ciobanu

Monday, August 20, 2018

Overview

Algebraic Recap

Back to FP

Monad

Applicative

Algebraic Recap

What Is Composition?

Composition is essential in *functional programming*. We use mathematical abstractions (such as *Semigroup* and *Monoid*) to define the composition of a wide variety of data types.

Short recap:

- **Sets**, e.g., $\mathbb{B} = \{\top, \bot\}$, $\mathbb{N} = \{0, 1, 2, \ldots\}$, $\mathbb{T} = \{Void, (), Bool, Int, ((), Bool), [(Int, Bool)], \ldots\}$
- **Operations**, e.g., there are four unary operations on Bool (const $\top$, const $\bot$, identity and not)
- **Associativity**, e.g., $(a \vee b) \vee c = a \vee (b \vee c)$
- **Commutativity**, e.g., $a \vee b = b \vee a$

Semigroup

A **semigroup** is an algebraic structure consisting of a set and an *associative binary operation*.

Given a set $\mathbb{S}$ and an operation $+$, then:

$$\forall\, a, b \in \mathbb{S}, \quad \exists\, c \in \mathbb{S}, \quad a + b = c$$
$$\forall\, a, b, c \in \mathbb{S}, \quad (a + b) + c = a + (b + c)$$

Examples of *semigroups*: concatenation on (non-empty) lists, addition and multiplication of numbers, conjunction and disjunction of booleans, appending parts in a path or chunks of a file, combining IO actions, etc.

Monoid

A **monoid** is an algebraic structure consisting of a set, an associative binary operation, and an *identity element*.

Given a set $\mathbb{S}$, an operation $+$, and the identity element e, then:

$$\forall\, a, b \in \mathbb{S}, \quad \exists\, c \in \mathbb{S}, \quad a + b = c$$
$$\forall\, a, b, c \in \mathbb{S}, \quad (a + b) + c = a + (b + c)$$
$$\forall\, a \in \mathbb{S}, \quad e + a = a + e = a$$

Examples of *monoids*: concatenation on (possibly empty) lists, addition and multiplication on numbers, conjunction and disjunction on booleans, appending parts in a path or chunks of a file, combining IO actions, etc.

Semiring 1/2

A **semiring** is an algebraic structure consisting of a set $\mathbb{S}$, a commutative monoid $(\mathbb{S}, +, 0)$ and monoid $(\mathbb{S}, *, 1)$, such that $*$ distributes over $+$ and 0 annihilates $*$.

So, given a set $\mathbb{S}$, and two operations $+$ and $*$, and the identity elements 0 and 1, we can say:

Semiring 2/2

$$\forall\, a, b \in \mathbb{S}, \quad \exists\, c \in \mathbb{S}, \quad a + b = c$$

$$\forall\, a, b, c \in \mathbb{S}, \quad (a + b) + c = a + (b + c)$$

$$\forall\, a \in \mathbb{S}, \quad 0 + a = a + 0 = a$$

$$\forall\, a, b \in \mathbb{S}, \quad a + b = b + a$$

$$\forall\, a, b \in \mathbb{S}, \quad \exists\, c \in \mathbb{S}, \quad a * b = c$$

$$\forall\, a, b, c \in \mathbb{S}, \quad (a * b) * c = a * (b * c)$$

$$\forall\, a \in \mathbb{S}, \quad 1 * a = a * 1 = a$$

$$\forall\, a, b, c \in \mathbb{S}, \quad a * (b + c) = (a * b) + (a * c)$$

$$\forall\, a, b, c \in \mathbb{S}, \quad (a + b) * c = (a * c) + (b * c)$$

$$\forall\, a \in \mathbb{S}, \quad a * 0 = 0 * a = 0$$

Semiring Example: Bool

$$\forall\, a, b \in \mathbb{B}, \quad \exists\, c \in \mathbb{B}, \quad a \vee b = c$$

$$\forall\, a, b, c \in \mathbb{B}, \quad (a \vee b) \vee c = a \vee (b \vee c)$$

$$\forall\, a \in \mathbb{B}, \quad \bot \vee a = a \vee \bot = a$$

$$\forall\, a, b \in \mathbb{B}, \quad a \vee b = b \vee a$$

$$\forall\, a, b \in \mathbb{B}, \quad \exists\, c \in \mathbb{B}, \quad a \wedge b = c$$

$$\forall\, a, b, c \in \mathbb{B}, \quad (a \wedge b) \wedge c = a \wedge (b \wedge c)$$

$$\forall\, a \in \mathbb{B}, \quad \top \wedge a = a \wedge \top = a$$

$$\forall\, a, b, c \in \mathbb{B}, \quad a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$$

$$\forall\, a, b, c \in \mathbb{B}, \quad (a \vee b) \wedge c = (a \wedge c) \vee (b \wedge c)$$

$$\forall\, a \in \mathbb{B}, \quad a \wedge \bot = \bot \wedge a = \bot$$

Semiring Example: Types

$$\forall\, a, b \in \mathbb{T}, \quad \exists\, c \in \mathbb{T}, \quad \mathit{Either}\; a\; b \cong c$$

$$\forall\, a, b, c \in \mathbb{T}, \quad \mathit{Either}\; (\mathit{Either}\; a\; b)\; c \cong \mathit{Either}\; a\; (\mathit{Either}\; b\; c)$$

$$\forall\, a \in \mathbb{T}, \quad \mathit{Either}\; \mathit{Void}\; a \cong \mathit{Either}\; a\; \mathit{Void} \cong a$$

$$\forall\, a, b \in \mathbb{T}, \quad \mathit{Either}\; a\; b \cong \mathit{Either}\; b\; a$$

$$\forall\, a, b \in \mathbb{T}, \quad \exists\, c \in \mathbb{T}, \quad (a, b) \cong c$$

$$\forall\, a, b, c \in \mathbb{T}, \quad ((a, b), c) \cong (a, (b, c))$$

$$\forall\, a \in \mathbb{T}, \quad ((), a) \cong (a, ()) \cong a$$

$$\forall\, a, b, c \in \mathbb{T}, \quad (a, \mathit{Either}\; b\; c) \cong \mathit{Either}\; (a, b)\; (a, c)$$

$$\forall\, a, b, c \in \mathbb{T}, \quad ((\mathit{Either}\; a\; b), c) \cong \mathit{Either}\; (a, c)\; (b, c)$$

$$\forall\, a \in \mathbb{T}, \quad (a, \mathit{Void}) \cong (\mathit{Void}, a) \cong \mathit{Void}$$

Back to FP

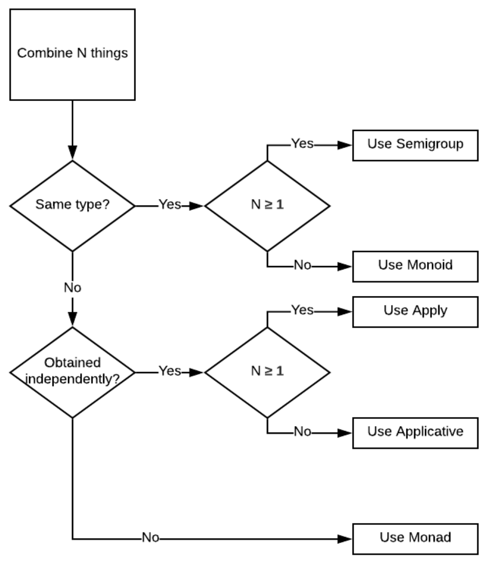
Monoid in Haskell

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


class Semigroup a => Monoid a where
  mempty :: a


instance Semigroup [a] where
  a <> b = a ++ b


instance Monoid [a] where
  mempty = []
```

Monoid Usage

```
λ> "foo" <> "bar"
"foobar"

fold :: (Foldable t, Monoid m) => t m -> m
λ> fold ["fo", "o", "bar"]
"foobar"

foldMap :: (Monoid m, Foldable t) => (a -> m) -> t a -> m
λ> foldMap Sum [1,2,3,4,5]
Sum {getSum = 15}

λ> foldMap All [True, True, True]
All {getAll = True}

λ> foldMap All [True, True, False]
All {getAll = False}
```

Combining Different Types



Monad

Monad

When viewed from the perspective of composability, monads allow us to compose things that are not obtained independently, but rather in a more sequential manner.

```
conn          <- getSqlConnection
someData      <- runSomeSelectQuery conn
                 writeToDisk someData

-- or:
getSqlConnection
    >>= runSomeSelectQuery
    >>= writeToDisk
```

Just a Monoid in the...

```haskell
-- These two functions are equivalent:
(>>=) :: m a -> (a -> m b) -> m b
join  :: m (m a) -> m a

-- Forward
ma >>= mab = join (mab <$> ma)
-- Backward
join mma = mma >>= id
```

Any **Monad** is a *Monoid*. The easiest way to look at it is by looking at the **join** operation, which takes two **m**'s and returns a single **m**, or combines two **m**'s into a single one.

See @KenScambler's excellent tweets about this at https://twitter.com/KenScambler/status/956111889519357952

Applicative

Applicative

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

class Functor f => Applicative f where
  (<*>) :: f (a -> b) -> f a -> f b
  pure :: a -> f a

-- Applicative should really be like this.
class Functor f => Apply f where
  (<*>) :: f (a -> b) -> f a -> f b

class Apply f => Applicative f where
  pure :: a -> f a
```

Applicative Example

```
op :: A -> B -> C
a :: A
b :: B

op a b :: C

fa :: f A
fb :: f B

op <$> fa <*> fb :: f C

λ> (+) <$> Just 1 <*> Just 2
Just 3

λ> (+) <$> Just 1 <*> Nothing
Nothing
```

Is this a semigroup / monoid?

```haskell
class Functor f => TupleSemigroup f where
  (<>) :: f a -> f b -> f (a, b)

class TupleSemigroup f => TupleMonoid f where
  mempty :: f ()
```

The answer is, yes, if we alter the rules a bit to say that the operation is associative up to **isomorphism** (and that using the identity element results in isomorphic structures). The claim is these classes are equivalent to **Apply** and **Applicative**. How do we prove it? By implementing these in terms of Apply and Applicative, and then implementing Apply/Applicative in terms of these clases.

Equivalence

```
-- Applicative -> Tuple*
instance Apply f => TupleSemigroup f where
  (<>) :: f a -> f b -> f (a, b)
  fa <> fb = (,) <$> fa <*> fb

instance Applicative f => TupleMonoid where
  mempty :: f ()
  mempty = pure ()

-- Tuple* -> Applicative
instance TupleSemigroup f => Apply f where
  (<*>) :: f (a -> b) -> f a -> f b
  fa2b <*> fa = (\(a2b, a) -> a2b a) <$> (fa2b <> fa)

instance TupleMonoid f => Applicative f where
  pure :: a -> f a
  pure a = const a <$> mempty
```

Wait, is TupleMonoid really a Monoid?

$$\forall\, fa, fb \in \mathbb{T}, \quad \exists\, fc \in \mathbb{T}, \quad fa <> fb \cong fc \cong f(a, b)$$

$$\forall\, fa, fb, fc \in \mathbb{T}, \quad (fa <> fb) <> fc \cong fa <> (fb <> fc)$$

$$f(a, b) <> fc \cong fa <> f(b, c)$$

$$f((a, b), c) \cong f(a, (b, c))$$

$$\forall\, fa \in \mathbb{T}, \quad mempty <> fa \cong fa <> mempty \cong fa$$

$$() <> fa \cong fa <> () \cong fa$$

$$f(a, ()) \cong f((), a) \cong fa$$

Alternative

```haskell
-- Semigroup
class Functor f => Alt f where
  (<|>) :: f a -> f a -> f a

-- Monoid
class Alt f => class Plus f where
  empty :: f a

-- Near Semiring-ish
class Applicative f, Plus f => Alternative f

-- Alternative guarantees the following laws:
(f <|> g) <*> x == (f <*> x) <|> (g <*> x)
empty <*> f == empty
```

Applicative Semiring Example

```
λ> ([(+1), (+2)] <|> [(+3), (+4)]) <*> [1,2]
[2,3,3,4,4,5,5,6]

λ> ([(+1), (+2)] <*> [1,2]) <|> ([(+3), (+4)] <*> [1, 2])
[2,3,3,4,4,5,5,6]

λ> [(+1), (+2)] <*> empty
[]
```

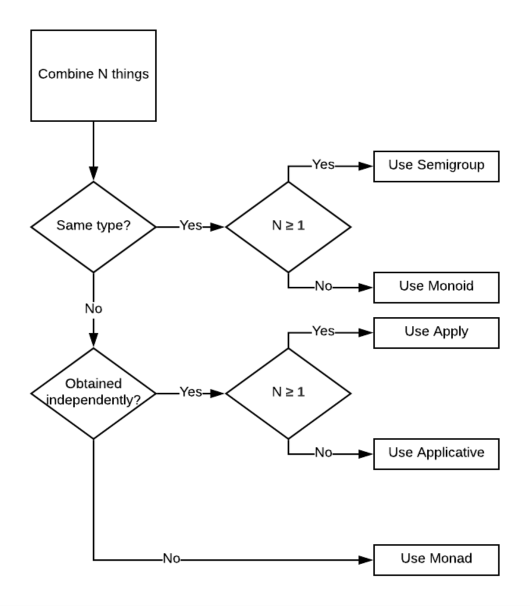
Alternative Intuition

To get some intuition, we can think of:

- **<*>** as: multiplication, conjunction, or carthesian product
- **pure** as: 1, True, or the unit set
- **<|>** as: addition, disjunction, or concatenation
- **empty** as: 0, False, or the empty set

In a way, Applicative is a higher kinded monoid, and Alternative is a higher kinded semiring (except it's not commutative in Plus).

Combining Different Types



RV is Hiring

runtime verification

You smart, energetic, self-driven, and mathematically inclined?
You believe that programs must be provably correct?
Then join our new Runtime Verification Bucharest office!

Start by impressing us with your problem solving skills:

```
while a ≠ b
  if a > b then a = a − b
            else b = b − a
```

1) What does this loop do?
2) What is the most precise loop invariant?

Send your solution and application at:
https://runtimeverification.com/careers/

Starting salary

2000 Euro / month

Thank you for listening!

cvlad

vladciobanu

cvlad.info