Combinatorial Species

Jonathan Dygert

October 31, 2019

Table of contents

Combinatorial species were introduced by French mathematician André Joyal in his 1981 paper, "Une théorie combinatoire des séries formelles", to unify branches of combinatorics. The theory of combinatorial species is still young, and the community is not yet in consensus on its utility.

Combinatorial species provide a great mathematical representation for data structures. With species, we can

- compare different structures
- reason about their possible contents
- perform operations such as differentiation
- develop transformations between structures

We can also use combinatorial species as a framework for automated testing. If we claim that the output of a function will always have a certain property, we can automatically generate all possible inputs of a given size and see if the property holds. (Demonstrated later)

Linked List

As an example of a species, we can represent a linked list as follows:

$$\underbrace{L}_{\text{A list}} \quad \underbrace{=}_{\text{is}} \quad \underbrace{1}_{\text{empty}} \quad \underbrace{+}_{\text{or}} \quad \underbrace{X}_{\text{an element}} \quad \underbrace{\bullet}_{\text{and}} \quad \underbrace{L}_{\text{another list}}$$

Expand this recursive definition to get $L = 1 + X + X^2 + X^3 + \cdots$.
Thus, a list can have 0 elements, 1 element, 2 elements, and so on.
This also shows us that a linked list is isomorphic to an array-based list.

What is a species?

Informally, a species is "a family of structures parameterized by a set of labels which identify locations in the structures." [3] A species maps a given finite set of labels into a set of structures containing those labels. The species must be ignorant of the contents of the labels, and it must be possible to relabel without changing the structure itself.

Definition

A *species* F is a pair of mappings, $F_\bullet$ and $F_\leftrightarrow$, where

- $F_\bullet$ maps a finite set U to a finite set of structures, $F_\bullet[U]$, which can be "built from" the labels, commonly called the set of F-structures over U.
- $F_\leftrightarrow$ "lifts" a bijection $\sigma : U_1 \leftrightarrow U_2$ between two label sets onto a bijection between F-structures,

$$F_\leftrightarrow[\sigma] : F_\bullet[U_1] \leftrightarrow F_\bullet[U_2]$$

This mapping must be functorial. That is, it should satisfy

- $F_\leftrightarrow[id] = id$
- $F_\leftrightarrow[\sigma_1 \circ \sigma_2] = F_\leftrightarrow[\sigma_1] \circ F_\leftrightarrow[\sigma_2]$

where id is the identity mapping and $\circ$ is composition.

[1, 3]

$F_\bullet$ is typically written as F for simplicity. Because $F_\leftrightarrow$ is functorial, the values of the labels are unimportant. The important property of the labels is that they are distinct, so that we can distinguish different locations of the structure.

A species corresponds to a data structure generic over a single unrestricted type parameter, such as Java's `LinkedList<T>`. However, the labels do not correspond to the data held by the data structure. Instead, they should be thought of as names for the locations within the structure.

As we explain species, we will implement them in the programming language Haskell. Here are the extensions and imports used.

```haskell
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE TypeApplications #-}

import Data.List (foldl', delete, tails, inits, permutations)
import Control.Monad (guard)
```

The first part of a species, $F_\bullet$, takes a list of labels and creates a list of structures. We say a type is enumerable if we have a function enumerate which takes a list of labels and produces a list of structures.

```
class Enumerable f where
  enumerate :: [a] -> [f a]
```

The second part of a species, the mapping $F_\leftrightarrow$, corresponds to the functor type class which is already present in Haskell.

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

Zero

0 yields no structures no matter what labels it is given. This is a type that has no values and is the type of infinite loops and return type of functions that terminate the program.

Zero

0 yields no structures no matter what labels it is given. This is a type that has no values and is the type of infinite loops and return type of functions that terminate the program.

```haskell
data Zero a

instance Functor Zero where
  fmap = undefined

instance Enumerable Zero where
  enumerate _ = []
```

One

1 yields a single structure when given no labels. It holds no data and has a single value. This is like the return type `void` in Java, which can be thought of as nothing, or more precisely, a unit type that can always be created from nothing.

One

1 yields a single structure when given no labels. It holds no data and has a single value. This is like the return type `void` in Java, which can be thought of as nothing, or more precisely, a unit type that can always be created from nothing.

```haskell
data One a = One

instance Functor One where
  fmap _ One = One

instance Enumerable One where
  enumerate [] = [One]
  enumerate _ = []
```

Singleton

X yields a single structure when given a single label. It holds exactly one piece of data. This corresponds to the identity type, with no difference from the type it holds.

Singleton

X yields a single structure when given a single label. It holds exactly one piece of data. This corresponds to the identity type, with no difference from the type it holds.

```haskell
data X a = X a

instance Functor X where
  fmap f (X a) = X (f a)

instance Enumerable X where
  enumerate [x] = [X x]
  enumerate _ = []
```

Sum

Species sum represents the word "or" and corresponds to a disjoint (tagged) union. A structure of $F + G$ is either an F-structure or a G-structure along with a tag that specifies which. Formally,

$$(F + G)[U] = F[U] \uplus G[U]$$

where $\uplus$ is disjoint union over sets. [3]

Sum (cont.)

```haskell
infixl 6 :+

data (f :+ g) a = Inl (f a) | Inr (g a)

instance (Functor f, Functor g) => Functor (f :+ g) where
  fmap h (Inl x) = Inl (fmap h x)
  fmap h (Inr x) = Inr (fmap h x)

instance (Enumerable f, Enumerable g) => Enumerable (f :+ g) where
  enumerate ls = map Inl (enumerate ls) ++ map Inr (enumerate ls)
```

Natural Numbers

We can generalize 0 and 1 to any natural number n, which takes no labels and produces n distinct structures. n is the species sum of n total 1 species.

A Boolean (with values true or false) would have the species $2 = 1 + 1$

Product

Species product represents the word "and", and corresponds to a pair. A structure of $(F \bullet G)[U]$ will be a pair of structures, one from F and one from G, with the labels of U split between the pair. That is,

$$(F \bullet G)[U] = \{ (x, y) \mid U_1 \uplus U_2 = U,\ x \in F[U_1],\ y \in G[U_2] \}$$

Product (cont.)

```haskell
infixl 7 :*

data (f :* g) a = f a :* g a

instance (Functor f, Functor g) => Functor (f :* g) where
  fmap h (x :* y) = fmap h x :* fmap h y

instance (Enumerable f, Enumerable g) => Enumerable (f :* g) where
  enumerate ls = [ x :* y
                 | (fls, gls) <- splits ls
                 , x <- enumerate fls
                 , y <- enumerate gls
                 ]

splits :: [a] -> [([a], [a])]
splits [] = [([],[])]
splits (x : xs) = (map . first) (x:) ss ++ (map . second) (x:) ss
  where ss = splits xs
        first f (x, y) = (f x, y)
        second f (x, y) = (x, f y)
```

Properties

Species sum and product have the typical properties we expect of these operations. They are associative, commutative, and have identities of zero and one, respectively. Multiplication distributes over addition, and zero is an annihilator for multiplication.

Regular Species

The species we have seen so far are all regular. Any species we construct from these using sum, product, and self-reference will be regular.

Definition

A species is *regular* if and only if it can be expressed in terms of 0, 1, X, $+$, $\bullet$, and μ (self-reference). [3]

Regular Species

The species we have seen so far are all regular. Any species we construct from these using sum, product, and self-reference will be regular.

Definition

A species is *regular* if and only if it can be expressed in terms of 0, 1, X, $+$, $\bullet$, and μ (self-reference). [3]

A species is also regular when it has no non-trivial symmetries. In other words, all of its possible structures will change if the labels are rearranged.

Definition

A species F is *regular* if and only if, for all structures $f \in F[U]$, the only permutation σ such that $F_{\leftrightarrow}[\sigma](f) = f$ is the identity permutation. [3]

Set

E is the species of a set, or unordered collection. For any set of labels, there will be exactly one structure, the set itself. E is irregular, since it has every possible symmetry. No matter what order the labels are in, we will get the same set.

Set

E is the species of a set, or unordered collection. For any set of labels, there will be exactly one structure, the set itself. E is irregular, since it has every possible symmetry. No matter what order the labels are in, we will get the same set.

```
newtype Bag a = Bag [a]
  deriving Functor

instance Eq a => Eq (Bag a) where
  Bag xs == Bag ys = xs `subBag` ys && ys `subBag` xs
    where subBag b = null . foldl' (flip delete) b

instance Enumerable Bag where
  enumerate ls = [Bag ls]
```

Cycle

C is the species of a directed cycle. This is a nonempty loop that goes in a specific direction. C is irregular because any permutation that rotates the labels along the loop is symmetric.

Cycle

C is the species of a directed cycle. This is a nonempty loop that goes in a specific direction. C is irregular because any permutation that rotates the labels along the loop is symmetric.

```haskell
newtype Cycle a = Cycle [a]
  deriving Functor

instance Eq a => Eq (Cycle a) where
  Cycle xs == Cycle ys = any (== ys) (rotations xs)
    where rotations xs = zipWith (++) (tails xs) (inits xs)

instance Enumerable Cycle where
  enumerate [] = []
  enumerate (x : xs) = (map (Cycle . (x:)) . permutations) xs
```

Composition

Species composition represents the word "of". $F \circ G$ means F-structures made of G-structures. To form a structure of $(F \circ G)[U]$, split U into non-empty partitions and send each of those partitions separately to G. Then, collect all those G-structures and send them to F.

$L \circ E$ is a list of sets

$R = 1 + X \bullet (L \circ R)$ is a rose tree, a tree where the nodes have an unspecified number of children.

```haskell
newtype (f :. g) a = C (f (g a))

instance (Functor f, Functor g) => Functor (f :. g) where
  fmap h (C fg) = C $ (fmap . fmap) h fg

instance (Enumerable f, Enumerable g) => Enumerable (f :. g) where
  enumerate ls = [C y | p <- partitions ls
                      , gs <- mapM enumerate p
                      , y <- enumerate gs]

partitions :: [a] -> [[[a]]]
partitions [] = [[]]
partitions (x : xs) = [ (x : ys) : p
                      | (ys, zs) <- splits xs
                      , p <- partitions zs
                      ]
```

Differentiation

Informally, taking the derivative of a species puts a single "hole" in the structures, a distinguished location not holding any data. That is,

$$F'[U] = F[U \cup \{*\}]$$

where $*$ is a label distinct from all elements of U. [3]

Differentiation

Informally, taking the derivative of a species puts a single "hole" in the structures, a distinguished location not holding any data. That is,

$$F'[U] = F[U \cup \{*\}]$$

where $*$ is a label distinct from all elements of U. [3]

$L' = L^2$. If we put a hole in a list, we split it into two lists.

$E' = E$. If we put a hole in a set, it does not change the set.

$C' = L$. If we put a hole in a cycle, we break the loop into a list.

```haskell
newtype Diff f a = Diff (f (Maybe a))
  deriving Functor

instance Enumerable f => Enumerable (Diff f) where
  enumerate ls = map Diff (enumerate (Nothing : map Just ls))
```

Species differentiation holds its expected properties compared to differentiation in calculus.

- $1' = 0$
- $X' = 1$
- $(F + G)' = F + G$
- $(F \bullet G)' = F \bullet G' + F' \bullet G$
- $(F \circ G)' = (F' \circ G) \bullet G'$

Now we have the basics of species defined and can demonstrate their uses in Haskell!

First, we add an instance for Haskell's lists and make everything printable.

```
instance Enumerable [] where
  enumerate = permutations

deriving instance Show (Zero a)
deriving instance Show (One a)
deriving instance Show a => Show (X a)
deriving instance (Show (f a), Show (g a)) => Show ((f :+ g) a)
deriving instance (Show (f a), Show (g a)) => Show ((f :* g) a)
deriving instance Show a => Show (Cycle a)
deriving instance Show a => Show (Bag a)
deriving instance Show (f (g a)) => Show ((f :. g) a)
deriving instance Show (f (Maybe a)) => Show (Diff f a)
```

Now we can try out enumerating our species.

```
> enumerate @Cycle "abc"
[Cycle "abc",Cycle "acb"]

> enumerate @Bag "abc"
[Bag "abc"]

> enumerate @[] "abc"
["abc","bac","cba","bca","cab","acb"]

> length $ enumerate @[] [1..6]
120

> length $ enumerate @(Diff Cycle) [1..6]
120
```

The species $E \bullet E$ or E^2 corresponds to the powerset.

```
> mapM_ print $ enumerate @(Bag :* Bag) "abc"
Bag "abc" :* Bag ""
Bag "ab" :* Bag "c"
Bag "ac" :* Bag "b"
Bag "a" :* Bag "bc"
Bag "bc" :* Bag "a"
Bag "b" :* Bag "ac"
Bag "c" :* Bag "ab"
Bag "" :* Bag "abc"
```

The species $E \circ E$ corresponds to set partitioning.

```
> mapM_ print $ enumerate @(Bag :. Bag) "abc"
C (Bag [Bag "abc"])
C (Bag [Bag "ab",Bag "c"])
C (Bag [Bag "ac",Bag "b"])
C (Bag [Bag "a",Bag "bc"])
C (Bag [Bag "a",Bag "b",Bag "c"])
```

Suppose we have a custom datatype.

```
data Family a = Monkey Bool a
              | Group a [Family a]
  deriving Show
```

We can represent this using species we already defined.

```
type SBool = One :+ One
newtype SFamily a = SFamily ((SBool :* X :+ X :* [] :. SFamily) a)
  deriving Show

instance Enumerable SFamily where
  enumerate = map SFamily . enumerate
```

We can convert from the species to our datatype.

```
bool :: SBool a -> Bool
bool (Inl One) = False
bool (Inr One) = True

family :: SFamily a -> Family a
family (SFamily sf) = case sf of
  Inl (b :* (X a)) -> Monkey (bool b) a
  Inr ((X a) :* (C sfs)) -> Group a (map family sfs)
```

Now we can easily generate values of our datatype.

```
> mapM_ print $ map family $ enumerate "ab"
Group 'a' [Monkey False 'b']
Group 'a' [Monkey True 'b']
Group 'a' [Group 'b' []]
Group 'b' [Monkey False 'a']
Group 'b' [Monkey True 'a']
Group 'b' [Group 'a' []]
```

We can use this to test functions that operate on our datatype. Suppose we have a function that removes all false monkeys and a function that checks whether a family has any false monkeys.

```
removeFalseMonkeys :: Family a -> Family a
hasFalseMonkeys :: Family a -> Bool
```

We can test the removal function by generating families as input. The list below will contain all families of at most size 5 which fail the test.

```
removalFailures :: [Family Int]
removalFailures = do
  size <- [1..5]
  let labels = [1..size]
  input <- family <$> enumerate labels
  let output = removeFalseMonkeys fam
  guard $ hasFalseMonkeys output
  return input
```

We can generalize this to arbitrary functions and properties.

```
failures :: Enumerable f
         => (f Int -> g Int)
         -> (g Int -> a)
         -> (a -> Bool)
         -> Int
         -> [g Int]
failures conversion function badProperty sizeLimit = do
  size <- [1..sizeLimit]
  let labels = [1..size]
  input <- conversion <$> enumerate labels
  let output = function input
  guard $ badProperty output
  return input
```

Now we can represent our test more clearly.

```
removalFailures :: [Family Int]
removalFailures = failures family removeFalseMonkeys hasFalseMonkeys 5
```

We can also now easily test other functions. For example, we can test sorting a list.

```
sort :: [Int] -> [Int]
isSorted :: [Int] -> Bool

sortFailures :: [[Int]]
sortFailures = failures id sort (not . isSorted) 4
```

This example gives us all failures of sorting a list of at most 4 integers.

Conclusion

Combinatorial species are an interesting young branch of combinatorics, with many extensions such as weighted, multi-sort, virtual, and molecular species. [3] Species provide computer scientists a great method for understanding data structures and can be applied to automated testing to help improve software correctness.

Ideas for improving the project:

- Add type operator for μ to make anonymous recursive types possible
- Extend the testing framework to make it easier to test many properties
- Add typeclass to group together Functor, Enumerable, 0, 1, $+$, $\bullet$, etc.

Bibliography

[1] François Bergeron et al. *Combinatorial species and tree-like structures*. Vol. 67. Cambridge University Press, 1998.

[2] André Joyal. "Une théorie combinatoire des séries formelles". In: *Advances in Mathematics* 42.1 (1981), pp. 1–82. DOI: 10.1016/0001-8708(81)90052-9.

[3] Brent Yorgey. "Species and Functors and Types, Oh My!" In: vol. 45. Nov. 2010, pp. 147–158. DOI: 10.1145/2088456.1863542.