Submitted to the 2010 Haskell Symposium

Functional Pearl: Species and Functors and Types, Oh My!

Brent Yorgey
University of Pennsylvania
byorgey@cis.upenn.edu

Abstract

The theory of *combinatorial species*, although invented as a purely mathematical formalism to unify much of combinatorics, can also serve as a powerful and expressive language for talking about algebraic data types. With potential applications to automatic test generation, generic programming, and language design, the theory deserves to be much better known in the functional programming community. This paper aims to teach the theory of combinatorial species using motivation and examples from the world of functional programming. It also introduces the *species* library, available on Hackage, which is used to illustrate the concepts introduced and can serve as a platform for continued study and research.

General Terms Combinatorial Species, Algebraic Data Types

1. Introduction

The theory of *combinatorial species* was invented by André Joyal in 1981 [15] as an elegant framework for understanding and unifying much of enumerative combinatorics. Since then, mathematicians have continued to develop the theory, proving many fundamental results and producing at least one excellent reference text on the topic [4]. Many connections to computer science and functional programming have been pointed out in elegant detail, notably by Flajolet, Salvy, and Zimmermann [11, 12]. However, despite all, this beautiful theory is still not widely known among functional programmers.

Suppose Jane Programmer has has created the following $Widget$ type, which will form a core data structure in her groundbreaking widget software for the hot new jPhone platform:

$$\begin{aligned}\mathbf{data}\ Widget\ a = {} & Tagged\ Bool\ a \\ \mid {} & Children\ [\,Widget\ a\,]\end{aligned}$$

That is, a widget consists either of a $Bool$ paired with an a, or a list of widgets. While continuing to develop her program, Jane might have the following questions:

- *How can I exhaustively enumerate all $Widget\ a$ values of a certain size, or up to a certain size?* For example, Jane might want to do this in order to exhaustively test a function taking $Widget$s as input, or simply because an algorithm she is implementing requires an enumeration of possible $Widget$ values. She could use the SmallCheck tool [21] which can do exhaustive testing in this way. She could also write a $Widget$-enumerating function by hand, possibly with the help of a generic programming library, but this would be somewhat ad-hoc. And what if she decides, say, that the ordering of the children of a $Widget$ does not matter? SmallCheck is not designed to take this possibility into account, and generic programming libraries cannot handle this in a nice, uniform way either.

  As we will see in Section 4, Jane can use the *species* library to do this in an automatic yet flexible way.
- *How can I randomly generate $Widget$ values of a certain size?* Again, she may wish to do this either for testing purposes, or perhaps as inputs to some sort of Monte Carlo algorithm or simulation. QuickCheck [9] is the go-to tool for automatic testing via random generation in Haskell; however, it is difficult to use it to generate random instances of recursive data structures, and writing the required $Arbitrary$ instances is still somewhat ad-hoc. There are generic programming libraries which can automatically generate $Arbitrary$ instances for QuickCheck, but they can't fundamentally do any better than QuickCheck can at dealing with recursive types.

  There is a well-developed theory on automatically deriving random generation routines from combinatorial species descriptions [7, 12]; although the *species* library in particular does not implement it yet, it is planned as future work.
- *How can I count the number of $Widget\ a$ values of a certain size?* Questions of this sort are useful in, for example, predicting performance or considering the feasibility of data structures or algorithms. As we'll see in Section 7, counting the number of unique structures described by a certain combinatorial species can be accomplished by mapping the species onto generating functions.
- *How can I easily construct isomorphisms to mediate between different representations of data?* Perhaps $Widget$s need to be put into a different format before being shipped off to some other interface. As we'll see beginning in Section 3, after boiling data structures down to their essence as combinatorial species expressions, it is easy to manipulate them according to various algebraic laws and automatically derive isomorphisms between them.

The theory of combinatorial species provides an elegant framework for answering all these questions and more. The core idea is that species expressions form an abstract syntax which we can interpret in a number of different ways, depending on what we wish to compute.

The literate Haskell source for this paper is available from `http://www.cis.upenn.edu/~byorgey/papers/species-pearl.lhs`.

2. Combinatorial species

Intuitively, a species describes a *family of structures*, parameterized by a set of *labels* out of which the structures are built. In programming language terms, a species is like a polymorphic type constructor with a single type argument.

Definition 1. A *species* F consists of a pair of mappings, $\mathsf{F}_\bullet$ and $\mathsf{F}_\leftrightarrow$, with the following properties:

- $\mathsf{F}_\bullet$, given a finite set U of *labels*, sends U to a finite set of *structures* $\mathsf{F}_\bullet[U]$ which can be "built out of" the given labels. We call $\mathsf{F}_\bullet[U]$ the set of F-*structures* with *support* U, or sometimes just F-structures *over* U.
- $\mathsf{F}_\leftrightarrow$, given a bijection $\sigma : U_1 \xleftrightarrow{\sim} U_2$ between two label sets U_1 and U_2 (*i.e.* a *relabeling*), "lifts" σ to a bijection between F-structures,

  $$\mathsf{F}_\leftrightarrow[\sigma] : \mathsf{F}_\bullet[U_1] \xleftrightarrow{\sim} \mathsf{F}_\bullet[U_2].$$

  Moreover, this lifting must be *functorial*: the identity relabeling should lift to the identity on structures, and composition of relabelings should commute with lifting.

We usually omit the subscript on $\mathsf{F}_\bullet$ when it is clear from the context.

For example, Figure 1 shows an example of lifting a bijection σ between labels to a relabeling of binary trees.

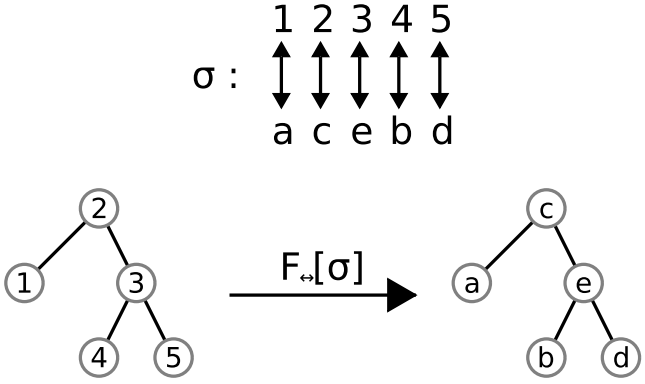


Figure 1. Relabeling structures

Fans of category theory will recognize a much more concise version of this definition: a species is an endofunctor on $\mathbb{B}$, the category of finite sets with bijections as arrows. You won't need to know any category theory to understand this paper or to make use of the *species* library; however, the categorical point of view does add considerable conciseness and depth to the study of combinatorial species.

The ability to relabel structures means that the actual labels we use don't matter; we get "the same structures", up to relabeling, for any label sets of the same size. We might say that species are *parametric* in the label sets of a given size. In particular, F's action on all label sets of size n is completely determined by its action on any particular such set: if $|U_1| = |U_2|$ and we know $\mathsf{F}[U_1]$, we can determine $\mathsf{F}[U_2]$ by lifting any bijection between U_1 and U_2. So we often take the finite set of natural numbers $\{1, \ldots, n\}$ (also written $[n]$) as *the* canonical label set of size n, and write $F[n]$ for the set of F-structures built from this set.

Although it is not required by the definition, we will generally only consider structures which contain each label exactly once. At first glance this may seem strange: after all, values of type $[Int]$ are not required to contain every Int exactly once! The confusion stems from the tempting but incorrect assumption that labels play the role of the "data" held by data structures. Instead, labels should be thought of as *names* for the locations or "holes" within a structure. The idea is that data structures can be decomposed into a *shape* together with some sort of *content* [1, 14]. In this case, a *labeled shape* is an instance of a data structure containing distinct labels, and the content can be specified by a mapping from labels to data (which need not be injective). Requiring each label to occur exactly once in a structure serves the dual purposes of controlling the structure's size, and giving us a way to refer to the holes in a structure when filling it with content.

Given these purposes, however, one might well wonder why we need labels at all. To specify the size of a structure we would need only a single natural number, not a whole set; and it is easy to imagine other ways to fill structures with data without having to give names to holes. For example, given a binary tree shape we can unambiguously fill it with a list of data using an inorder traversal; we can even refer to individual holes by paths from the root. And indeed, for so-called *regular* species (Section 3) we do not really need the labels; we will make this more precise when we talk about unlabeled species in Section 5. However, for non-regular species (Section 6) we really do need the labels to make the story about shapes precise.

3. Regular species

Although the formal definition of species is good to keep in mind as a source of intuition, in practice we take an algebraic approach, building up complex species from a few primitive species and species operations. We start our tour of the species menagerie with *regular* species. These should seem like old friends to functional programmers, since they correspond closely to the algebraic data types found in languages like ML, OCaml, and Haskell. We'll step back to define regular species more abstractly in Section 3.2, but first let's see how to build them!

3.1 Basic regular species

For each primitive species or species operation, we will define a corresponding Haskell data type embodying the $\mathsf{F}_\bullet$ mapping—that is, values of the type will correspond to F-structures. The $\mathsf{F}_\leftrightarrow$ mapping, in turn, can be elegantly expressed by an instance of the $Functor$ type class, whose method $fmap :: (a \to b) \to f\ a \to f\ b$ shows how to relabel a structure $f\ a$ by applying a relabeling map $a \to b$ to each of its labels. (Note that we can also use $fmap$ to fill in a labeled shape with content, by applying it to a mapping from labels to data.) For each species we also exhibit a method to enumerate all distinct labeled structures on a given set of labels, via an instance of the $Enumerable$ type class shown in Figure 2. The actual $Enumerable$ type class used in the *species* library is more sophisticated, but this will serve as an introduction.

```
class Enumerable f where
  enumerate :: [a] → [f a]
```

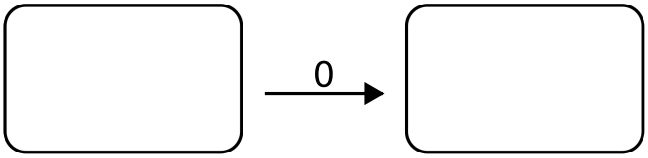
Figure 2. The $Enumerable$ type class

Finally, for each species or species operation we also exhibit a picture as an aid to intuition. These pictures are not meant to be formal, but they will generally conform to the following rules:

- The left-hand side of each picture shows a canonical set of labels (depicted as circled numbers), either of a "random" size, or of a size that is "interesting" for the species being defined. Although the canonical label set $[n]$ is used, of course the labels could be replaced by any others.
- In the middle is an arrow labeled with the name of the species being defined.

- On the right-hand side is a set of structures, or some sort of schematic depiction of the construction of a "typical" structure (the species then being understood to construct such structures "in all possible ways"). When the name of a species is superimposed on a set of labels, it represents a structure of the given species built from the labels.

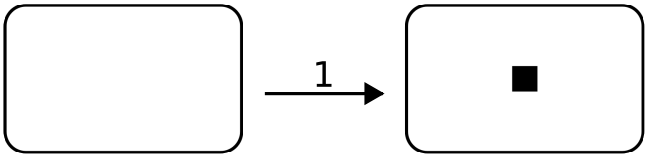
Zero The species 0 (Figure 3) corresponds to the constantly void type constructor. That is, it yields no structures no matter what labels it is given as input. Note that we are forced to cheat a bit in the *Functor* instance for 0, since Haskell does not allow empty case expressions.




```
data 0 a
instance Functor 0 where
  fmap = ⊥
instance Enumerable 0 where
  enumerate _ = []
```

Figure 3. The primitive species 0

One The species 1 (Figure 4) yields a single unit structure when applied to an empty set of labels, and no structures otherwise. In other words, there is exactly one structure of type $1\ a$, and it contains no holes where values of type a can be stored. It corresponds to nullary constructors in algebraic data types. Note that the unit structure built by 1 is shown in Figure 4 as a filled square, to emphasize the fact that it contains no labels.



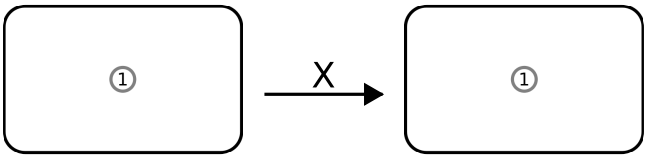

```
data 1 a = 1
  deriving Show
instance Functor 1 where
  fmap _ 1 = 1
instance Enumerable 1 where
  enumerate [] = [1]
  enumerate _ = []
```

Figure 4. The primitive species 1

The species of singletons The species of *singletons*, X (Figure 5), yields a single structure when applied to a singleton label set, and no structures otherwise. That is, X corresponds to the identity type constructor, which has exactly one way of building a structure with a single hole.

Species sum We define species sum (Figure 6) to correspond to type sum, *i.e.* disjoint (tagged) union. Given species F and G and a set of labels U, the set of $(\mathsf{F}+\mathsf{G})$-structures over U is the disjoint union of the sets of F- and G-structures over U:

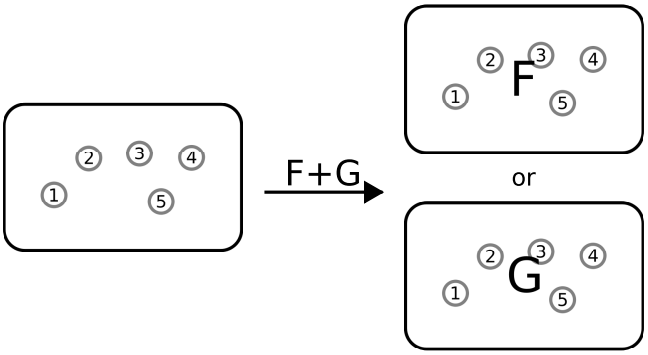
$$(\mathsf{F}+\mathsf{G})[U] = F[U] \uplus G[U].$$




```
data X a = X a
  deriving Show
instance Functor X where
  fmap f (X a) = X (f a)
instance Enumerable X where
  enumerate [x] = [X x]
  enumerate _   = []
```

Figure 5. The species X of singletons

In other words, an $(\mathsf{F}+\mathsf{G})$-structure is either an F-structure or a G-structure, along with a tag specifying which. For example, $1+\mathsf{X}$ corresponds to the familiar *Maybe* type constructor.




```
infixl 6 +
data (f + g) a = Inl (f a) | Inr (g a)
  deriving Show
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)
(+) :: (f a → g a) → (h a → j a)
    → (f + h) a    → (g + j) a
(fg + hj) (Inl fa) = Inl (fg fa)
(fg + hj) (Inr ha) = Inr (hj ha)
```

Figure 6. Species sum

It is not hard to verify that, up to isomorphism, 0 is the identity for species addition, and that $+$ is associative and commutative. We can also generalize 0 and 1 by defining the species n, for each $n \geqslant 0$, to be the species which generates n distinct structures for the empty label set, and no structures for any nonempty label set; n is isomorphic to $\underbrace{1+\cdots+1}_{n}$. For example, 2 corresponds to the constantly *Bool* type constructor, $\mathbf{data}\ CBool\ a = CBool\ Bool$.

Since the algebraic laws for species sum correspond directly to generic isomorphisms between structures, we can directly represent the laws as Haskell code. We define a type of embedding-projection pairs, shown in Figure 7. A value of type $f \leftrightarrow g$ is an isomorphism between f and g, witnessed by a pair of functions, one in each direction. We also define the identity isomorphism as well as composition and inversion of isomorphisms.

```
infix 1 ↔
data f ↔ g = (↔) { to   :: ∀a ∘ f a → g a,
                   from :: ∀a ∘ g a → f a
                 }
ident :: f ↔ f
ident = id ↔ id

(⋙) :: (f ↔ g) → (g ↔ h) → (f ↔ h)
(fg ↔ gf) ⋙ (gh ↔ hg) = (gh ∘ fg) ↔ (gf ∘ hg)

inv :: (f ↔ g) → (g ↔ f)
inv (fg ↔ gf) = gf ↔ fg
```

Figure 7. Isomorphisms

Armed with these definitions, Figure 8 presents the algebraic laws for sum in Haskell form. The one technical issue to note is that for the congruences $inSumL$ and $inSumR$, we must be careful to use lazy pattern matches, since the isomorphism between f and g may not needed. Always forcing the proof to weak-head normal form can cause isomorphisms between recursively defined structures to diverge, since a recursive call must be made before even learning whether the isomorphism will be needed.

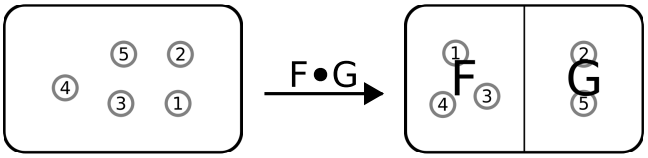
```
sumIdL :: 0 + f ↔ f
sumIdL = (λ(Inr x) → x) ↔ Inr

sumComm :: f + g ↔ g + f
sumComm = swapSum ↔ swapSum
  where swapSum (Inl x) = Inr x
        swapSum (Inr x) = Inl x

sumAssoc :: f + (g + h) ↔ (f + g) + h
sumAssoc = reAssocL ↔ reAssocR
  where reAssocL (Inl x)         = Inl (Inl x)
        reAssocL (Inr (Inl x))   = Inl (Inr x)
        reAssocL (Inr (Inr x))   = Inr x
        reAssocR (Inl (Inl x))   = Inl x
        reAssocR (Inl (Inr x))   = Inr (Inl x)
        reAssocR (Inr x)         = Inr (Inr x)

inSumL :: (f ↔ g) → (f + h ↔ g + h)
inSumL∼(fg ↔ gf) = (fg + id) ↔ (gf + id)

inSumR :: (f ↔ g) → (h + f ↔ h + g)
inSumR∼(fg ↔ gf) = (id + fg) ↔ (id + gf)
```

Figure 8. Algebraic laws for sum

Species product Just as species sum corresponds to type sum, we define species product (Figure 9) to correspond to type product. However, we must be careful to preserve the invariant that each label occurs exactly once in the resulting structures. So, to form an $(\mathsf{F} \bullet \mathsf{G})$-structure over U, we split U into two disjoint subsets, and form the ordered pair of an F-structure built from the first subset and a G-structure built from the second. Doing this in all possible ways yields the species $(\mathsf{F} \bullet \mathsf{G})$. Formally,

$$(\mathsf{F} \bullet \mathsf{G})[U] = \sum_{U = U_1 \uplus U_2} \mathsf{F}[U_1] \times \mathsf{G}[U_2],$$

where $\sum$ denotes disjoint union and $\times$ denotes Cartesian product.




```
infixl 7 •
data (f • g) a = f a • g a
  deriving Show

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)

(•) :: (f a → g a) → (h a → j a)
    → (f • h) a    → (g • j) a
(fg • hj) (x • y) = fg x • hj y
```

Figure 9. Species product

For example, $\mathsf{X} \bullet \mathsf{X}$ (which can be abbreviated X^2) is the species of *ordered pairs*. X yields no structures unless it is given a single label, so the only way to get an $\mathsf{X} \bullet \mathsf{X}$ structure is if we start with two labels and partition them into two singleton sets to pass on to the X's. Of course, there are two ways to do this, reflecting the two possible orderings of the labels. Similarly, X^3 is the species of ordered triples, with $3! = 6$ orderings for the labels, and so on.

Up to isomorphism, 1 is an identity for species product, and 0 is an annihilator. It is also not hard to check that $\bullet$ is associative and commutative, and distributes over $+$ (as usual, all up to isomorphism). Thus, species form a commutative semiring. The isomorphisms justifying these algebraic laws are shown in Figure 10.

Fixed points and the implicit species theorem If we add a least fixed point operator μ, we now get the *regular types* or *algebraic data types* familiar to any functional programmer [19]. For example, the species L of *linear orderings* (or *lists* for short) can be defined as

$$\mathsf{L} = \mu\ell.1 + \mathsf{X} \bullet \ell.$$

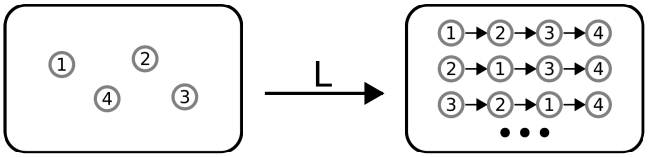
That is, a list is either empty (1) or an element paired with a list ($\mathsf{X} \bullet \ell$). For any set U of labels, $\mathsf{L}[U]$ is the set of all linear orderings of U (Figure 11); of course, if $|U| = n$ then $|\mathsf{L}[U]| = n!$.

Actually, mathematicians would not write $\mathsf{L} = \mu\ell.1 + \mathsf{X} \bullet \ell$, but simply

$$\mathsf{L} = 1 + \mathsf{X} \bullet \mathsf{L}.$$

```
prodIdL :: 1 • f ↔ f
prodIdL = (λ(1 • f) → f) ↔ (λf → (1 • f))

prodIdR :: f • 1 ↔ f
prodIdR = (λ(f • 1) → f) ↔ (λf → (f • 1))

prodAbsorbL :: 0 • f ↔ 0
prodAbsorbL = ⊥ ↔ ⊥

prodComm :: f • g ↔ g • f
prodComm = swapProd ↔ swapProd
  where swapProd (f • g) = g • f

prodAssoc :: f • (g • h) ↔ (f • g) • h
prodAssoc = reAssocL ↔ reAssocR
  where reAssocL (f • (g • h)) = (f • g) • h
        reAssocR ((f • g) • h) = f • (g • h)

prodDistrib :: f • (g + h) ↔ (f • g) + (f • h)
prodDistrib = distrib ↔ undistrib
  where distrib (f • Inl g)       = Inl (f • g)
        distrib (f • Inr h)       = Inr (f • h)
        undistrib (Inl (f • g)) = f • Inl g
        undistrib (Inr (f • h)) = f • Inr h

inProdL :: (f ↔ g) → (f • h ↔ g • h)
inProdL~(fg ↔ gf) = (fg • id) ↔ (gf • id)

inProdR :: (f ↔ g) → (h • f ↔ h • g)
inProdR~(fg ↔ gf) = (id • fg) ↔ (id • gf)
```

Figure 10. Algebraic laws for product




```
instance Enumerable [] where
  enumerate = Data.List.permutations

listRec :: [] ↔ 1 + (X • [])
listRec = unroll ↔ roll
  where unroll []                = Inl 1
        unroll (x : xs)          = Inr (X x • xs)
        roll   (Inl 1)           = []
        roll   (Inr (X x • xs)) = x : xs
```

Figure 11. The species L of linear orderings

This is not because they are being sloppy, but because of the *implicit species theorem* [4], which is a combinatorial analogue of the implicit function theorem from analysis. Suppose we have an implicit species equation $Y = H(X, Y)$. Then, if

- Y yields no structures on the empty label set ($H(0, 0) = 0$), and
- Y is not trivially reducible to itself ($\frac{\partial H}{\partial Y}(0, 0) = 0$),

the implicit species theorem guarantees that there is a unique solution for Y with $Y(0) = 0$.

The species L, as defined above, does not actually meet these criteria, since L does yield a structure on the empty label set. However, if we let L_+ denote the species of nonempty lists, with $\mathsf{L}_+ + 1 = \mathsf{L}$, then we have $\mathsf{L}_+ = \mathsf{X} \bullet (\mathsf{L}_+ + 1)$, which does meet the criteria and hence has a unique solution, so in fact L is uniquely determined as well, and we are justified in forgetting about μ and simply manipulating the implicit equation $\mathsf{L} = 1 + X \bullet \mathsf{L}$ however we like. For example, expanding L in its own definition, we find that

$$\begin{aligned}\mathsf{L} &= 1 + \mathsf{X} \bullet \mathsf{L} \\ &= 1 + \mathsf{X} \bullet (1 + \mathsf{X} \bullet \mathsf{L}) \\ &= 1 + \mathsf{X} + \mathsf{X}^2 \bullet \mathsf{L}\end{aligned}$$

Continuing this process, we find that $\mathsf{L} = 1 + \mathsf{X} + \mathsf{X}^2 + \mathsf{X}^3 + \ldots$, which corresponds to the observation that a list is either empty, or a single element, or an ordered pair, or an ordered triple, and so on. Note that Figure 11 also defines an isomorphism directly corresponding to the implicit equation $\mathsf{L} = 1 + \mathsf{X} \bullet \mathsf{L}$.

As another example of recursive species and the power of the implicit species theorem, consider the Haskell data types shown in Figure 12.

```
data BTree a = Empty
             | Node a (BTree a) (BTree a)
  deriving Show
data Paren a = Leaf a
             | Pair (Paren a) (Paren a)
  deriving Show
```

Figure 12. Binary trees and binary parenthesizations

$BTree$ is the type of binary trees with data at internal nodes, and $Paren$ is the type of *binary parenthesizations*, that is, full binary trees with data stored in the leaves (Figure 13).

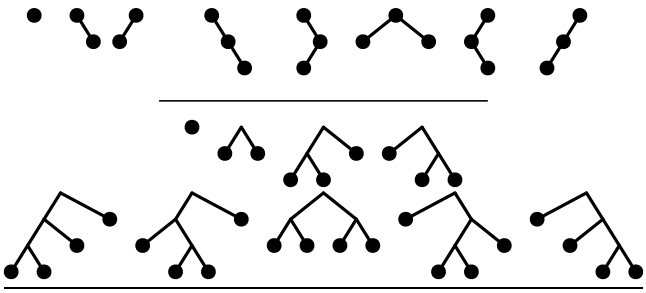


Figure 13. Binary trees (top) and parenthesizations (bottom)

It is not hard to write down equations implicitly defining species corresponding to these types:

$$\mathsf{B} = 1 + \mathsf{X} \bullet \mathsf{B}^2$$
$$\mathsf{P} = \mathsf{X} + \mathsf{P}^2$$

Figure 14 shows the isomorphisms witnessing these implicit equations. Again, the implicit equation for B itself does not fulfill the conditions of the implicit species theorem, but we can easily pull the same trick as we did with lists to turn it into one that does.

Suppose we happen to notice that there seem to always be the same number of B-structures over n labels as there are P-structures over $n + 1$ labels (say, by enumerating small instances). Is there some sort of isomorphism lurking here? In particular, can we pair an extra element with a $BTree$ to get something isomorphic to a $Paren$?

```
bTreeRec :: BTree ↔ 1 + X • BTree • BTree
bTreeRec = unroll ↔ roll
  where unroll Empty          = Inl 1
        unroll (Node x l r)   = Inr (X x • l • r)
        roll   (Inl 1)        = Empty
        roll   (Inr (X x • l • r)) = Node x l r
parenRec :: Paren ↔ X + Paren • Paren
parenRec = unroll ↔ roll
  where unroll (Leaf x)     = Inl (X x)
        unroll (Pair l r)   = Inr (l • r)
        roll   (Inl (X x))  = Leaf x
        roll   (Inr (l • r)) = Pair l r
```

Figure 14. Implicit equations for B and P

Well, pairing an extra element with a $BTree$ corresponds to the species $\mathsf{X} \bullet \mathsf{B}$. So let's just do some algebra and see what we get:

$$\begin{aligned}\mathsf{X} \bullet \mathsf{B} &= \mathsf{X} \bullet (1 + \mathsf{X} \bullet \mathsf{B}^2) \\ &= \mathsf{X} + \mathsf{X}^2 \bullet \mathsf{B}^2 \\ &= \mathsf{X} + (\mathsf{X} \bullet \mathsf{B})^2\end{aligned}$$

Thus, $\mathsf{X} \bullet \mathsf{B}$ satisfies the same implicit equation as P—so by the implicit species theorem, they must be isomorphic! Not only that, we can directly read off the isomorphism as the composition of the isomorphisms corresponding to our algebraic manipulations (Figure 15). Of course, coding this by hand is a bit of a pain, but it is not hard to imagine deriving it automatically: the *species* library cannot yet do this, but it is planned for a future release.

```
bToP :: X • BTree ↔ Paren
bToP  =  inProdR bTreeRec
      >>> prodDistrib
      >>> inSumL prodIdR
      >>> inSumR
          (    inProdR
               (    inProdL prodComm
                >>> inv prodAssoc
               )
           >>> prodAssoc
           >>> inProdL bToP
           >>> inProdR bToP
          )
      >>> inv parenRec
```

Figure 15. The isomorphism between $\mathsf{X} \bullet \mathsf{B}$ and P

Could we have derived this isomorphism without the theory of species? Of course. This particular isomorphism is not even very complicated. The point is that by boiling things down to their essentials, the theory allowed us to elegantly and easily *derive* the isomorphism with just a few lines of algebra.

3.2 Regular species, formally

What are regular species, really? We could define them as "those species which can be built from unit, singleton, sum, product, and fixed point", but there is a more direct definition that gives us much better insight into the difference between regular and non-regular species. We must first define what we mean by the *symmetries* of a structure.

Definition 2. A permutation $\sigma \in \mathcal{S}_n$ (where $\mathcal{S}_n$ denotes the symmetric group of all permutations of size n) is a *symmetry* of an F-structure $f \in \mathsf{F}[U]$ if and only if σ fixes f, that is, $\mathsf{F}_{\leftrightarrow}[\sigma](f) = f$.

For example, Figure 16 depicts a tree with a set of labels at each node. This structure has many nontrivial symmetries, such as the permutation which swaps 4 and 6 but leaves all the other labels unchanged; since 4 and 6 are in the same set, swapping them has no effect.

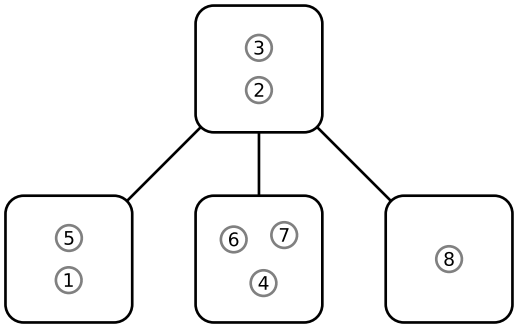


Figure 16. A labeled structure with nontrivial symmetries

However, the binary trees shown in Figure 1 have only the trivial symmetry, since permuting their labels in any nontrivial way yields different trees.

Definition 3. A species F is *regular* if every F-structure has the identity permutation as its only symmetry; such structures are also called regular.

It turns out (although we will not prove it) that this characterizes precisely those species which can be built from unit, singleton, sum, product, and least fixed point, with one important caveat: we must allow countably infinite sums and products. Of course, we cannot write down infinite sums or products in Haskell, so there are some regular types which cannot be expressed as simple algebraic data types in Haskell. For example, the regular species of prime-length lists,

$$\mathsf{X}^2 + \mathsf{X}^3 + \mathsf{X}^5 + \mathsf{X}^7 + \mathsf{X}^{11} + \ldots,$$

cannot be written as a simple algebraic data type.[1]

3.3 Other operations on regular species

Although the primitive species and species operations described in Section 3.1 suffice to describe any regular species, the class of regular species is closed under several other interesting operations as well.

Species composition Another fundamental way to build new structures is by *composing* other structures. Given species F and G, the *composition* $\mathsf{F} \circ \mathsf{G}$ is a species which builds "F-structures made out of G-structures", with the underlying labels distributed to the G-structures so that each label occurs exactly once in the overall structure (Figure 17). However, in order to ensure we get only a finite number of $(\mathsf{F} \circ \mathsf{G})$-structures of each size, G must not yield any structures on the empty label set. This corresponds exactly to the criterion for composing formal power series, namely, that the inner series have no constant term.

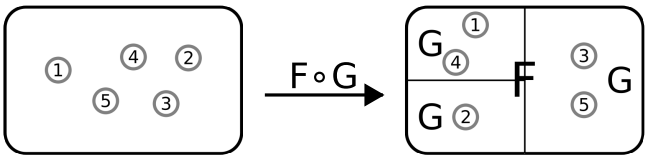
Specifically, to build an $(\mathsf{F} \circ \mathsf{G})$-structure over a label set U, we

- partition U into some number of nonempty disjoint parts, $U = U_1 \uplus U_2 \uplus \cdots \uplus U_k$;

[1] Although I am sure it can be expressed using GADTs and type-level arithmetic…

- create a G-structure on each of the U_i;
- create an F-structure on these G-structures.

Doing this in all possible ways yields the set of $(\mathsf{F} \circ \mathsf{G})$-structures over U.




```
data (f ∘ g) a = C {unC :: f (g a)}
instance (Functor f, Functor g) ⇒
         Functor (f ∘ g) where
  fmap h = C ∘ (fmap ∘ fmap) h ∘ unC
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
  ]
```

Figure 17. Species composition

For example, $\mathsf{R} = \mathsf{X} \bullet (\mathsf{L} \circ \mathsf{R})$ (where L denotes the species of linear orderings) defines the species of *rose trees* with each node having a data element and any number of children. We can also easily encode *nested data types* [6] (such types are sometimes called "non-regular", although that nomenclature is confusing in the current context, since they do in fact correspond to regular species). For example, $\mathsf{B} = \mathsf{X} + \mathsf{B} \circ \mathsf{X}^2$ is the species of *complete binary trees*; a B-structure is either a single leaf, or a complete binary tree with *pairs* of elements at the leaves.

It is not hard to verify that composition is associative (but not commutative), and that it has X as both a left and right identity. Composition also distributes over both sum and product from the right: $(\mathsf{F}+\mathsf{G})\circ\mathsf{H} = (\mathsf{F}\circ\mathsf{H})+(\mathsf{G}\circ\mathsf{H})$, and similarly for $(\mathsf{F}\bullet\mathsf{G})\circ\mathsf{H}$. These laws are shown as isomorphisms in Figure 18.

As promised at the beginning of this section, regular species are closed under composition. Although proving this formally takes a bit of work, it makes intuitive sense: if an F-structure has no symmetries, and in each hole we put a G-structure which also has no symmetries, the resulting composed structure cannot have any symmetries either.

Differentiation Of course, no discussion of an algebra for data types would be complete without a mention of differentiation. There has been a great deal of fascinating work in the functional programming community on differentiating data structures [2, 13, 16, 17]. As usual, however, the mathematicians beat us to it!

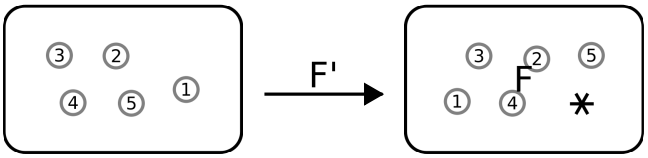
Given a species F, its *derivative* F' sends label sets U to the set of F-structures on $U \cup \{*\}$, where $*$ is a new element distinct from all elements of U. That is,

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

For example, L'-structures are lists with a distinguished hole. Since the structure on either side of the distinguished location

```
compIdL :: (X ∘ f) ↔ f
compIdL = (λ(C (X f)) → f) ↔ (C ∘ X)
compIdR :: Functor f ⇒ (f ∘ X) ↔ f
compIdR = (fmap unX ∘ unC) ↔ (C ∘ fmap X)
  where unX (X x) = x
compAssoc :: Functor f ⇒ f ∘ (g ∘ h) ↔ (f ∘ g) ∘ h
compAssoc = reAssocL ↔ reAssocR
  where reAssocL = C ∘ C ∘ fmap unC ∘ unC
        reAssocR = C ∘ fmap C ∘ unC ∘ unC
compDistribSum :: (f + g) ∘ h ↔ (f ∘ h) + (g ∘ h)
compDistribSum = distrib ↔ undistrib
  where distrib   (C (Inl fh)) = Inl (C fh)
        distrib   (C (Inr gh)) = Inr (C gh)
        undistrib (Inl (C fh)) = C (Inl fh)
        undistrib (Inr (C gh)) = C (Inr gh)
compDistribProd :: (f • g) ∘ h ↔ (f ∘ h) • (g ∘ h)
compDistribProd = distrib ↔ undistrib
  where distrib   (C (fh • gh)) = C fh • C gh
        undistrib (C fh • C gh) = C (fh • gh)
```

Figure 18. Algebraic laws for composition




```
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))
```

Figure 19. Species differentiation

is also a list, we have $\mathsf{L}' = \mathsf{L}^2$. The reader may enjoy proving this formally using the implicit species theorem and the algebraic identities for differentiation listed below.

Here is another place where it is crucial to remember the distinction between labels and data. Since every label occurs exactly once in regular structures, an F'-structure on n labels looks like an F-structure containing those labels as well as a single additional distinguished label. Once we fill in the structure by assigning a data element to each of the original n labels, there will be a single empty hole left over. So the Haskell *Diff* type is somewhat misleading, since the Haskell type $f\ (Maybe\ a)$ includes data structures containing multiple occurrences of $Nothing$, but we restrict ourselves to structures containing exactly one occurrence of $Nothing$. So, for example, even though as a Haskell type $Diff\ 1$ is perfectly well inhabited by the value 1, $Diff\ 1$ is uninhabited by values containing exactly one $Nothing$, which justifies treating it as isomorphic to 0.

Although regular species are closed under differentiation, it should be noted that there are regular species which are the derivative of a non-regular species (X, for example, is the derivative of the non-regular species E_2, to be defined in Section 6). Differentiation satisfies all the algebraic laws you would expect from calculus, and we leave expressing these isomorphisms in Haskell as a challenge for the reader:

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

Cardinality restriction For a species F and a natural number n, F_n denotes the species F restricted to label sets of cardinality n. That is,

$$\mathsf{F}_n[U] = \begin{cases} \mathsf{F}[U] & |U| = n \\ \emptyset & \text{otherwise.} \end{cases}$$

More generally, if P is any predicate on natural numbers, F_P denotes the restriction of F to label sets whose size satisfies P.

For example, L_{even} is the species of lists of even length. We have

$$\mathsf{L}_{even} = \mathsf{1} + \mathsf{X}^2 \bullet \mathsf{L}_{even} = \mathsf{1} + \mathsf{X} \bullet \mathsf{L}_{odd}$$

and

$$\mathsf{L} = \mathsf{L}_{even} + \mathsf{L}_{odd}.$$

More generally, for any species we have

$$\mathsf{F} = \mathsf{F}_{even} + \mathsf{F}_{odd} = \mathsf{F}_0 + \mathsf{F}_1 + \mathsf{F}_2 + \ldots$$

As a final note, we often write F_+ as an abbreviation for $\mathsf{F}_{\geqslant 0}$, the species of nonempty F-structures.

Of course, we cannot represent general cardinality restriction via a Haskell type, as we have with the other species operations, since we would have to somehow embed a predicate on integers into the type level.

4. An embedded language of species

The central idea, of having multiple related semantics for one underlying abstract syntax of species, can be precisely expressed by a type class. The basic *Species* type class, as defined in the *species* library, is shown in Figure 20. The actual *Species* class contains a few additional methods, but this is the core essence.

$$\begin{array}{l}
\textbf{class}\ (\mathit{Differential}.C\ s) \Rightarrow \mathit{Species}\ s\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{singleton} & :: s \\
E & :: s \\
\mathit{cycle} & :: s \\
(\circ) & :: s \to s \to s \\
(\times) & :: s \to s \to s \\
(\Box) & :: s \to s \to s \\
\mathit{ofSize} & :: s \to (\mathbb{Z} \to \mathit{Bool}) \to s
\end{array}
\end{array}$$

Figure 20. The *Species* type class

Some things may seem to be missing (0, 1, sum and product, differentiation) but these are required by the $\mathit{Differential}.C$ constraint from the *numeric-prelude* package, which ensures that species are a differentiable ring. The remainder of the class requires primitive singleton, set, and cycle species; composition ($\circ$), cartesian product ($\times$), and functor composition ($\Box$) operations; as well as a cardinality-restricting operator *ofSize*.

In essence, the *Species* type class embeds the domain-specific language of species into Haskell, allowing the user to write down species expressions which can be interpreted in different ways depending on the type at which they are instantiated.

Using this instance, it is not hard to put together the *Enumerable* instances we have already seen into code which enumerates all the labelled structures of a given species. The user can then just call the *enumerate* method on an expression of type $\mathit{Species}\ s \Rightarrow s$, along with some labels to use:

```
> enumerate (cycle `o` (nonEmpty linOrd)) "abc"
        :: [Comp Cycle [] Char]
[<"cba">,<"cab">,<"bca">,<"bac">,<"acb">,
 <"abc">,<"a","cb">,<"a","bc">,<"ca","b">,
 <"ac","b">,<"ba","c">,<"ab","c">,
 <"b","a","c">,<"a","b","c">]
```

Since the *species* library is able to automatically generate *Species* expressions representing any user-defined data type, we can also enumerate values of type *Widget Int*:

```
> enumerate widget [1,2] :: [Widget Int]
[ Children 2 [Children 1 []]
, Children 2 [Tagged True 1]
, Children 2 [Tagged False 1]
, Children 1 [Children 2 []]
, Children 1 [Tagged True 2]
, Children 1 [Tagged False 2]]
```

5. Unlabeled structures

Before moving on to non-regular species, it's worth pausing to make precise what is meant by the "shape" of a structure.

Definition 4. For a species F, an *unlabeled* F*-structure*, or F*-shape*, is an *equivalence class* of labeled F-structures, where two structures s and t are considered equivalent if there is some relabeling σ such that $\mathsf{F}_{\leftrightarrow}[\sigma](s) = t$.

In other words, two labeled structures are equivalent if they are just relabelings of each other. For example, Figure 21 shows three rose tree structures. The first two are equivalent, but the third is not equivalent to the first two, since there is no way to turn it into the first two merely by changing the labels.

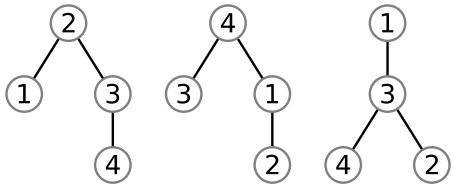


Figure 21. Equivalent and inequivalent trees

Although unlabeled structures formally consist of sets of labeled structures, we can think of them as normal structures built from "indistinguishable labels"; for a given species F, there will be one unlabeled F-structure for each possible "shape" that F-structures can take. For example, Figure 22 shows all the rose tree shapes (unlabeled rose tree structures) on four nodes.

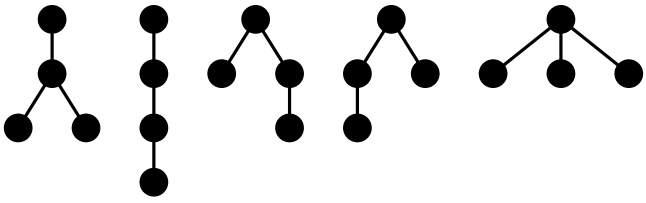


Figure 22. Unlabeled rose trees of size 4

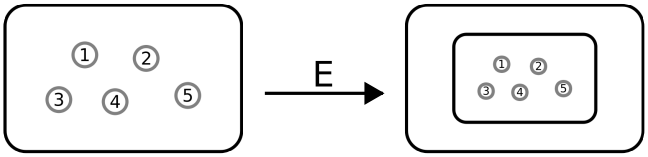
For *regular* species, the distinction between labeled and unlabeled structures is not very interesting. Since every possible permutation of the labels of a regular structure results in another distinct labeled structure, there are always exactly $n!$ times as many labeled

as unlabeled structures of size n. For regular species, a method to enumerate all *unlabeled* structures is easy. In fact, it is a slight simplification of the code we have already exhibited for enumerating labeled structures: instead of taking a list of labels as input, we take simply a natural number size, and output structures full of unit values. Enumerating unlabeled structures for *non-regular* species, however, is much more complicated. An implementation of unlabeled enumeration, along with many other algorithms over species, can be found in the *species* package. For example, we can enumerate all unlabelled sets of sets of size 4, corresponding to integer partitions of 4:

```
> enumerateU (set `o` set) 4
        :: [Comp Set Set ()]
[{{(),(),(),()}},{{(),()},{(),()}},
 {{()},{(),(),()}},{{()},{()},{(),()}},
 {{()},{()},{()},{()}}]
```

6. Beyond regular species

The species of sets The primitive species of *sets*, usually denoted E (from the French *ensemble*), represents unordered collections of elements (Figure 23). For any given set of labels, there is exactly one set structure, the set of labels itself.



$\mathbf{newtype}\ \mathit{Set}\ a = \mathit{Set}\ [a]$
  $\mathbf{deriving}\ \mathit{Functor}$

$\mathbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ (\mathit{Set}\ a)\ \mathbf{where}$
  $\mathit{Set}\ xs \equiv \mathit{Set}\ ys = \mathit{all}\ (\in ys)\ xs \wedge \mathit{all}\ (\in xs)\ ys$

$\mathbf{instance}\ \mathit{Enumerable}\ \mathit{Set}\ \mathbf{where}$
  $\mathit{enumerate} = (:[]) \circ \mathit{Set}$

Figure 23. The species E of sets

It is easy to see that E is not regular, since E-structures have every possible symmetry; permuting the labels of a set leaves the set unchanged.

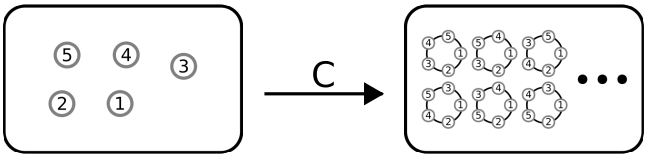
In Figure 23, we declared a Haskell data type for E by declaring Set to be isomorphic to $[\,]$ via a **newtype** declaration, and derived a $\mathit{Functor}$ instance for Set from the existing instance for lists, using GHC's *newtype deriving* extension. In Haskell, as in other languages, we can only declare regular types, so this is the best we can do. The point, of course, is that sometimes when we use lists we really *mean* for them to be sets; using Set, we can declare our semantic intent while still using lists as the underlying representation.

We can use E to build many other interesting and useful species. For example:

- $\mathsf{X} \bullet \mathsf{E}$ is the species of *pointed sets*, also known as the species of *elements* and sometimes written ε. Pointed set structures consist of a single distinguished label paired with a set structure on the rest of the labels. From this it should be clear that there are precisely n labeled ε-structures on a set of n labels, one for each choice of the distinguished label.

  We can also think of an $(\mathsf{X} \bullet \mathsf{E})$ structure as consisting *solely* of the distinguished label, since the set of remaining labels adds no information. In other words, we can treat E as a sort of "sink" for the elements we don't care about, and the same technique can be used generally for building structures out of a subset of the labels.
- $\mathsf{E} \bullet \mathsf{E}$ is the species of *subsets*, often abbreviated $\wp$ in reference to the power set operator. Again, $\wp$-structures technically consist of a subset of the labels paired with its complement, but we may (and often do) ignore the complement.
- $(\mathsf{E} \circ \mathsf{E}_+)$ is the species of *set partitions*: its structures are collections of nonempty sets.

The species of cycles The primitive species C of *directed cycles* (Figure 24) yields all directed cyclic orderings (known in the mathematical literature as *necklaces*) of the given labels. By convention, cycles are always nonempty. Also, when drawing cycles, by convention they are read clockwise.



$\mathbf{newtype}\ \mathit{Cycle}\ a = \mathit{Cycle}\ [a]$
  $\mathbf{deriving}\ \mathit{Functor}$

$\mathbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ (\mathit{Cycle}\ a)\ \mathbf{where}$
  $\mathit{Cycle}\ xs \equiv \mathit{Cycle}\ ys = \mathit{any}\ (\equiv ys)\ (\mathit{rotations}\ xs)$
    $\mathbf{where}\ \mathit{rotations}\ xs = \mathit{zipWith}\ (+\!\!+)\ (\mathit{tails}\ xs)\ (\mathit{inits}\ xs)$

$\mathbf{instance}\ \mathit{Enumerable}\ \mathit{Cycle}\ \mathbf{where}$
  $\mathit{enumerate}\ [\,] = [\,]$
  $\mathit{enumerate}\ (x : xs) = (\mathit{map}\ (\mathit{Cycle} \circ (x:)) \circ \mathit{permutations})\ xs$

Figure 24. The species C of cycles

C is also non-regular, since each labeled C-structure has certain nontrivial permutations that leave it fixed, namely, the ones which only "rotate" the labels.

An example of an interesting species we can build using C is $\mathsf{E} \circ \mathsf{C}$, the species of *permutations*, corresponding to the observation that every permutation can be decomposed into a collection of disjoint cycles.

We note also that a cycle with a hole in it is isomorphic to a list, that is, $\mathsf{C}' = \mathsf{L}$ (Figure 25).

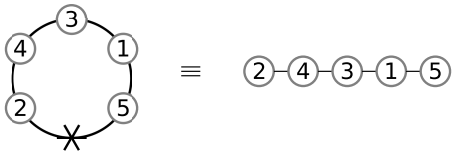
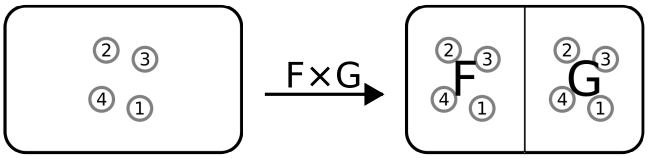


Figure 25. Differentiating a cycle

Cartesian product Given two species F and G, we may define the *Cartesian product* $\mathsf{F} \times \mathsf{G}$ by

$$(\mathsf{F} \times \mathsf{G})[U] = \mathsf{F}[U] \times \mathsf{G}[U],$$

where the $\times$ on the right denotes the standard Cartesian product of sets. That is, an $(\mathsf{F} \times \mathsf{G})$-structure is a pair of an F-structure and a G-structure, both of which are built over *all* the labels, instead of partitioning the labels as with normal product (Figure 26). However, instead of thinking of the labels as being duplicated, we think of an $(\mathsf{F} \times \mathsf{G})$-structure as two structures which are *superimposed* on the same label set. In particular, when specifying the content for an $(\mathsf{F} \times \mathsf{G})$-structure, we should still only map each label to a single piece of data.




```
data (f × g) a = f a × g a
instance (Functor f, Functor g) ⇒
         Functor (f × g) where
  fmap f (x × y) = fmap f x × fmap f y
instance (Enumerable f, Enumerable g) ⇒
         Enumerable (f × g) where
  enumerate ls = [x × y | x ← enumerate ls
                        , y ← enumerate ls]
```

Figure 26. Cartesian product of species

One interesting use of Cartesian product is to model some typeclass-polymorphic data structures, where the type class methods provide us with a second observable structure on the data elements. For example, a type constructor F with an *Eq* constraint on its argument can be modeled by the species

$$\mathsf{F} \times (\mathsf{E} \circ \mathsf{E}_+).$$

Structures of this species consist of an F-structure with a superimposed partition on the labels, corresponding to the observable equivalence classes. For example, Figure 27 shows a binary tree shape with a superimposed partition indicating which sets of elements are observably equal.

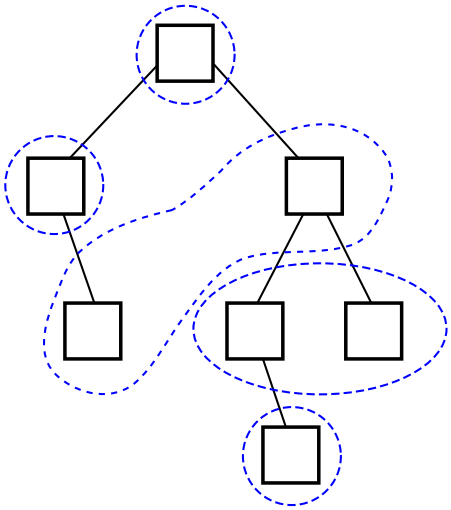


Figure 27. A $(\mathsf{B} \times (\mathsf{E} \circ \mathsf{E}_+))$-shape

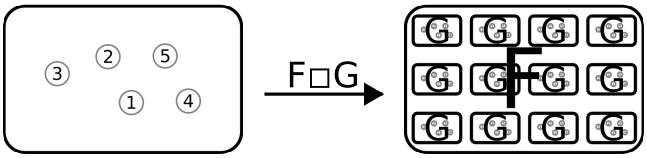
Likewise, we can model an *Ord* constraint by superimposing an $(\mathsf{L} \circ \mathsf{E}_+)$-structure, which additionally places an observable ordering on the equivalence classes.

Cartesian product has E as both a left and right identity, and is associative, commutative, and distributes over species sum. Again, implementing these laws as isomorphisms is left as an exercise for the reader.

Functor composition The final species operation we will explore is *functor composition*. Given species F and G, we define their functorial composite by

$$(\mathsf{F} \,\square\, \mathsf{G})[U] = \mathsf{F}[\mathsf{G}[U]],$$

that is, F-structures over *the set of all* G*-structures* on U (Figure 28). An $(\mathsf{F} \,\square\, \mathsf{G})$-structure is an F-structure of G-structures, just like $(\mathsf{F} \circ \mathsf{G})$-structures, but instead of partitioning the labels U among several G-structures, we give all the labels to every G-structure. So, just as with Cartesian product structures, $(\mathsf{F} \,\square\, \mathsf{G})$-structures appear to contain each label multiple times, but in fact we should still think of them as containing each label once, with a rich structure superimposed on it.




```
data (f □ g) a = FC {unFC :: f (g a)}
instance (Functor f, Functor g) ⇒
         Functor (f □ g) where
  fmap h = FC ∘ (fmap ∘ fmap) h ∘ unFC
instance (Enumerable f, Enumerable g) ⇒
         Enumerable (f □ g) where
  enumerate = map FC ∘ enumerate ∘ enumerate
```

Figure 28. Functor composition

The functor composition operation is especially useful for defining species of graphs and relations. For example, recalling that $\wp = \mathsf{E} \bullet \mathsf{E}$ is the species of subsets and E_2 is the species of sets restricted to sets of size two,

$$\wp \,\square\, (\mathsf{E}_2 \bullet \mathsf{E})$$

defines the species of simple graphs. An $(\mathsf{E}_2 \bullet \mathsf{E})$-structure is a set of two labels, which we can think of as an undirected edge, and a simple graph is a subset of the set of all possible edges.

In fact, many graph-like species can be defined as $\wp \,\square\, \mathsf{G}$ for a suitable species G. For example, $\mathsf{G} = \mathsf{X}^2 \bullet \mathsf{E}$ gives directed graphs without self-loops, and $\mathsf{G} = \varepsilon \times \varepsilon$ gives directed graphs with self-loops allowed (recalling that $\varepsilon = \mathsf{X} \bullet \mathsf{E}$ is the species of elements). The reader may enjoy discovering how to represent the species of undirected graphs with self-loops allowed.

7. Generating functions and counting

What else can we do with combinatorial species? A core part of the story we haven't mentioned yet is the correspondence between each species and several different *generating functions*, which encode various aggregate information about the species.

For example, we can associate to each combinatorial species F an *exponential generating function* (egf) of the form

$$\mathsf{F}(x) = \sum_{n \geqslant 0} f_n \frac{x^n}{n!},$$

where f_n is the number of labeled F-structures that can be built from a label set of size n. (Note that x is a purely formal parameter, and we need not concern ourselves with convergence; for a more detailed explanation of generating functions, see Wilf's *generatingfunctionology* [22].) Thus we have, for example,

- $\mathsf{0}(x) = \frac{0}{0!}x^0 + \frac{0}{1!}x^1 + \cdots = 0$,
- $\mathsf{1}(x) = 1$,
- $\mathsf{X}(x) = x$,
- $\mathsf{L}(x) = \displaystyle\sum_{n \geqslant 0} \frac{n!}{n!} x^n = \frac{1}{1-x}$,

- $\mathsf{E}(x) = \sum_{n \geqslant 0} \frac{1}{n!} x^n = e^x$,
- $\mathsf{C}(x) = \sum_{n \geqslant 1} \frac{(n-1)!}{n!} x^n = -\log(1-x)$.

(Here is another very good reason to call the species of sets E!) At first glance this may seem arbitrary; but the astounding fact is that species sum, product, composition, and differentiation correspond precisely to the same operations on formal power series! For example, if $\mathsf{F}(x)$ and $\mathsf{G}(x)$ count the number of labeled F- and G-structures as defined above, it is easy to see that $\mathsf{F}(x)+\mathsf{G}(x)$ counts the number of labeled $(\mathsf{F}+\mathsf{G})$-structures in the same way, since every $(\mathsf{F}+\mathsf{G})$-structure is either an F-structure or a G-structure (with a tag). And although it is not as immediately apparent, we can verify that $\mathsf{F}(x)\mathsf{G}(x) = (\mathsf{F} \bullet \mathsf{G})(x)$ as well:

$$\begin{aligned}
\mathsf{F}(x)\mathsf{G}(x) &= \left(\sum_{n \geqslant 0} f_n \frac{x^n}{n!}\right)\left(\sum_{n \geqslant 0} g_n \frac{x^n}{n!}\right) \\
&= \sum_{n \geqslant 0} \sum_{k=0}^{n} \left(f_k \frac{x^k}{k!}\right)\left(g_{n-k} \frac{x^{n-k}}{(n-k)!}\right) \\
&= \sum_{n \geqslant 0} \sum_{k=0}^{n} f_k g_{n-k} \frac{x^n}{k!(n-k)!} \\
&= \sum_{n \geqslant 0} \left(\sum_{k=0}^{n} \binom{n}{k} f_k g_{n-k}\right) \frac{x^n}{n!}
\end{aligned}$$

The expression in the outermost parentheses is exactly the number of $(\mathsf{F} \bullet \mathsf{G})$-structures on a label set of size n: for each k from 0 to n, there are $\binom{n}{k}$ ways to pick k of the n labels to put in the F-structure, f_k ways to create an F-structure from them, and g_{n-k} ways to create a G-structure from the remaining labels.

The reader may also enjoy working out why species differentiation corresponds to exponential generating function differentiation. Seeing the correspondence between species composition and egf substitution takes more work, but also works out in the end: the interested reader should look up *Faà di Bruno's formula*. There are also generating function operations corresponding to Cartesian product and functor composition; although they are not as natural as the other operations, they are simple to define, and the reader may enjoy working out what they are.

The upshot is that we can count labeled structures by interpreting species expressions as exponential generating functions. In fact, this has been known in the functional programming community for some time [18, 20]. The *species* library defines an EGF type with an appropriate $Species$ instance, which allows us to compute as follows:

```
> take 10 . labelled $ 3 + x*x
[3,0,2,0,0,0,0,0,0,0]
> take 10 . labelled
      $ cycle `o` (nonEmpty linOrd)
[0,1,3,14,90,744,7560,91440,1285200,
 20603520]
```

The library can even compute generating functions for some recursively defined species, using a combinatorial analogue of the Newton-Raphson method. For example, once Jane has used the *species* library to derive all the appropriate instances for her $Widget$ type via Template Haskell, she can use the *species* library to count them:

```
> take 10 . labelled $ widget
[0,3,6,72,1368,36000,1213920,49956480,2427788160
,136075645440]
```

To each species we can also associate an *ordinary generating function* (ogf) and a *cycle index series*; the first counts unlabeled structures (shapes), and the second is a generalization of both exponential and ordinary generating functions which also keeps track of symmetries. There is not space to describe them here, but more information can be found in Bergeron *et at.* [4] or in the documentation for the *species* library, which includes facilities for computing with all three types of generating functions.

8. Extensions and applications

It should come as no surprise that we have barely scratched the surface; the theory of combinatorial species is rich and deep. In closing, we mention just a few extensions to the theory discussed here which hold promise for shedding additional light on functional programming and algebraic data types. We also mention a few potential applications of the theory.

8.1 Extensions

Weighted species Assigning *weights* to the structures built by a species allows us to count and enumerate structures in much more refined ways. For example, we could define the species of binary trees, weighted by the number of leaves, and then easily count or enumerate just those trees with a certain number of leaves.

Multisort species We have only considered species which map a single set of labels to a set of structures, corresponding to type constructors of a single argument. However, all of the theory generalizes quite straightforwardly to *multisort* species, which build structures from multiple sets of labels (*sorts*). For example, we can use multisort species to define the species of simple graphs with edge weights, or a species corresponding to any type constructor of multiple arguments. Extending the *species* library to handle general multisort species presents an interesting challenge, due to Haskell's lack of kind polymorphism, and is a topic for further research.

Virtual species It is possible to complete the semiring of species to a ring by the addition of so-called *virtual* species, allowing us to make sense of subtraction of species. The direct value of virtual species to functional programming is not clear, but one practical implication of their existence is that when manipulating species equations, we may freely use intermediate steps which introduce subtraction, without worrying about issues of definedness. It is also worth exploring whether they have any direct applications.

8.2 Applications

Automated testing One interesting application is to use species expressions as input to a test-generator-generator, for either random [9] or exhaustive [21] testing. In fact, Canou and Darrasse have already created such a library for random test generation in OCaml [7]. There has also been some interesting recent work by Duregård on automatic derivation of QuickCheck generators for algebraic data types [10], and by Bernardy *et al.* on using parametricity to improve random test generation [5]; combining these approaches with insights from the theory of species seems promising.

Language design What if we had a programming language that actually allowed us to declare non-regular data types? What would such a language look like? Could it be made practical? These questions have been explored by Carette and Uszkay [8] and Abbott *et al.* [3], but continued research along these lines seems interesting.

References

[1] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In *Foundations of Software Science and Computation Structures*, pages 23–38. 2003.

[2] Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of containers. In *Typed Lambda Calculi and Applications, TLCA*, volume 2701 of *LNCS*. Springer-Verlag, 2003.

[3] Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Constructing polymorphic programs with quotient types. In *7th International Conference on Mathematics of Program Construction (MPC 2004)*, volume 3125 of *LNCS*. Springer-Verlag, 2004.

[4] F. Bergeron, G. Labelle, and P. Leroux. *Combinatorial species and tree-like structures*. Number 67 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, Cambridge, 1998.

[5] Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. Testing polymorphic properties. In *ESOP 2010: Proceedings of the 19th European Symposium on Programming*, pages 125–144, London, UK, 2010. Springer-Verlag.

[6] Bird and Meertens. Nested datatypes. In *MPC: 4th International Conference on Mathematics of Program Construction*. LNCS, Springer-Verlag, 1998.

[7] Benjamin Canou and Alexis Darrasse. Fast and sound random generation for automated testing and benchmarking in objective caml. In *ML '09: Proceedings of the 2009 ACM SIGPLAN workshop on ML*, pages 61–70, New York, NY, USA, 2009. ACM.

[8] Jacques Carette and Gordon Uszkay. Species: making analytic functors practical for functional programming. *Submitted to MSFP 2008*.

[9] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM.

[10] Jonas Almström Duregård. AGATA: Random generation of test data. Master's thesis, Chalmers University of Technology, December 2009.

[11] P. Flajolet, B. Salvy, and P. Zimmermann. Lambda-upsilon-omega: The 1989 cookbook. Technical Report 1073, Institut National de Recherche en Informatique et en Automatique, August 1989. 116 pages.

[12] Philippe Flajolet and Bruno Salvy. Computer algebra libraries for combinatorial structures. *Journal of Symbolic Computation*, 20(5-6):653–671, 1995.

[13] Gérard Huet and Inria Rocquencourt France. Functional pearl: The zipper. *J. Functional Programming*, 7:7–5, 1997.

[14] C. Barry Jay and J. Robin B. Cockett. Shapely types and shape polymorphism. In *ESOP '94: Proceedings of the 5th European Symposium on Programming*, pages 302–316, London, UK, 1994. Springer-Verlag.

[15] André Joyal. Une théorie combinatoire des Séries formelles. *Advances in Mathematics*, 42(1):1–82, 1981.

[16] Conor McBride. The Derivative of a Regular Type is its Type of One-Hole Contexts. Available at `http://www.cs.nott.ac.uk/~ctm/diff.ps.gz`, 2001.

[17] Conor McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 287–295, San Francisco, California, USA, 2008. ACM.

[18] M. Douglas McIlroy. Power series, power serious. *Journal of Functional Programming*, 9(03):325–337, 1999.

[19] Peter Morris, Thorsten Altenkirch, and Conor Mcbride. Exploring the regular tree types. 2004.

[20] Dan Piponi. A small combinatorial library, November 2007. `http://blog.sigfpe.com/2007/11/small-combinatorial-library.html`.

[21] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 37–48, New York, NY, USA, 2008. ACM.

[22] Herbert S. Wilf. *Generatingfunctionology*. Academic Press, 1990.