9-2010

Species and Functors and Types, Oh My!

Brent A. Yorgey  
*University of Pennsylvania*, byorgey@cis.upenn.edu

Follow this and additional works at: [http://repository.upenn.edu/cis_papers](http://repository.upenn.edu/cis_papers)

Part of the [Discrete Mathematics and Combinatorics Commons](http://repository.upenn.edu/cis_papers) and the [Programming Languages and Compilers Commons](http://repository.upenn.edu/cis_papers)

**Recommended Citation**

Brent A. Yorgey, "Species and Functors and Types, Oh My!", . September 2010.

This paper is posted at ScholarlyCommons. [http://repository.upenn.edu/cis_papers/761](http://repository.upenn.edu/cis_papers/761)

For more information, please contact libraryrepository@pobox.upenn.edu.
Species and Functors and Types, Oh My!

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 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 basic theory of combinatorial species using motivation and examples from the world of functional programming. It also introduces the species library, available on Hack-age, which is used to illustrate the concepts introduced and can serve as a platform for continued study and research.

Keywords
combinatorial species, algebraic data types

Disciplines
Discrete Mathematics and Combinatorics | Programming Languages and Compilers

This conference paper is available at ScholarlyCommons: http://repository.upenn.edu/cis_papers/761
Species and Functors and Types, Oh My!

Brent A. 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 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 basic 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.

Categories and Subject Descriptors D.3.3 [Language Constructs and Features]: Data types and structures; D.1.1 [Programming Techniques]: Applicative (Functional) Programming; G.2.1 [Combinatorics]

General Terms Languages, Theory

Keywords Combinatorial species, algebraic data types

1. Introduction
The theory of combinatorial species was invented by André Joyal in 1981 [16] as an elegant framework for understanding and unifying much of enumerative combinatorics. Since then, mathematicians have continued to develop the theory, proving a wide range of fundamental results and producing at least one excellent reference text on the topic [4]. Connections to computer science and functional programming have been pointed out in detail, notably by Flajolet, Salvy, and Zimmermann [12, 13]. Sadly, however, this beautiful theory is not widely known among functional programmers.

Suppose Dorothy G. Programmer has created the following data type to aid in her ethological study of alate simian family groups:

```haskell
data Family a = Monkey Bool a | Group a [Family a]
```

That is, a family (parameterized by names of type `a`) is either a single monkey with a boolean indicating whether it can fly, or an alpha male together with a group of families.

While developing and testing her software, Dorothy might want to do things such as enumerate or count all the family structures of a certain size, or randomly generate family structures. There exist tools for accomplishing at least two of these tasks: QuickCheck [9] and SmallCheck [22] can be used to do random and exhaustive generation, respectively.

However, suppose Dorothy now decides that the order of the families in a group doesn’t matter, although she wants to continue using the same list representation. Suddenly she is out of luck: Haskell has no way to formally describe this rather common situation, and there is no easy way to inform QuickCheck and SmallCheck of her intentions. She could add a `Bag` newtype,

```haskell
newtype Bag a = Bag [a]
```

and endow it with custom QuickCheck and SmallCheck generators, but this is rather ad-hoc. What if she later decides that the order of the families does matter after all, but only up to cyclic rotations? Or that groups must always contain at least two families? Or what if she wants to have a data structure representing the graph of interactions between different family groups?

What Dorothy needs is a coherent framework in which to describe these sorts of sophisticated structures. The theory of species is precisely such a framework: for example, her original data structure can be described succinctly by the regular species expression

\[ F = \mathcal{X} + \mathcal{X} \cdot (\mathcal{L} \circ F) \]

Section 3 explains how to interpret this expression. The variants on her structure correspond to non-regular species (Section 5) and can be expressed with only simple tweaks to the original expression. The payoff is that these species expressions form an abstract syntax (Section 6) with multiple useful semantic interpretations, including ways to exhaustively enumerate, count, or randomly generate structures (Sections 7 and 8).

This paper is available at http://www.cis.upenn.edu/~byorgey/papers/species-pearl.lhs as a literate Haskell document. The species library itself, together with a good deal of documentation and examples, is available on Hackage [10] at http://hackage.haskell.org/package/species.

2. Combinatorial species
Intuitively, a species describes a family of structures, parameterized by a set of labels which identify locations in the structures. 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 set of mappings, \( F_\bullet \) and \( F_\ast \), with the following properties:

- \( F_\bullet \), given a finite set \( U \) of labels, sends \( U \) to a finite set of structures \( F_\bullet[U] \) which can be “built from” the given labels. We call \( F_\bullet[U] \) the set of \( F \)-structures with support \( U \), or sometimes just \( F \)-structures over \( U \).
- \( F_\ast \), given a bijection \( \sigma : U_1 \leftrightarrow U_2 \) between two label sets \( U_1 \) and \( U_2 \) (i.e. a relabeling), “lifts” \( \sigma \) to a bijection between \( F \)-structures,

\[
F_\ast[\sigma] : F_\bullet[U_1] \leftrightarrow F_\bullet[U_2].
\]
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 $F_*$ when it is clear from context.

For example, Figure 1 illustrates lifting a relabeling $\sigma$ to a relabeling of binary trees.

```
\sigma : 1 2 3 4 5
    \quad a c e b d
```

![Figure 1. Relabeling a binary tree](image)

Note that the notion of structures in this definition is entirely abstract: a “structure” is just an element of the set output by $F_*$, which could be anything (subject to the restrictions on $F$).

Fans of category theory will recognize a much more concise version of this definition: a species is an endofunctor on $B$, 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 determined by its action on any particular such set: if $|U_1| = |U_2|$ and we know $F(U_1)$, we can determine $F(U_2)$ 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.

As a final note on the definition, it is tempting to assume that labels play the role of the “data” held by data structures. Instead, however, labels should be thought of as names for the locations within a structure. The idea is that data structures can be decomposed into a shape together with some sort of content [1, 15]. In this case, a labeled shape is some sort of structure built out of labels, and the content can be specified by a mapping from labels to data (which need not be injective).

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 small set of primitive species and species operations. We start our tour of the species menagerie with what I term regular species.\footnote{There is no widely accepted name for this class of species; I call them regular since they correspond to the regular tree types of Morris et al. [20].} These should seem like old friends: intuitively, regular species correspond to Haskell’s algebraic data types. 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 $F_*$ mapping—that is, values of the type will correspond to $F$-structures. The $F_*$ mapping, in turn, can be 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. (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 not fundamentally so.

```
class Enumerable f where
    enumerate :: [a] \to [f a]
```

![Figure 2. The Enumerable type class](image)

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 an arbitrary 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. 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](image)

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 locations where values of type a can be stored. It
corresponds to nullary constructors in algebraic data types. 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
instance Functor 1 where
  fmap _ _ = 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 location.

```
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 _ = []
```

*Figure 5.* The species $X$ of singletons

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 $(F + G)$-structures over $U$ is the disjoint union of the sets of $F$- and $G$-structures over $U$:

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

In other words, an $(F + G)$-structure is either an $F$-structure or a $G$-structure, along with a tag specifying which. For example, $1 + X$ corresponds to the familiar `Maybe` type constructor. We can generalize 0 and 1 by defining the species $n$, for each $n \geq 0$, 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 $1 + \cdots + 1$. For example, 2 corresponds to the

```
data CBool a = CBool Bool.
```

It is not hard to verify that, up to isomorphism, 0 is the identity for species addition, and that $+$ is associative and commutative. Since these algebraic laws correspond directly to generic isomorphisms between structures, we can 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.

```
instance ( Enumerable f , Enumerable g ) \Rightarrow
  Enumerable ( f + g ) where
  enumerate bs = map Inl ( enumerate bs )
  \uplus map Inr ( enumerate bs )
```

*Figure 6.* Species sum

```
  (+) :: [ f a \rightarrow g a ] \rightarrow ( h a \rightarrow f a )
  \rightarrow ( f + h ) a \rightarrow ( g + j ) a
  (fg + gh) ( Inl fa ) = Inl ( fg fa )
  (fg + gh) ( Inr ha ) = Inr ( hj ha )
```

*Figure 7.* Isomorphisms

Armed with these definitions, Figure 8 presents the algebraic laws for sum in Haskell form, as implemented in the species library. The one technical issue to note is that for the congruences in `SumL` and `SumR` (and the corresponding `ProdL` and `ProdR` shown in the next section), we must be careful to use lazy pattern matches, since the isomorphism between $f$ and $g$ may not be needed. Always forcing the proof of $(f \leftrightarrow g)$ to weakhead normal form can cause some isomorphisms between recursive structures (such as the one shown in Figure 15) to diverge.

Species product Just as species sum corresponds to type sum, we define species product (Figure 9) to correspond to type product, in
sumIdL :: 0 + f ↦ f
sumIdL = (λ(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 (Inr x)
reAssocL (Inr (Inl x)) = Inr x
reAssocR (Inl (Inr x)) = Inr x
reAssocR (Inr (Inl x)) = Inl x

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

Figure 8. Algebraic laws for sum

such a way that the resulting structures contain each label exactly once. So, to form an (F • G)-structure over U, we split U into two disjoint subsets, and form an 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 (F • G). Formally,

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

where \sum denotes repeated disjoint union and \times denotes Cartesian product of sets.

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 bs = [x • y | (fbs, gbs) ← splits bs, x ← enumerate fbs, y ← enumerate gbs]
splits :: [a] → ([a], [a])
splits [] = ([], [])
splits (x : xs) = (map \circ first) (x : ss) + (map \circ second) (x : ss)

where ss = splits xs
first f (x, y) = (f x, y)
second f (x, y) = (x, f y)

For example, X • 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 X • 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 • 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, although their straightforward implementations are omitted in the interest of space.

prodIdL :: 1 • f ↦ f
prodIdR :: f • 1 ↦ f
prodAbsorbL :: 0 • f ↦ 0
prodComm :: f • g ↦ g • f
prodAssoc :: f • (g • h) ↦ (f • g) • h
prodDistrib :: f • (g + h) ↦ (f • g) + (f • h)
inProdL :: (f ↦ g) → (f • h ↦ g • h)
inProdR :: (f ↦ g) → (h • f ↦ h • g)

Figure 10. Algebraic laws for product

Least 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 [20]. For example, the species L of linear orderings (or lists for short) can be defined as

L = µf.1 + X • L.

That is, a list is either empty (1) or an element paired with a list (X • L). For any set U of labels, [L][U] is the set of all linear orderings of U (Figure 11); of course, |L[n]| = n!.

instance Enumerable [w] 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

Actually, mathematicians would not write L = µf.1 + X • L, but simply

L = 1 + X • L.

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 a species equation which implicitly defines F in terms of itself. If

\[ F = \sum_{n=0}^{\infty} \frac{1}{n!} [L][n] \]

\[ F(x) = \sum_{n=0}^{\infty} \frac{1}{n!} [L][n] \]
F yields no structures on the empty label set, and is not trivially reducible to itself, then the implicit species theorem guarantees that there is a unique solution for F with F(0) = 0, which is exactly the least fixed point of the implicit equation. Of course, the criteria given above are somewhat vague; a more precise formulation is explained in Section 8.1.

The species $L$, as defined above, does not actually meet these criteria, since it yields a structure on the empty label set. However, if we let $L_+$ denote the species of nonempty lists, with $L_+ + 1 = L$, then we have $L_+ = X \cdot (L_+ + 1)$, which does meet the criteria and hence has a unique solution. Thus, $L = 1 + L_+$ is uniquely determined as well, and we are justified in forgetting about $\mu$ simply manipulating the implicit equation

We can also “solve” the implicit equation for $L$ which corresponds to the observation that a list is either empty, or has at least one element. Of course, the criteria corresponding to this list is $L = 1 + X \cdot L$ however we like. For example, expanding $L$ in its own definition, we find that

$$L = 1 + X \cdot L$$

$$= 1 + X \cdot (1 + X \cdot L)$$

$$= 1 + X + X^2 \cdot L$$

Continuing this process, we find that $L = 1 + X + X^2 + X^3 + \ldots$, 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. We can also “solve” the implicit equation for $L$ to obtain $L = \frac{1}{2 - X}$.

This may seem nonsensical at this point—we have defined neither subtraction nor division of species—but it is perfectly valid in the context of virtual species (Section 8.1), and directly corresponds to the exponential generating function for $L$ (Section 7).

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

```
data BTTree a = Empty  
               | Node a (BTTree a) (BTTree a)
data Paren a = Leaf a  
             | Pair (Paren a) (Paren a)
```

**Figure 12.** Binary trees and binary parenthesizations

$BTTree$ 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:

$$B = 1 + X \cdot B^2$$

$$P = X + P^2$$

Figure 14 shows the isomorphisms witnessing these implicit equations. Again, $B$ itself does not fulfill the conditions of the implicit species theorem, but we can easily express it in terms of 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 $BTTree$ to get something isomorphic to a $Paren$?

Well, pairing an extra element with a $BTTree$ corresponds to the species $X \cdot B$. Let’s just do some algebra and see what we get:

$$X \cdot B = X \cdot (1 + X \cdot B^2)$$

$$= X + X^2 \cdot B^2$$

$$= X + (X \cdot B)^2$$

Thus, $X \cdot B$ 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 \cdot BTTree <> Paren  
bToP = inProdR bTreeRec  
       prodDistrib  
       inSumL prodIdR  
       inSumR (  
               inProdR (  
                   inProdL prodComm  
                   inv prodAssoc)  
               prodAssoc  
               inProdL bToP  
               inProdR bToP)  
       inv parenRec
```

**Figure 14.** Implicit equations for $B$ and $P$

```
```

**Figure 15.** The isomorphism between $X \cdot B$ and $P$

Could we have come up with this isomorphism without the theory of species? Of course. This particular isomorphism is not even that complex. The point is that by boiling things down to their essentials, the theory allowed us to elegantly and easily derive the isomorphism with only a few lines of algebra.
3.2 Regular species, formally

We are now ready to state the precise definition of regular species. A first characterization is as follows:

Definition 2. A species \( F \) is regular if it can be expressed in terms of 1, \( X, +, \cdot \), and least fixed point.

This definition validates the promised intuition that regular species correspond to Haskell algebraic data types, since normal Haskell98 data type declarations provide exactly these features (forgetting for the moment about infinite data structures).

However, there is a more direct characterization that makes apparent why this particular collection of species is interesting. We must first define what we mean by the symmetries of a structure. Recall that \( S_n \) denotes the termsymmetric group of order \( n \), which has permutations of size \( n \) (that is, bijections between \( \{1, \ldots, n\} \) and itself) as elements, and composition of permutations as the group operation.

Definition 3. A permutation \( \sigma \in S_n \) is a symmetry of an \( F \)-structure \( f \in F[U] \) if and only if \( \sigma \) fixes \( f \), that is, \( F_{\sigma} \cdot \sigma(f) = f \).

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.

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 4. 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 that these two definitions are equivalent (with the slight caveat that we must allow countably infinite sums and products in the first definition). That species built from sum, product, and fixed point have no symmetries is not hard to see; less obvious is the fact that up to isomorphism, every species with no symmetries can be expressed in this way (a proof sketch is given in Section 8.1). Of course, since we cannot write down infinite sums or products in Haskell, there are some regular species which cannot be expressed as simple algebraic data types. For example, the regular species of prime-length lists,

\[
X^2 + X^3 + X^5 + X^7 + X^{11} + \ldots
\]

cannot be written as a simple algebraic data type.\(^2\) But aside from infinite sums and products, as long as we stick to data structures

\(^2\)Although I am sure it can be expressed using GADTs and type-level arithmetic...
As noted at the beginning of this section, regular species are closed under composition. Although we won’t prove this formally, it makes intuitive sense: if an $F$-structure has no symmetries, and in each location 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 mentioning differentiation. There has been a great deal of fascinating work in the functional programming community on differentiating data structures [2, 14, 17, 18]. As usual, however, the mathematicians beat us to it!

Intuitively, the derivative of a data type $D$ is the type of $D$-structures with a single “hole”; that is, a distinguished location not containing any data. This is useful, for example, in building zipper structures [14] which keep track of a movable focus within a data structure. We can make this precise in the context of species by using a “dummy label” to correspond to the hole.

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

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

![Figure 18. Species differentiation](image)

**newtype** `Diff f a = Diff (f (Maybe a))`

**deriving** `Functor`

**instance** `Enumerable f ⇒ Enumerable (Diff f)` where

```haskell```
    enumerate ls = map Diff (enumerate (Nothing : map Just ls))
```

4. **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 5.** 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 $\sigma$ such that $F_\sigma(s) = t$.

In other words, two labeled structures are equivalent if they are relabelings of each other. For example, Figure 19 shows three rose tree structures. The first two are equivalent, but the third is not equivalent to the first two, since there is obviously no way to turn it into the first two merely by changing the labels.

![Figure 19. Equivalent and inequivalent trees](image)

Although unlabeled structures formally consist of equivalence classes of labeled structures, we can informally 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 20 shows all the rose tree shapes on four nodes.

For regular species, the distinction between labeled and unlabeled structures is uninformative. Since every possible permutation of the labels of a regular structure results in a distinct labeled structure, there are always exactly $n!$ times as many labeled as unlabeled structures of size $n$. Thus, a method to enumerate all unlabeled structures of a regular species 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; a partial implementation can be found in the species package. For example, we can enumerate all unlabeled sets of size 4, corresponding to integer partitions of 4:

\[
\text{enumerateU (set \text{'o'} \text{nonEmpty set}) 4} = [\{\}, \{\text{'o'}\}, \{\text{'o'}, \text{'o'}\}, \{\text{'o'}, \text{'o'}, \text{'o'}\}]
\]

As promised, the theory of combinatorial species can describe structures with nontrivial symmetries just as easily as regular structures. This section introduces common non-regular species and combinatorics.

The species of sets The primitive species of sets, usually denoted \(E\) (from the French ensemble), represents unordered collections of elements. For any given set of labels, there is exactly one set structure, the set of labels itself. It is easy to see that \(E\) is not regular, since \(E\)-structures have every possible symmetry; permuting the elements of a set leaves the set unchanged.

Although the standard mathematical name for this species is the species of sets, a better name for it from a computer science perspective is the species of bags. The term set usually indicates both that the order of the elements doesn’t matter and that there are no duplicate elements; the species of bags only embodies the former, since we can have non-injective mappings from labels to data. However, to model sets we can certainly restrict ourselves to injective mappings.

\[
\text{newtype Bag a = Bag [a]} \\
\text{deriving Functor}
\]

\[
\text{instance Eq a \Rightarrow Eq (Bag a) where}
\text{Bag xs \equiv Bag ys \Rightarrow xs \equiv ys 'subBag' ys}
\text{where subBag b = null \circ foldl' (flip delete) b}
\]

\[
\text{instance Enumerable Bag where}
\text{enumerate [] = []}
\text{enumerate (x : xs) = (map (Cycle \circ (x:)) \circ permutations) xs}
\]

The species of cycles The primitive species \(C\) of directed cycles (Figure 22) yields all directed cyclic orderings (known in the mathematical literature as necklaces) of the given labels. By convention, cycles are always nonempty, and are read clockwise when represented pictorially.

\[
\text{newtype Cycle a = Cycle [a]} \\
\text{deriving Functor}
\]

\[
\text{instance Eq a \Rightarrow Eq (Cycle a) where}
\text{Cycle xs \equiv Cycle ys \Rightarrow xs \equiv ys 'rotations' xs}
\text{where rotations xs = zipWith (+) (tails xs) (inits xs)}
\]

\[
\text{instance Enumerable Cycle where}
\text{enumerate [] = []}
\text{enumerate (x : xs) = (map (Cycle \circ (x:)) \circ permutations) xs}
\]

\[
\text{Figure 20. Unlabeled rose trees of size 4}
\]

\[
\text{Figure 21. The species E of sets}
\]

\[
\text{Figure 22. The species C of cycles}
\]

\(C\) is also non-regular, since each labeled \(C\)-structure is fixed by certain nontrivial permutations, namely, the ones which only "rotate" the labels.

An example of an interesting species we can build using \(C\) is \(E \circ C\); 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, \(C' = L\) (Figure 23).

\[
\text{Cartesian product} \quad \text{Given two species } F \text{ and } G, \text{ we may define the Cartesian product } F \times G \text{ by}
\text{(F \times G)[U] = F[U] \times G[U],}
\]

\[
\text{in refer-}
\]
where the $\times$ on the right denotes the standard Cartesian product of sets. That is, an $(F \times G)$-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 24). However, instead of thinking of the labels as being duplicated, we think of an $(F \times G)$-structure as two structures which are superimposed on the same label set. In particular, when specifying the content for an $(F \times G)$-structure, we should still only map each label to a single piece of data.

Figure 23. Differentiating a cycle

![Figure 23](image)

Figure 24. Cartesian product of species

One interesting use of Cartesian product is to model some type class-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

$$ F \times (E \circ O_{\epsilon}). $$

Structures of this species consist of an $F$-structure with a superimposed partition on the labels, with each part corresponding to an equivalence class. For example, Figure 25 shows a binary tree shape with a superimposed partition indicating which sets of elements are equal. Likewise, we can model an $Ord$ constraint by superimposing an $(L \circ O_{\epsilon})$-structure, which additionally places an observable ordering on the equivalence classes.

This works particularly well in conjunction with the approach of Bernardy et al. [5] for testing polymorphic functions. Because of parametricity, it suffices to test polymorphic functions on randomly generated shapes filled with carefully chosen data; if the function works correctly for the chosen data then by parametricity it will work correctly for any data. The above discussion shows that we can treat $Eq$ and $Ord$ constraints as part of the shape of an input structure, and choose data to match.

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

$$(F \Box G)[U] = F[G[U]],$$

that is, $F$-structures over the set of all $G$-structures on $U$ (Figure 26). Like $(F \circ G)$-structures, an $(F \Box G)$-structure is an $F$-structure of $G$-structures, but instead of partitioning the labels $U$ among the $G$-structures, we give all the labels to every $G$-structure. As with Cartesian product structures, $(F \Box G)$-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.

Figure 25. A $(B \times (E \circ O_{\epsilon}))$-shape

![Figure 25](image)

**Functor composition**

$$\begin{align*}
\text{data } (f \circ g) a &= FC \{ \text{un}FC :: f (g a) \} \\
\text{instance } (Functor f, Functor g) &\Rightarrow \\
&\text{Functor } (f \circ g) \text{ where} \\
fmap h &= FC \circ (\text{fmap} \circ \text{fmap}) h \circ \text{un}FC \\
\text{instance } (Enumerable f, Enumerable g) &\Rightarrow \\
&\text{Enumerable } (f \circ g) \text{ where} \\
\text{enumerate} &= \text{map } FC \circ \text{enumerate} \circ \text{enumerate}
\end{align*}$$

Figure 26. Functor composition

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

$$\phi \Box (E_2 \bullet E)$$

defines the species of simple graphs. An $(E_2 \bullet E)$-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 $\phi \Box G$ for a suitable species $G$. For example, $G = X^2 \bullet E$ gives directed graphs without self-loops, and $G = \epsilon \times \epsilon$ gives directed graphs with self-loops allowed (recalling that $\epsilon = X \times \epsilon$ is the species of elements). The reader may enjoy discovering how to represent the species of undirected graphs with self-loops allowed.

6. **An embedded language of species**

We have defined a type corresponding to each primitive species and species operation, but we would also like to be able to write down
and compute with species expressions at the term level. The perfect way to do this is with a type class defining a domain-specific language of species expressions. The expressions can then be interpreted in different ways (for example, as exponential generating functions, cycle index series, abstract syntax trees, or random generation routines) depending on the types at which they are instantiated.

The basic Species type class, as defined in the species library, is shown in Figure 27. The actual Species class contains additional methods, but this is the core essence.

```haskell
class (Differential.C s) ⇒ Species s where
  singleton :: s
  set :: s
  cycle :: s
  (°) :: s → s → s
  (×) :: s → s → s
  □ :: s → s
  ofSize :: s → (Z → Bool) → s
```

Figure 27. The Species type class

Some things may seem to be missing (0, 1, sum and product, differentiation) but these are actually provided by the 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 (°), cartesian product (×), and functor composition (□) operations; and a cardinality-restricting operator ofSize.

It is not hard to put together the Enumerable instances we have already seen into code which enumerates all the labeled structures of a given species. The user can then call the enumerate method on an expression of type Species s ⇒ s, along with some labels to use:

```haskell
-- cycles of lists
> enumerate (cycle 'o' (nonEmpty linOrd)) "abc"
  :: [Comp Cycle [] Char]

-- simple graphs on three vertices
> enumerate (subset @0 @0 @2) [1,2,3]
  :: [Comp Set Set Int]
  = [[],{1,2},{1,3},{1,2},{2,3},{2,3},{2,3},{1,3},{1,2}]!

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

```haskell
> enumerate family [1,2] :: [Family Int]
  = [Group 2 [Group 1 [], Group 2 [Monkey True 1]],
     Group 2 [Monkey False 1],
     Group 1 [Group 2 [], Group 1 [Monkey True 2]],
     Group 1 [Monkey False 2]]
```

We can also control how the enumeration happens by explicitly specifying the species to use for the Family type, rather than using the default.

7. Generating functions and counting

What else can we do with combinatorial species? A key element of the story we haven’t seen yet is the correspondence between species and generating functions. Generating functions are an indispensable tool in combinatorics, and have a well-developed theory [23]. Much of their power lies in the surprising fact that many natural power series operations (addition, multiplication, substitution... ) have natural combinatorial interpretations (disjoint union, independent choice, partition... ). Every species can be associated with several different generating functions, each of which encodes certain aggregate information about the species.

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

\[ F(x) = \sum_{n \geq 0} \frac{f_n}{n!} x^n, \]

where \( f_n \) is the number of distinct labeled F-structures 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 [23].) Thus we have, for example,

- \( 0(x) = \frac{0}{0!} x^0 + \frac{1}{1!} x + \cdots = 0 \)
- \( 1(x) = 1 \)
- \( X(x) = x \)
- \( L(x) = \sum_{n \geq 0} \frac{1}{n!} x^n = \frac{1}{1 - x} \)
- \( E(x) = \sum_{n \geq 0} \frac{1}{n!} x^n = e^x \)
- \( C(x) = \sum_{n \geq 0} \frac{(n - 1)!}{n!} x^n = -\log(1 - x) \)

(Here is another good reason to call the species of sets \( E \))! At first glance this may seem arbitrary, but quite the opposite is true: species sum, product, composition, and differentiation correspond precisely to the same operations on formal power series! For example, if \( F(x) \) and \( G(x) \) count the number of labeled F- and G-structures as defined above, it is easy to see that \( F(x) + G(x) \) counts the number of labeled \( (F + G) \)-structures in the same way, since every \( (F + G) \)-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 \( F(x) G(x) = (F \bullet G)(x) \) as well:

\[
F(x) G(x) = \left( \sum_{n \geq 0} \frac{f_n}{n!} x^n \right) \left( \sum_{n \geq 0} \frac{g_n}{n!} x^n \right) = \sum_{n \geq 0} \left( \sum_{k=0}^{n} \frac{f_k}{k!} \frac{g_{n-k}}{(n-k)!} \right) x^n = \sum_{n \geq 0} \left( \sum_{k=0}^{n} \frac{n!}{k!} \frac{f_k g_{n-k}}{(n-k)!} \right) x^n = \sum_{n \geq 0} \left( \sum_{k=0}^{n} \frac{n!}{k!} \frac{f_k g_{n-k}}{(n-k)!} \right) x^n.
\]

The expression in the outermost parentheses is precisely the number of labeled \( (F \bullet G) \)-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; the interested reader should look up
*Faà di Bruno’s formula*. There are also generating function opera-
tions corresponding to Cartesian product and functor composition.
Although they are not as natural as the other operations, they are
simple to define and easy to compute.

As a result, we can count labeled structures by interpreting
species expressions as exponential generating functions, conve-
niently represented by infinite lazy lists of coefficients. In fact,
this particular technique of counting labeled structures has been
known in the functional programming community for some time
[19, 21]. The *species* library defines an *EGF* type with an appro-
priate *Species* instance, and a *labeled* function to extract the coef-
ficients from an egf. For example:

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

Thus, there are no \( C \circ L_1 \) structures of size 0, one with a single
label, three with two labels, 14 of size three, and so on. The library
can even compute generating functions for some recursively de-
defined species, using a quadratically converging combinatorial ana-
logue of the Newton-Raphson method. For example, once Dorothy
has used the *species* library to derive all the appropriate instances
for her *Family* type via Template Haskell, she can use the *labeled*
function to count them:

```haskell
> take 10 . labeled $ family
[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 al. [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 sur-
face; the theory of combinatorial species is both rich and deep. In
closing, we will look at some extensions to the theory discussed
here which may lead to a deeper understanding of functional pro-
gramming and data types, as well as potential applications of the
theory. At the time of writing, the *species* library does not include
support for any of these extensions, but their inclusion is planned
for future releases.

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 can define the species of binary
trees, weighted by the number of leaves, and then easily count or
enumerate only 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 gen-
eralizes straightforwardly to multisort species, which build struc-
tures from multiple sets of labels (*sorts*). For example, multisort

species are exactly what we need to make the implicit species the-
orem precise. First, we can write an implicit equation for \( F \) in the
form \( \tilde{F} = H(X, F) \), where \( H \) is a two-sort species. For example, if
\( H(X,Y) = 1 + X \cdot Y \) then \( L = H(X, L) \) is the implicit equation
defining the species of lists. Now the necessary conditions for the
implicit species theorem to apply can now be stated precisely:

- \( H(0,0) = 0 \) (no structures on the empty label set)
- \( \frac{\partial H}{\partial F}(0,0) = 0 \) (\( F \) does not trivially reduce to itself)

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 in a way analogous to the set-theoretic completion of the
natural numbers to the integers. We consider pairs of species \((F, G)\)
where \( F \) is considered “positive” and \( G \) “negative”; more precisely,
we define an equivalence relation on pairs of species such that
\( (F, G) \sim (H, K) \) if and only if \( F + K \) is isomorphic to \( G + H \),
and define virtual species as the equivalence classes of this rela-
tion. Virtual species allow us to define a multiplicative inverse for
the species \( E \) of sets, and from there to define multiplicative and
compositional inverses for other suitable species, solve differential
equations, and define a *combinatorial logarithm* which generalizes
the notion of structures built from connected components. Virtual
species allow us to give a sensible and consistent meaning to equa-
tions like \( L = 1/(1 - X) \).

**Molecular species**

**Definition 6.** A species \( F \) is *molecular* if all \( F \)-structures are
isomorphic (i.e. relabelings of one another).

For example, the species \( X^2 \) of ordered pairs is molecular, since
we can go from any ordered pair to any other by relabeling. On
the other hand, the species \( L \) of linear orderings is not molecular,
since any two list structures of different lengths are funda-
mentally of linear orderings is not molecular,

- The molecular species are precisely those that cannot be de-
  composed as the sum of two nonzero species.
- Every molecular species is equivalent to \( X^n \) “quotiented by some
  symmetries”; in particular, the molecular species of size \( n \) are in one-to-one correspondence with the conjugacy classes of
  subgroups of \( S_n \). This gives us a way to completely clas-
sify molecular species and to compute with them directly. For
  example, there are four conjugacy classes of subgroups of \( S_3 \),
  each representing a different symmetry on three locations: the
  trivial subgroup corresponds to \( X^3 \) itself (no symmetry); swapping
  two locations yields \( X \cdot E_2 \); cycling the locations yields \( E_3 \);
  and identifying all the locations yields \( E_3 \).
- Every species can be written uniquely (up to isomorphism and
  reordering of terms) as a sum of molecular species. This, com-
bined with the previous fact, immediately gives us a complete
classification of all combinatorial species. It also provides a
method for finding canonical representatives of virtual species:
given a pair \((F,G)\), decompose each into a sum of molecular
species and cancel any that occur in both \( F \) and \( G \). As a corol-
lary, we can always detect when a species that “looks” virtual
is actually non-virtual, such as \( L - 1 \).

Now we see why species with no nontrivial symmetries can
always be built from \( I \), \( X \), \(+\), and \( \cdot \); any species with no sym-
metries must be isomorphic to a sum of molecular species with
no symmetries; but molecular species with no symmetries must
be of the form \( X^n \). Hence regular species are always of the form
\( n_0 + n_1 X + n_2 X^2 + \ldots \) with \( n_i \in \mathbb{N} \). Adding a fixed point oper-
ator allows us to write down certain infinite such sums using only finite expressions.

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 [22] testing. In fact, Canou and Darrasse [7] have already created a library for random test generation in OCaml based on the ideas of combinatorial species. There has also been some interesting recent work by Duregård on automatic derivation of QuickCheck generators for algebraic data types [11], 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? Carette and Uszkay [8] have explored this question by creating a Haskell library allowing the user to program with species. Abbott et al. have explored a similar question from a more theoretical point of view, with their more general notion of quotient containers [3]. More work needs to be done to explain the precise relationship between containers and species, and to transfer these approaches into practical technology available to programmers.

Acknowledgments

I would like to thank the anonymous reviewers for many detailed and helpful comments, and Jeremy Gibbons for the initial encouragement a year ago to write this paper. This work was partially supported by the National Science Foundation, under grant 0910786 TRELLYS.

References