University of Pennsylvania
ScholarlyCommons

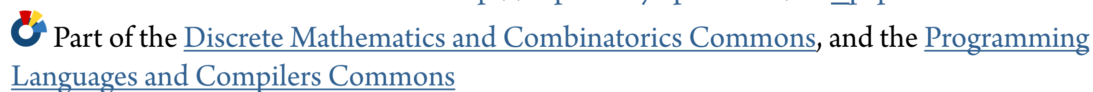
Departmental Papers (CIS) Department of Computer & Information Science

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


Part of the Discrete Mathematics and Combinatorics Commons, and the Programming Languages and Compilers Commons

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
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:

```
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,

```
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 $\mathsf{F} = 2 \bullet \mathsf{X} + \mathsf{X} \bullet (\mathsf{L} \circ \mathsf{F})$; 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 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 from" 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 $\sigma : U_1 \overset{\sim}{\leftrightarrow} U_2$ between two label sets U_1 and U_2 (*i.e.* a *relabeling*), "lifts" σ to a bijection between F-structures,

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

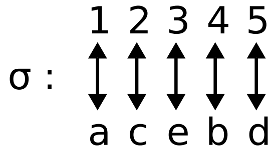
Haskell'10, September 30, 2010, Baltimore, Maryland, USA.

Copyright © 2010 ACM 978-1-4503-0252-4/10/09. . . $10.00

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 context.

For example, Figure 1 illustrates lifting a relabeling σ to a relabeling of binary trees.



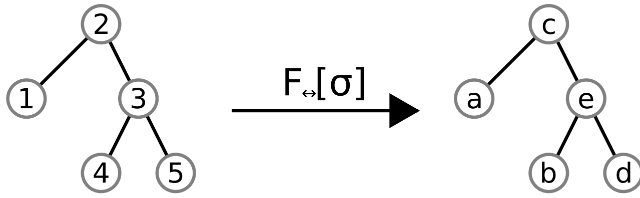


Figure 1. Relabeling a binary tree

Note that the notion of *structures* in this definition is entirely abstract: a "structure" is just an element of the set output by $\mathsf{F}_\bullet$, 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 $\mathbb{B}$, 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 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.

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.[1] 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.

[1] 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].

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 expressed by an instance of the *Functor* type class, whose method $\mathit{fmap} :: (a \to b) \to f\ a \to f\ b$ 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.

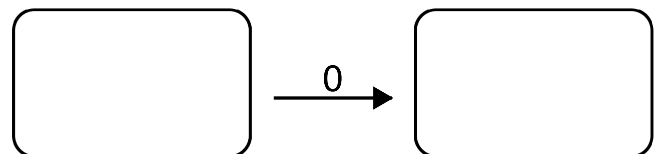
```
class Enumerable f where
  enumerate :: [a] → [f a]
```

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 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 $\mathsf{0}$ (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 $\mathsf{0}$, since Haskell does not allow empty case expressions.

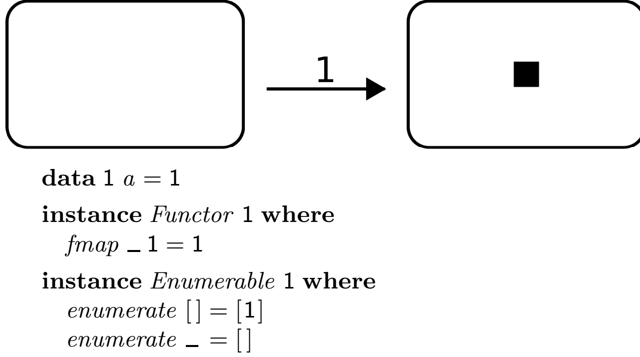



```
data 0 a

instance Functor 0 where
  fmap = ⊥

instance Enumerable 0 where
  enumerate _ = []
```

Figure 3. The primitive species $\mathsf{0}$

One The species $\mathsf{1}$ (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 $\mathsf{1}\ a$, and it contains no locations where values of type a can be stored. It

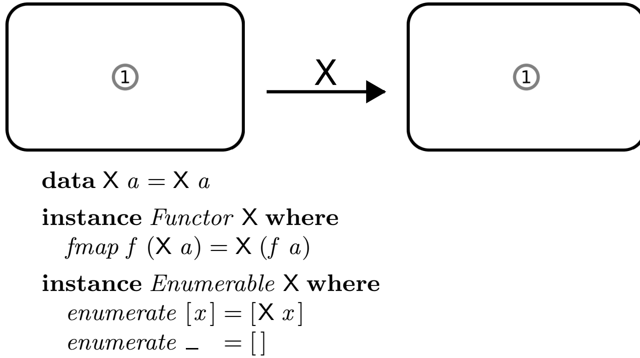
corresponds to nullary constructors in algebraic data types. The unit structure built by 1 is shown in Figure 4 as a filled square, to emphasize the fact that it contains no labels.




```
data 1 a = 1

instance Functor 1 where
  fmap _ 1 = 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

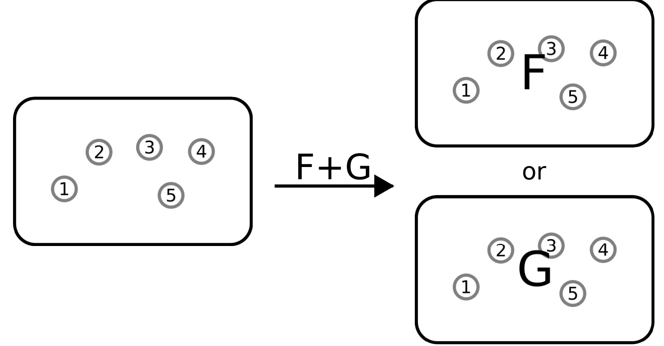
Species sum We define species sum (Figure 6) to correspond to type sum, *i.e.* disjoint (tagged) union. Given species F and G and a set of labels U, the set of $(\mathsf{F}+\mathsf{G})$-structures over U is the disjoint union of the sets of F- and G-structures over U:

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

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.

We can 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, **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.




```
infixl 6 +
data (f + g) a = Inl (f a) | Inr (g a)

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

instance (Enumerable f, Enumerable g) ⇒
         Enumerable (f + g) where
  enumerate ls = map Inl (enumerate ls)
              ++ map Inr (enumerate ls)

(+) :: (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

```
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, as implemented in the *species* library. The one technical issue to note is that for the congruences *inSumL* and *inSumR* (and the corresponding *inProdL* and *inProdR* 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 weak-head 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

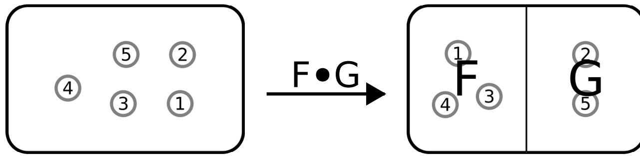
```
sumIdL :: 0 + f ↔ f
sumIdL = (λ(Inr x) → x) ↔ Inr

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

such a way that the resulting structures contain each label exactly once. So, to form an $(\mathsf{F} \bullet \mathsf{G})$-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 $(\mathsf{F} \bullet \mathsf{G})$. 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 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 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)
```

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, $\mathsf{1}$ is an identity for species product, and $\mathsf{0}$ 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, although their straightforward implementations are omitted in the interest of space.

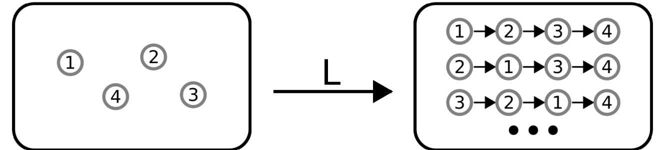
```
prodIdL     :: 1 • f ↔ f
prodIdR     :: f • 1 ↔ f
prodAbsorbL :: 0 • f ↔ 0
prodComm    :: f • g ↔ g • f
prodAssoc   :: f • (g • h) ↔ (f • g) • h
prodDistrib :: f • (g + h) ↔ (f • g) + (f • h)
inProdL     :: (f ↔ g) → (f • h ↔ g • h)
inProdR     :: (f ↔ g) → (h • f ↔ h • g)
```

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

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

That is, a list is either empty ($\mathsf{1}$) 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, $|\mathsf{L}[n]| = n!$.




```
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

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

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

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 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 $\mathsf{F}(0) = 0$, 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 $\mathsf{L}_+ + 1 = \mathsf{L}$, then we have $\mathsf{L}_+ = \mathsf{X} \bullet (\mathsf{L}_+ + 1)$, which does meet the criteria and hence has a unique solution. Thus, $\mathsf{L} = 1 + \mathsf{L}_+$ 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 $\mathsf{L} = 1 + \mathsf{X} + \mathsf{X}^2 + \mathsf{X}^3 + \dots$, 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 $\mathsf{L} = \frac{1}{1-\mathsf{X}}$. 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 BTree a = Empty
             | Node a (BTree a) (BTree a)
data Paren a = Leaf a
             | Pair (Paren a) (Paren a)
```

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).

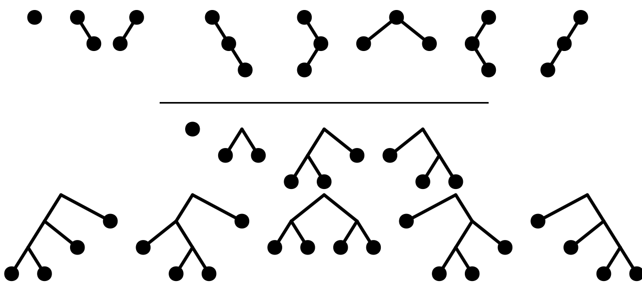


Figure 13. Binary trees (top) and parenthesizations (bottom)

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, B itself does not fulfill the conditions of the implicit species theorem, but we can easily express it in terms of one that does.

```
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

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*?

Well, pairing an extra element with a *BTree* corresponds to the species $\mathsf{X} \bullet \mathsf{B}$. 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 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, $+$, $\bullet$, 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 $\mathcal{S}_n$ 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 $\sigma \in \mathcal{S}_n$ 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.

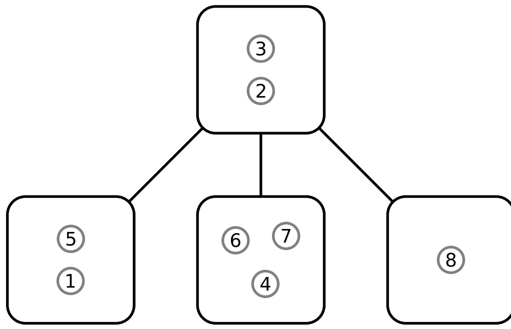


Figure 16. A labeled structure with nontrivial symmetries

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 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,

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

cannot be written as a simple algebraic data type.[2] But aside from infinite sums and products, as long as we stick to data structures with no symmetries, Haskell's data types are perfectly adequate to express any data structure we could possibly think up.

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

3.3 Other operations on regular species

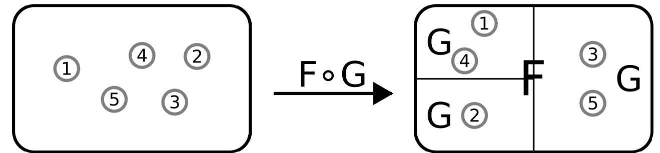
In addition to sum and product, the class of regular species is also closed under other fundamental operations.

Species composition 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, $U = U_1 \uplus U_2 \uplus \cdots \uplus U_k$;
- 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.




```
newtype (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*, as defined in `Data.Tree`, 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 $(\mathsf{F} \bullet \mathsf{G}) \circ \mathsf{H}$.

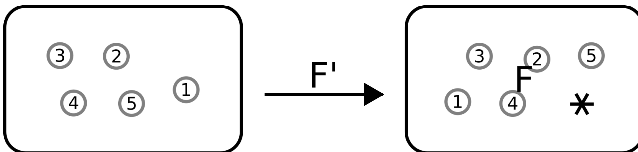
As noted at the beginning of this section, regular species are closed under composition. Although we won't prove this formally, it makes intuitive sense: if an F-structure has no symmetries, and in each location we put a G-structure which also has no symmetries, the resulting composed structure cannot 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 $U \cup \{*\}$, where $*$ is a new element distinct from all elements of U. That is,

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




```
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 18. Species differentiation

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

Figure 18 shows a representation of this idea in Haskell. It is important to note that $\mathit{Diff}\ f\ a$ will often admit values which are not among those listed by $\mathit{enumerate}$. For example, although as a Haskell type $\mathit{Diff}\ 1\ \mathit{Int}$ is perfectly well inhabited by the value $\mathit{Diff}\ 1$, $\mathit{enumerate}$ will always produce the empty list at type $\mathit{Diff}\ 1\ \mathit{Int}$. Isomorphisms between structure types should map between values listed by $\mathit{enumerate}$ and not necessarily between all Haskell values of the given types; thus, for example, we are justified in treating $\mathit{Diff}\ 1$ as isomorphic to $\mathsf{0}$.

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 5).

Why refer to this operation as *differentiation*? Simply because, somewhat astonishingly, it satisfies all the same algebraic laws as the differentiation operation in calculus! Explaining the intuition behind these isomorphisms and expressing them in Haskell is left 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

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

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

Unfortunately, it is difficult to represent general cardinality restriction with a Haskell type, since we would have to somehow embed a predicate on integers into the type level.

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 σ such that $\mathsf{F}_{\leftrightarrow}[\sigma](s) = t$.

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.

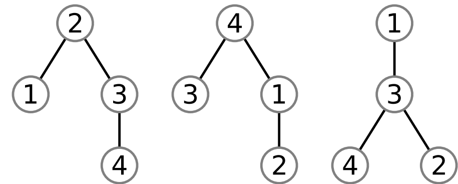


Figure 19. Equivalent and inequivalent trees

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

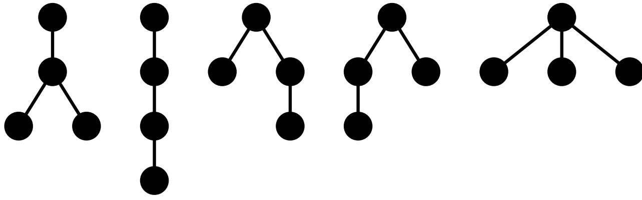


Figure 20. Unlabeled rose trees of size 4

values. Enumerating unlabeled structures for *non-regular* species, however, is much more complicated; a partial implementation of unlabeled enumeration can be found in the *species* package. For example, we can enumerate all unlabeled sets of sets of size 4, corresponding to integer partitions of 4:

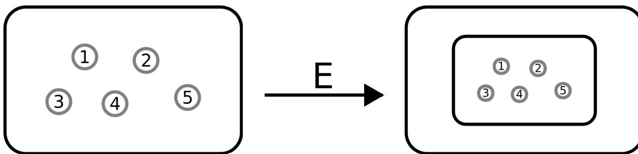
```
> enumerateU (set ‘o‘ nonEmpty set) 4
        :: [Comp Set Set ()]
[{{(),(),(),()}},{{(),()},{(),()}},
 {{()},{(),(),()}},{{()},{()},{(),()}},
 {{()},{()},{()},{()}}]
```

5. Beyond regular species

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 combinators.

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.



$$\begin{array}{l}
\mathbf{newtype}\ \mathit{Bag}\ a = \mathit{Bag}\ [a] \\
\quad \mathbf{deriving}\ \mathit{Functor} \\
\mathbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ (\mathit{Bag}\ a)\ \mathbf{where} \\
\quad \mathit{Bag}\ xs \equiv \mathit{Bag}\ ys = xs\ \text{‘}\mathit{subBag}\text{‘}\ ys \wedge ys\ \text{‘}\mathit{subBag}\text{‘}\ xs \\
\quad\quad \mathbf{where}\ \mathit{subBag}\ b = \mathit{null} \circ \mathit{foldl}'\ (\mathit{flip}\ \mathit{delete})\ b \\
\mathbf{instance}\ \mathit{Enumerable}\ \mathit{Bag}\ \mathbf{where} \\
\quad \mathit{enumerate}\ ls = [\mathit{Bag}\ ls]
\end{array}$$

Figure 21. The species E of sets

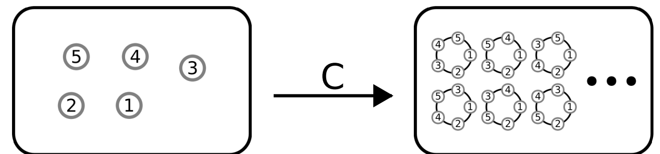
In Figure 21, we declare a Haskell data type for E by declaring Bag to be isomorphic to $[\,]$ via a **newtype** declaration, and deriving a $\mathit{Functor}$ instance for Bag from the existing instance for lists, using GHC's *newtype deriving* extension. Since we don't care about the order of the elements in a Bag, we create an Eq instance to identify any two Bags with the same elements.

We can also use E to build 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. Thus, 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 describing structures containing only a subset of the labels.
- $\mathsf{E} \bullet \mathsf{E}$ is the species of *subsets*, sometimes 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 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.



$$\begin{array}{l}
\mathbf{newtype}\ \mathit{Cycle}\ a = \mathit{Cycle}\ [a] \\
\quad \mathbf{deriving}\ \mathit{Functor} \\
\mathbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ (\mathit{Cycle}\ a)\ \mathbf{where} \\
\quad \mathit{Cycle}\ xs \equiv \mathit{Cycle}\ ys = \mathit{any}\ (\equiv ys)\ (\mathit{rotations}\ xs) \\
\quad\quad \mathbf{where}\ \mathit{rotations}\ xs = \mathit{zipWith}\ (+\!\!+)\ (\mathit{tails}\ xs) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\mathit{inits}\ xs) \\
\mathbf{instance}\ \mathit{Enumerable}\ \mathit{Cycle}\ \mathbf{where} \\
\quad \mathit{enumerate}\ [\,] = [\,] \\
\quad \mathit{enumerate}\ (x : xs) \\
\quad\quad = (\mathit{map}\ (\mathit{Cycle} \circ (x:)) \circ \mathit{permutations})\ xs
\end{array}$$

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 $\mathsf{E} \circ \mathsf{C}$, 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 23).

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

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

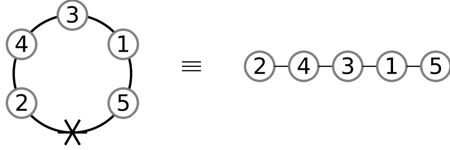
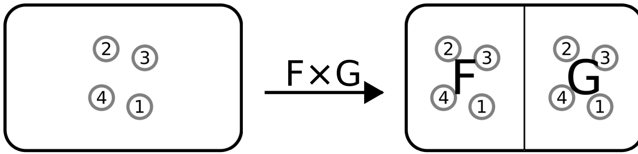


Figure 23. Differentiating a cycle

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 24). 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 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

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

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 $(\mathsf{L} \circ \mathsf{E}_+)$-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

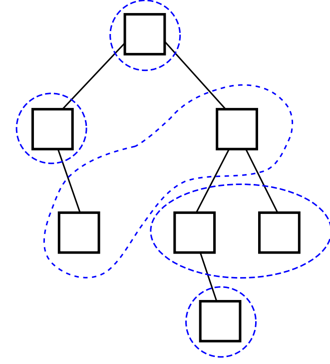
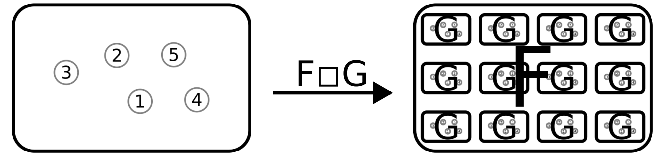


Figure 25. A $(\mathsf{B} \times (\mathsf{E} \circ \mathsf{E}_+))$-shape

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 26). Like $(\mathsf{F} \circ \mathsf{G})$-structures, an $(\mathsf{F} \,\square\, \mathsf{G})$-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, $(\mathsf{F} \,\square\, \mathsf{G})$-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 26. 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.

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.

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

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 ($\circ$), cartesian product ($\times$), and functor composition ($\Box$) 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 $\mathit{Species}\ s \Rightarrow s$, along with some labels to use:

```
-- cycles of lists
> 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">]

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

```
> 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

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

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,

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

(Here is another 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 $\mathsf{F}(x)$ 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 precisely the number of labeled $(\mathsf{F} \bullet \mathsf{G})$-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; 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 easy to compute.

As a result, we can count labeled structures by interpreting species expressions as exponential generating functions, conveniently 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 appropriate $Species$ instance, and a $labeled$ function to extract the coefficients from an egf. For example:

```
> take 10 . labeled $ 3 + x*x
[3,0,2,0,0,0,0,0,0,0]

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

Thus, there are no $\mathsf{C} \circ \mathsf{L}_+$ 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 defined species, using a quadratically converging combinatorial analogue 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:

```
> 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 surface; 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 programming 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 generalizes straightforwardly to *multisort* species, which build structures from multiple sets of labels (*sorts*). For example, multisort species are exactly what we need to make the implicit species theorem precise. First, we can write an implicit equation for F in the form $\mathsf{F} = \mathsf{H}(\mathsf{X}, \mathsf{F})$, where H is a two-sort species. For example, if $\mathsf{H}(\mathsf{X}, \mathsf{Y}) = 1 + \mathsf{X} \bullet \mathsf{Y}$ then $\mathsf{L} = \mathsf{H}(\mathsf{X}, \mathsf{L})$ 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:

- $\mathsf{H}(0, 0) = 0$ (no structures on the empty label set)
- $\frac{\partial \mathsf{H}}{\partial \mathsf{Y}}(0, 0) = 0$ (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 $(\mathsf{F}, \mathsf{G}) \sim (\mathsf{H}, \mathsf{K})$ if and only if $\mathsf{F} + \mathsf{K}$ is isomorphic to $\mathsf{G} + \mathsf{H}$, and define virtual species as the equivalence classes of this relation. 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 equations like $\mathsf{L} = 1/(1 - \mathsf{X})$.

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 fundamentally non-isomorphic. We have the following three beautiful facts:

- The molecular species are precisely those that cannot be decomposed 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 $\mathcal{S}_n$. This gives us a way to completely classify molecular species and to compute with them directly. For example, there are four conjugacy classes of subgroups of $\mathcal{S}_3$, each representing a different symmetry on three locations: the trivial subgroup corresponds to X^3 itself (no symmetry); swapping two locations yields $\mathsf{X} \bullet \mathsf{E}_2$; cycling the locations yields to C_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, combined 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 corollary, we can always detect when a species that "looks" virtual is actually non-virtual, such as $\mathsf{L} - 1$.

Now we see why species with no nontrivial symmetries can always be built from 1, X, $+$, and $\bullet$: any species with no symmetries 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 $n_0 + n_1\mathsf{X} + n_2\mathsf{X}^2 + \ldots$ 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

[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. Available at `http://www.cas.mcmaster.ca/~carette/species/`, 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] Duncan Coutts, Isaac Potoczny-Jones, and Don Stewart. Haskell: batteries included. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 125–126, New York, NY, USA, 2008. ACM.

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

[12] 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.

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

[14] Gérard Huet. Functional pearl: The zipper. *J. Functional Programming*, 7:7–5, 1997.

[15] 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.

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

[17] 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.

[18] 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.

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

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

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

[22] 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.

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