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:

$$\begin{aligned}&\mathbf{data}\ \mathit{Family}\ a = \mathit{Monkey}\ \mathit{Bool}\ a\\ &\qquad\qquad\ \ \mid\ \mathit{Group}\ a\ [\mathit{Family}\ a]\end{aligned}$$

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,

$$\mathbf{newtype}\ \mathit{Bag}\ a = \mathit{Bag}\ [a]$$

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 $\sigma : U_1 \xleftrightarrow{\sim} U_2$ between two label sets U_1 and U_2 (*i.e.* a *relabeling*), "lifts" σ to a bijection between F-structures,

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

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