Species: making analytic functors practical for functional programming

Jacques Carette Gordon Uszkay

Computing and Software
McMaster University
Hamilton, Canada

Abstract

Inspired by Joyals theory of species, we show how to add new type constructors and constructor combinators to the tool set of functional languages. We show that all the important properties of inductive types lift to this new setting. Species are analytic functors, representing a broader range of structures than regular functors. This includes structures such as bags, cycles and graphs. The theory is greatly inspired by combinatorics rather than type theory: this adds interesting new tools to bear, but also requires more work on our part to show that species form a good foundations for a theory of type constructors. The combinatorial tools provide a calculus for these structures which has strong links with classical analysis, via a functorial interpretation of generating series. Furthermore, we show how generic programming idioms also generalise to this richer setting. Once the theory is understood, various methods of implementation are relatively straightforward.

Keywords: functional programming, combinatorial species, datatype generic programming.

1 Introduction

In typed functional languages, types are built inductively from base types using a few combinators: disjoint union (tagged sum), product, composition and recursion. Semantically, polynomial functors and (initial) algebras have been found to be the most convenient model. In practice and in theory, these have proven their worth. As is standard in the literature, we will refer to these types (and their constructors) as "polynomial types" (without recursion) and "regular types".

We will borrow from a vast treasure trove of work done in combinatorics and applied category theory, and apply their results to functional programming. We aim to show that we can cleanly generalize from polynomial types to "analytic types". The key is the theory of combinatorial species. Introduced by André Joyal in 1981 ([27], but also see [28] and the excellent book [5]), the combinatorial theory of species provides a categorical approach to defining families, or "species", of structures.

Informally, a species is the set of structures constructed by some "rule" over a set of discrete elements, where the structure does not depend on the value of the elements. Formally, a species is a functor from the category $\mathbb{B}$ of finite sets and bijections to the category $\mathbb{E}$ of finite sets and functions[1] These functors are called analytic because they permit a Taylor series expansion

$$F(A) = \sum_{n \geq 0} \left(A^n \times F_n\right) / \mathfrak{G}_n$$

where $\mathfrak{G}_n$ is the symmetric group over A^n. One can fruitfully think of A^n as a coefficient and $\mathfrak{G}n$ as $n!$, making $F(A)$ into an exponential generating function.

Our goal is to demonstrate that it is both possible and pratical to define type constructors as analytic functors using the theory of species, and that the main tools in use for regular types generalize to this setting. In particular, we

- show that species give us useful extensions to the theory of regular types,
- get new type constructors, for example bags, cycles, and new type combinators which should be considered *basic*,
- explore how the molecular theory of species provides a foundation for generating type constructors for unlabelled types,
- show that the implicit species theorem gives solutions for many combinatorial equations, and compare those solutions obtained by computing least fixed points,
- show how the combinatorial tools already in existence also apply to types,
- briefly explore how datatype generic programming techniques extend to species.

We start with an overview of the theory of species, and then detail how this can be used in a practical setting (with examples). This is followed by an outline of more advanced species theory and its potential applications. There are many related strands in the literature, and we attempt to put our work in perspective. Finally, we conclude and suggest directions for future research.

2 Basic Theory of Species

We provide an introduction to the theory of species, tailored to its uses in programming. The interested reader should consult [5] for a much more thorough treatment of most of the material we present here. For consistency, we use the syntactic conversions of [5]. We assume that the reader is familiar with polynomial types and category theory.

Informally, a structure is a pair of a function and set, (γ, U) where γ is a construction performed on U. A structure consists of labelled "nodes" [where data elements may be stored], and the relationship between these nodes. The set U acts as a set of *labels*. For the implementation minded reader, one may think of U as containing the keys into a hash table, with this table representing a heap.

A species $\mathcal{F}$ of structures is a functor from $\mathbb{B}$, the category of finite sets and bijections, to $\mathbb{E}$, the category of finite sets and functions. As this definition is so central, let us expand this definition: A species $\mathcal{F}$ is a rule that

(i) produces a finite set $\mathcal{F}[U]$ for each finite set U;

[1] although most modern treatments use $Set^{\mathbb{B}}$, we will proceed with classical definition as this extra generality is not needed in practice.

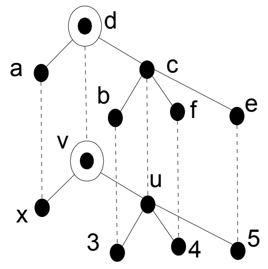

Fig. 1. Two structures of the same species, with a bijection transporting one to the other.

(ii) for each bijection $\sigma : U \to V$ produces a function $\mathcal{F}[\sigma] : \mathcal{F}[U] \to \mathcal{F}[V]$; and

(iii) the functions $\mathcal{F}[\sigma]$ need to preserve composition and the identity.

An individual structure from $\mathcal{F}$ over the set of elements U is called an $\mathcal{F}$ structure and is denoted $\mathcal{F}[U]$.

Formally, a structure transports along a bijection over the underlying sort, so for all bijections $\sigma : U \to V$, $\mathcal{F}[\sigma] : \mathcal{F}[U] \to \mathcal{F}[V]$ is also a bijection, preserving both composition and the identity. Figure 1 illustrates transport of structure.

Such a semantic description of species gives us ample choice in providing different syntactic descriptions. We will mostly concentrate on explicit constructions and functional equations.

Note that we could have restricted ourselves to the full subgroupoid $\mathbb{P}$ of $\mathbb{B}$ consisting of finite cardinals and bijections as $\mathbb{E}^{\mathbb{B}}$ and $\mathbb{E}^{\mathbb{P}}$ are equivalent functor categories; in practice this is reflected by the fact that we use (numerical) machine pointers as labels.

2.1 Notation

When a species $\mathcal{F}$ is applied to a set of elements A, it forms a set of possible concrete structures, denoted $\mathcal{F}[A]$. If a species is restricted to a particular cardinality n, then it is denoted $\mathcal{F}_n$, and is undefined over sets not of cardinality n.

Note that the standard names and symbols associated with species are often French, so a set is an "ensemble" and denoted $\mathcal{E}$, trees are arbres ($\mathcal{A}$), etc.

2.2 Simple Species

0 The species defined by $\mathbf{0}[U] = \emptyset$ for all sets U.

1 The characteristic specie of the empty set, $\mathbf{1}[U] = \begin{cases} \{\emptyset\} & \text{if } U = \emptyset \\ \emptyset & \text{otherwise} \end{cases}.$

$\mathcal{E}$ The set [2] specie defined by $\mathcal{E}[U] = U$. There is a unique $\mathcal{E}$-structure on any finite set U, namely U itself.

[2] This is the combinatorial version of set, in computer science this is generally called a "bag"; there is no guarantee of unique "values" among the elements.

$\mathcal{X}$ The singleton set, defined by $\mathcal{X}[U] = \begin{cases} \{U\} & \text{if } |U| = 1 \\ \emptyset & \text{otherwise} \end{cases}$.

$\mathcal{C}$ An ordered cycle. $\mathcal{C}[U] = \{c \mid c = (\pi, U), \pi \text{ is a circular permutation}\}$.

$\mathbf{0}$, $\mathbf{1}$ and $\mathcal{X}$ are familiar from polynomial types.

$\mathcal{E}$ is often also denoted $\exp(\mathcal{X})$ since it satifies the combinatorial differential equation $\mathcal{E}' \simeq \mathcal{E}$ (as well as having the same generating series as the exponential function, see 2.8). Another species, denoted either $\in$ or U, is the "underlying set" specie (defined by $\in[U] = U$) plays a minor rôle in the basic theory, but seems to have non-trivial applications in more advanced settings.

2.3 Species Operators

As with polynomials, we have several combinators for species. The first 5 are binary, while the last 3 are unary. Below we specify the action on the underlying sets, although a complete definition would also specify the transport of structures, but we omit these here as they are generally straightforward.

$+$ **(Disjoint Union)** $(\mathcal{F} + \mathcal{G})[U] = \mathcal{F}[U] + \mathcal{G}[U]$ where we use $+$ for disjoint union of sets.

$\cdot$ **(Product)** Elements are partitioned between two structures, one from each species; the product operator is sometimes denoted by a space. More precisely $(\mathcal{F} \cdot \mathcal{G})[U] = \sum_{U_1+U_2=U} \mathcal{F}[U_1] \times \mathcal{G}[U_2]$.

$\circ$ **(Composition, or Substitution)** Elements of U are partitioned into disjoint subsets which are given $\mathcal{G}$-structures, and an $\mathcal{F}$-structure is put on the resulting set of $\mathcal{G}$-structures. More precisely,

$$(\mathcal{F} \circ \mathcal{G})[U] = \sum_{\pi \text{ partition of } U} \mathcal{F}[\sum_{p\in\pi} \mathcal{G}[p]]$$

$\square$ **(Functorial Composition)** $\mathcal{F} \square \mathcal{G}$ is the family of $\mathcal{F}$-structures over the set of all possible $\mathcal{G}$-structures over the underlying set. It corresponds to the composition of functors $(\mathcal{F} \square \mathcal{G})[U] = \mathcal{F}\left[\mathcal{G}\left[U\right]\right]$.

$\times$ **(Cartesian Product)** A compound structure consisting of one structure from each species, with all of the elements in both structures, $(\mathcal{F} \times \mathcal{G})[U] = \mathcal{F}[U] \times \mathcal{G}[U]$.

$^\bullet$ **(Pointing)** Identifies a single distinguished element in the structure. For example, the root of a tree is a distinguished element. $\mathcal{F}^\bullet[U] = \mathcal{F}[U] \times U$

$'$ **(Differentiation)** The structures generated by replacing an element the original structures with a "hole" for an element (figure 3). $\mathcal{F}'[U] = \mathcal{F}[U + 1]$ (the disjoint union of U with a new singleton).

$\mathcal{F}_n$**,** $\mathcal{F}_{\leq n}$**,** $\mathcal{F}_{>n}$**, etc – Cardinality Restriction** Restricts the species to structures over sets of exactly n elements, less than or equal to n elements, greater than n elements, etc. $\mathcal{F}_n[U] = \begin{cases} F[u] & |U| = n \\ \emptyset & \text{otherwise} \end{cases}$

The sum, product and composition are the same as for polynomial types, and in fact so is the derivative [3]. We also have that $\mathbf{2} = \mathbf{1} + \mathbf{1}$ and $\mathbf{n} = \mathbf{1} + \mathbf{1} + \cdots + \mathbf{1}$

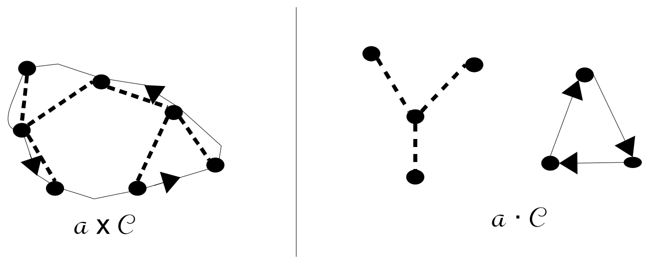

Fig. 2. Cartesian product vs. product of a cycle and a tree.

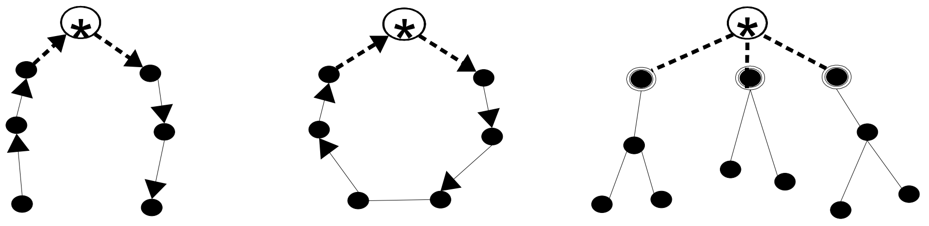

Fig. 3. Derivative of a list (two lists), a cycle (a list) and a tree (rooted trees).

(n times) are sensible definitions which give us that $\mathcal{F} + \mathcal{F} = 2\mathcal{F} = \mathbf{2} \cdot \mathcal{F}$. We can further prove all the expected properties: $+$ and $\cdot$ are associative and commutative, $\mathbf{0}$ is the unit for $+$, $\mathbf{1}$ is the unit for $\cdot$, $'$ satisfies all the usual rules (including that for composition, i.e. the chain rule), and so on.

Lemma 2.1 *For all species $\mathcal{F}$, $\mathcal{F}^{\bullet} \simeq \mathcal{X} \cdot \mathcal{F}'$.*

In some sense, pointing is even more primitive than derivative. The Cartesian product is especially interesting as it is a way to see the data data in two different ways, clearly related to *views* [39]. For example, $(\mathcal{A} \times \mathcal{C})[A]$ represents both a tree and a cycle over the set A, with all of the elements of A in both structures (figure 2).

Cardinality restriction is a clear contribution from combinatorics, but also turns out to be rather useful in programming practice, as we shall see later. As every structure has a particular size (given by the size of its label set), we have that all species have a *canonical decomposition*

$$\mathcal{F} = \sum_{n \geq 0} \mathcal{F}_n.$$

2.3.1 Atomic and Molecular Species

The previous two sections gave some tools for constructing species. We also have some tools for decomposing them. As theorem 2.5 shows, this decomposition is complete.

Definition 2.2 A species of structures $\mathcal{F}$ is *molecular* if there is only one isomorphism type of $\mathcal{F}$-structures, i.e. if any two arbitrary $\mathcal{F}$-structures are isomorphic.

In other words, $\mathcal{F}$ is molecular if and only if $\mathcal{F} \neq \mathbf{0}$, $\mathcal{F} = \mathcal{F}_n$ for some integer n, and the action induced by transport of structures $S[n] \times \mathcal{F}[n] \to \mathcal{F}[n]$ (with $S[n]$ the symmetric group of order n) is *transitive*. In other words, the classification of molecular species is equivalent to the classification of transitive actions of the symmetric group, which itself is equivalent to the classification of the conjugacy classes of sub-groups of the symmetric group [5].

For example, the species of lists of length n, $\mathcal{L}_n$, of cycles of length n, $\mathcal{C}_n$, and of bags of size n, $\mathcal{E}_n$ are all molecular.

Proposition 2.3 *$\mathcal{F}$ is molecular if and only if for all species $\mathcal{P}$, $\mathcal{Q}$ where $\mathcal{F} = \mathcal{P} + \mathcal{Q}$ then either $\mathcal{P} = \mathbf{0}$ or $\mathcal{Q} = \mathbf{0}$. As a consequence we have that all species $\mathcal{F}$ is the sum of its molecular sub-species*

$$\mathcal{F} = \sum_{\substack{\mathcal{M} \subset \mathcal{F} \\ \mathcal{M} \textit{ molecular}}} \mathcal{M}.$$

This sum is called the *molecular decomposition* of $\mathcal{F}$. This is a refinement of the canonical decomposition. Note that the product $\mathcal{F} \cdot \mathcal{G}$ and composition $\mathcal{F} \circ \mathcal{G}$ (for $\mathcal{G} \neq \mathbf{1}$) of molecular species is always molecular.

Definition 2.4 A species of structure $\mathcal{F} \neq \mathbf{1}$ is *atomic* if it is indecomposable (except trivially) as a product.

Yeh in [40] proved the following important result (with a rather delicate proof):

Theorem 2.5 *Every molecular specie $\mathcal{M}$ can be written as a finite product of atomic species*

$$\mathcal{M} = A_1^{n_1} A_2^{n_2} \cdots A_k^{n_k}$$

where each A_i is atomic and distinct, $n_i \in \mathbb{N}$, $i = 1, \ldots, k$. This decomposition is unique up to order of the factors and isomorphism.

For example $\mathcal{X}$, $\mathcal{E}_n$, $\mathcal{C}_n$ are atomic species, as are $\mathcal{E}_2 \circ \mathcal{E}_2$ and $\mathcal{E}_2 \circ \mathcal{X}^2$. Other atomic species include n-polygons (adirectional cycles), strongly connected n-graphs, various quotient species, as well as a zoo of structures which are rather difficult to describe. A list or all atomic species of cardinatlity $n \leq 5$ is provided in figure A.1.

2.4 Decomposition

Define $\mathfrak{Spe} = (S, \mathbf{0}, \mathbf{1}, +, \cdot) \,/ \sim$ (with S the set of all species and $\sim$ the equivalence relation induced by isomorphism of species) then $\mathfrak{Spe}$ is easily shown to form a *semi-ring*, and with $\prime$, a differential semi-ring. More importantly, if we denote by $\mathfrak{U}$ the (countable) set of all atomic species, then

Proposition 2.6 *The semi-ring $\mathfrak{Spe}$ of species is isomorphic to the semi-ring $\mathfrak{Spe} \simeq \mathbb{N}[\![\mathfrak{U}]\!] \simeq \mathbb{N}[\![\mathcal{X}, \mathcal{E}_2, \mathcal{E}_3, \mathcal{C}_3, \ldots]\!]$ of formal series with coefficients in $\mathbb{N}$ over the set of atomic species.*

Thus we have that $\mathcal{E} = \mathcal{E}_0 + \mathcal{E}_1 + \mathcal{E}_2 + \ldots$ and $\mathcal{C} = \mathcal{C}_1 + \mathcal{C}_2 + \mathcal{C}_3 + \ldots$ and functional programmers intuitively know the expansion $\mathcal{L} = \mathbf{1} + \mathcal{X} + \mathcal{X}^2 + \mathcal{X}^3 + \ldots$. More interesting is the molecular decomposition of the species $\mathcal{A}$ of trees (see figure A.2)

$$\begin{aligned}\mathcal{A} = \mathcal{X} + \mathcal{X}^2 + (\mathcal{X}^3 + \mathcal{X}\mathcal{E}_2) + (2\mathcal{X}^4 + \mathcal{X}^2\mathcal{E}_2 + \mathcal{X}\mathcal{E}_3) \\ + (3\mathcal{X}^5 + 4\mathcal{X}^3\mathcal{E}_2 + \mathcal{X}^2\mathcal{E}_3 + X(\mathcal{E}_2 \circ \mathcal{X}^2) + \mathcal{X}\mathcal{E}_4) + \dots\end{aligned}$$

2.5 Functional Equations

Just as in programming practice, we have some structures which are defined explicitly by some finite construction. But most structures of real interest are either defined by, or are shown to satisfy, a functional equation. For example we have that $\mathcal{L} \simeq \mathbf{1} + \mathcal{X}\mathcal{L}$. The common interpretation of that equation is as a least-fixed-point equation over a CPO. However, the strong links with classical analysis (see 2.8) have led to a different approach for species. The statement uses 2-sorted species (Funtors $F : \mathbb{B}^2 \to \mathbb{E}$), where the theory of k-sorted species is defined in complete analogy to the 1-sorted case.

Theorem 2.7 (Implicit Species Theorem [27]) *Let $H = H(X,Y)$ be a 2-sorted specie such that $H(\mathbf{0},\mathbf{0})$ is constant and $\frac{\partial H}{\partial Y}(\mathbf{0},\mathbf{0}) = \mathbf{0}$. Then the combinatorial equation $A = H(X,A)$ has a unique solution $A(X)$ (up to isomorphism) such that $A(\mathbf{0}) = H(\mathbf{0},\mathbf{0})$.*

A functor $F : \mathbb{E}^{\mathbb{B}} \to \mathbb{E}^{\mathbb{B}}$ is constant at $\mathbf{0}$ if there exists $K \in \mathbb{E}$ such that for every object $A \in \mathbb{E}^{\mathbb{B}}$, $(FA)\mathbf{0} = K$, and every arrow $f : A \to B$, $(Ff)\mathbf{0} = id_K : K \to K$. Joyal's original proof used $K = \emptyset$, but this can be generalized (see [30] for a proof for a slightly different case which can easily be adapted). This condition is necessary to insure that a *non-empty* finitary solution exists.

The statement of this theorem is essentially identical to the classical implicit function theorem. In the theory of species, it is this theorem which is used instead of Lambek's Lemma as the method of choice for establishing that equations have solutions. Interestingly, the proof uses an explicit iterative scheme to obtain a solution to the equation as a series expansion. In other words, the proof is constructive. Note that one can avoid 2-sorted species altogether (as Menni does), at the cost of losing the perfect parallel with classical analysis.

There are other functional equations for which such solutions are known to exist. In certain settings, certain (combinatorial) *differential* equations can also be shown to have unique solutions. The theory can also be applied to systems of functional equations.

2.6 Initial Algebras

There is a known link with the approach via initial algebra, well explained by Menni in [30]. We only mention the highlights.

Definition 2.8 A functor $F : \mathcal{C} \to \mathcal{C}$ is *special* if the F-algebra Alg_F has an intial object and moreover for every F-algebra A, A is a fixed point of F if and only if A is initial.

Definition 2.9 For species $\mathcal{F}, \mathcal{G}$ and operator $\Phi : \mathbb{E}^{\mathbb{B}} \to \mathbb{E}^{\mathbb{B}}$ (an endofunctor over species),

(i) $\mathcal{F}, \mathcal{G}$ have *contact of order n* if $\mathcal{F}_{\leq n} = \mathcal{G}_{\leq n}$, denoted $\mathcal{F} =_n \mathcal{G}$, meaning that for $1 \leq k \leq n, \mathcal{F}_n$ is identical to $\mathcal{G}_n$

(ii) Φ *preserves* contacts if $\forall n. \mathcal{F} =_n \mathcal{G} \implies \Phi\mathcal{F} =_n \Phi\mathcal{G}$

(iii) Φ *raises* contacts if $\forall n. \mathcal{F} =_n \mathcal{G} \implies \Phi\mathcal{F} =_{n+1} \Phi\mathcal{G}$.

The properties of $\mathfrak{Spe} = \mathbb{E}^{\mathbb{B}}$ give us that for any functor $F : \mathfrak{Spe} \to \mathfrak{Spe}$ which preserves directed colimits, Alg_F has an initial object.

Proposition 2.10 *If Φ is constant at* $\mathbf{0}$ *and raises contacts, then Φ is* special.

The important point here is that this does not require the extra baggage that comes with CPOs, and that one can still find solutions to functional equations in set-theoretic settings. To us, the implicit species theorem is a very powerful too that deserves further study.

2.7 Some Examples

Some less familiar but quite useful structures are easily represented. For example, commutative parenthizations is the species that represents arithmetic expressions.

$\wp = \mathcal{E} \cdot \mathcal{E}$	powerset, i.e. all subsets
$\wp^{[k]} = \mathcal{E}_k \cdot \mathcal{E}$	subsets of cardinality k
$Bal = \mathcal{L} \circ \mathcal{E}_+$	ballots, ordered (non-empty) partitions
$Par = \mathcal{X} + \mathcal{E}_2 \circ \mathcal{P}$	commutative parenthizations
$Oct = \mathcal{C} \circ \mathcal{L}$	octopus, a cycle of lists
$\mathcal{G} = \wp \square \wp^{[2]}$	simple graphs
$\mathcal{G}_o = \wp \square (\mathcal{E}^\bullet \times \mathcal{E}^\bullet)$	connected simple graphs

Considering how difficult it usually is to represent connected simple graphs, the expression for $\mathcal{G}_o$ above is particularly elegant.

We can also define a wide range of generalized trees. Structures which satisfy

$$\mathcal{A}_{\mathcal{R}} = \mathcal{X} \cdot \mathcal{R}(\mathcal{A}_{\mathcal{R}})$$

for a given species $\mathcal{R}$ are called $\mathcal{R}$-enriched trees; the existence of these is guaranteed by the implicit species theorem.

$\mathcal{L}_+ = \mathcal{X} \cdot \mathcal{R}(\mathcal{L}_+)$	R = 1 + X, non-empy lists
$\mathcal{A}_{\mathcal{L}} = \mathcal{X} \cdot \mathcal{L}(\mathcal{A}_{\mathcal{L}})$	ordered rooted (planar) trees
$\mathcal{A} = \mathcal{X} \cdot \mathcal{E}(\mathcal{A})$	root and unordered subtrees
$\mathfrak{a}^\bullet = \mathcal{A}$	implicit definition of unrooted tree

Note that all R-enriched trees for any species R exists and is unique, by the implicit species theorem.

2.8 A Little Combinatorics

Species were originally created to abstract out various ideas from combinatorics. Not only is the connection interesting in its own right, we will also show how this can be useful in programming practice. There is a direct connection between species and exponential generating functions, mediated by a counting function.

Definition 2.11 Let $\mathcal{F}$ be a species, then we associate to it the following exponential generating function:

$$\mathcal{F}(x) = \sum_{n=0}^{\infty} |\mathcal{F}_n| \frac{x^n}{n!}$$

where $|\mathcal{F}_n|$ denotes the number of different $\mathcal{F}$-structures of size n.

It is important to note that the number of $\mathcal{F}$-structures on a set U only depends on $|U|$ and not on the elements of U itself. This is built-in to the definition of species, which are defined up to bijections of over U.

Theorem 2.12 *Let $\mathcal{F}$ and $\mathcal{G}$ be two species, then we have*

a) $(\mathcal{F} + \mathcal{G})(x) = \mathcal{F}(x) + \mathcal{G}(x)$ *b)* $(\mathcal{F} \cdot \mathcal{G})(x) = \mathcal{F}(x) \cdot \mathcal{G}(x)$

c) $(\mathcal{F} \circ \mathcal{G})(x) = \mathcal{F}(\mathcal{G}(x))$ *d)* $(\mathcal{F} \Box \mathcal{G})(x) = \mathcal{F}(x) \Box \mathcal{G}(x)$

e) $(\mathcal{F}')(x) = \frac{d}{dx}\mathcal{F}(x)$ *f)* $(\mathcal{F}^{\bullet})(x) = x\frac{d}{dx}\mathcal{F}(x)$

g) $(\mathcal{F} \times \mathcal{G})(x) = \mathcal{F}(x) \times \mathcal{G}(x)$

In the above theorem, we use two non-standard operations on series, namely the *Hadamard product* and a certain renumbering on series, defined by

$$\left(\sum_{n=0}^{\infty} f_n \frac{x^n}{n!}\right) \times \left(\sum_{n=0}^{\infty} g_n \frac{x^n}{n!}\right) = \sum_{n=0}^{\infty} f_n g_n \frac{x^n}{n!}$$

$$\left(\sum_{n=0}^{\infty} f_n \frac{x^n}{n!}\right) \Box \left(\sum_{n=0}^{\infty} g_n \frac{x^n}{n!}\right) = \sum_{n=0}^{\infty} f_{g_n} \frac{x^n}{n!}$$

From first principles, we have that $\mathbf{0}(x) = 0$, $\mathbf{1}(x) = 1$, and $\mathcal{X}(x) = x$. With some additional work, we can derive:

Proposition 2.13 *For the species as defined previously (or below), we have*

$\mathcal{L}(x) = \frac{1}{1-x}$ $\mathcal{C}(x) = -\ln(1-x)$ $\mathcal{E}(x) = \exp(x)$

$\wp(x) = \exp(2x)$ $\mathcal{A}_{\mathcal{L}}(x) = \frac{1-\sqrt{1-4x}}{2}$ $\mathcal{A}(x) = -\mathit{LambertW}(-x))$

$\mathcal{G}(x) = \sum_{n=0}^{\infty} 2^{\binom{n}{2}} \frac{x^n}{n!}$ $\mathcal{G}_o(x) = \sum_{n=0}^{\infty} 2^{n^2} \frac{x^n}{n!}$

$\mathcal{E}_{\mathit{even}}(x) = \sinh(x)$ $\mathcal{E}_{\mathit{odd}}(x) = \cosh(x)$

We use the *LambertW* function which is the principal branch of the solution to the functional equation $y(x)e^{y(x)} = x$ [9].

It is also interesting to note that $\mathcal{C}' \simeq \mathcal{L}$ is reflected in $\mathcal{C}'(x) = 1/(1-x) = \mathcal{L}(x)$. This is one example of the more general results that $\mathcal{F} \simeq \mathcal{G} \implies \mathcal{F}(x) = \mathcal{G}(x)$ (although the converse is false).

Classical analysis and combinatorics mix in interesting ways. For example we have that combinatorially, the species of sets splits as the disjoint union of the species of sets with an even number of elements with the specie of sets with an odd number of elements, or equationally $\mathcal{E} = \mathcal{E}_{even} + \mathcal{E}_{odd}$. In terms of generating functions, this reads as the classical identity $\exp(x) = \sinh(x) + \cosh(x)$.

There are two other formal power series related to the enumeration of $\mathcal{F}$-structures.

type generating series enumerates the unlabelled structures, or isomorphisms of structures

cycle index series a power series in an infinite number of variables which exactly captures the notion of isomorphism of structures.

These play an important rôle in the further development of the theory.

3 Programming with Species

This section shows how all of this can be adapted to programming practice. The reader who thinks that what we outline here seems to good to be true should reread the previous section (and some of the references from that section as well as the ones here). The correct impression should be that we are leveraging such powerful theory that the applications therefore seem natural.

3.1 Defining Type Constructors

The strategy we will employ to define type constructors from species is by now classical – we create a meta-language directly based on the semantics. Our guiding example here is Moggi's seminal work on monads [32]. Our meta-language definition is particularly simple since it is a conservative extension of Haskell's own notation.

More precisely, we add 2 new basic constructors, for set and cycle, and 5 new combinators for functorial composition, catesian product, pointing, differentiation and cardinality restriction. We keep the notation of the previous section for idealized/pretty-printed code. Note however that we are working "one level up": our + combinator does not form the sum of two *types*, but rather the sum of two *type constructors*. This operation is just the natural lifting of sum to the constructor level. For brevity, we will denote the kind `TC` by `TC`. *Gordon says:* Do you need the TC notation anymore? You don't use it anywhere else now.

Cardinality restriction is more delicate. A simple-minded treatment would introduce dependent types. But as is well-known, we can model $\mathbb{N}$ at the type-level without difficulty, and this is the solution we take here too. We can deal with more general range restrictions in a similar way, and omit the details as they are well-known.

Definition 3.1 Let us call a species representable via a system of functional equations as described above (minus functorial composition) a *representable* species[3].

[3] Note that these are frequently called *decomposable structures* in the combinatorics literature.

It is important to keep in mind that we are not defining the implementation of data structures, and that the notion of set and cycle represent abstract relationships between elements of a set. In a concrete program instance, we would be forced to model a set using a data-structure that allows for deterministic traversals of the elements. But, as is done in several languages for efficiency purposes (like `Maple`, and other languages for mathematics), it is entirely possible that the underlying implementation uses a session-dependent ordering, thus effectively mimicking the lack of predictable ordering from a programmer's point of view. It is thus important that this additional "storage model" structure should be distinguished from the true "conceptual" structure. We see this as a valuable opportunity, especially since this can enable further concurrency.

It is worth noting that, in the setting of species, McBride's aphorism that "*The derivative of a type is its type of one-holed structures*" is quite literally the definition (see section 2.3).

As we have seen before, the most general case of specifying type constructors via species is via (systems of) combinatorial equation(s), when such a solution exist. But this is already common practice, we just suggest that a few further operations be allowed, and give a semantic means to deal with the extra generality.

3.2 A Haskell implementation

We have an experimental implementation of these new types constructors and operators in Haskell. It provides:

- a representation for species,
- several storage models for data in a species,
- generic implementations of length (cardinality), equality, show, map and fold
- a build function that constructs families of structures based on a species definition.

Our main purpose is to allow experiments in generic programming, and more precisely to understand which techniques [20] generalize. As we see species as an extension of the type system, we are not terribly concerned about the efficiency of our current model implementation.

Our model stores a structure (`Structure`) as a pair (`Spec`,`Storage`) detailing both the type and a storage representation (containing actual elements). The storage representation uses regular data types but with an interface that hides the implementation (what we show below is a version without hiding so we can discuss certain details). In particular, sets and cycles cannot be implemented naturally in Haskell, at least without resorting to unsafe operations, and are currently stored as lists. We then provide the appropriate definition of equality: permutation for sets and equality up to rotation for cycles. It is important to remark that our sets only depend on `Eq` and not `Ord`, unlike Haskell's `Data.Set`.

```
data Structure a =   Stored Spec (Storage a) deriving (Show)
data Spec =
  Error              -- 0
| Empty              -- 1, empty set
| Single             -- X, singleton set
| Set                -- E, arbitrary size set
| Cycle              -- C, arbitrary size cycle
| FixSize Int Spec   -- fixed cardinality specification
| Pt  Spec           -- pointed species
| Prod Spec Spec     -- product (ordered)
| Sum  [Spec]        -- disjoint union
| Comp Spec Spec     -- composition of species
| FuncComp Spec Spec -- functorial composition
| RTree Spec Label   -- T_R = X.R(T_R)

data Storage a =
       StorNone | Stor1 a | Storn [a]
     | StorMark (Maybe Int) (Storage a)
     | StorProd (Storage a) (Storage a)
     | StorSum  Spec (Storage a)
     | StorComp (Storage (Storage a))
```

Our system currently does not support the definition of all implicit species by systems of equations, but we do support R-enriched trees. This provides a substantial set of recursively defined structures, including all regular data types but not nested types. A more general solution for implicitly defined species will be part of our future work.

Pointed and derivative structures have “special” elements in the data store (for derivatives this is more accurately called a hole, rather than an element). The position for this special element in the structure is identified with the storage mark; if the marker is over a list of elements (`Storn`), it contains the index of that element.

Cartesian product, the application of two structures over the same data set, is not a natural fit in Haskell. The easiest way to implement Cartesian products, is to use pointers (or indices), adding a level of indirection between the structure and the data. This is not particularly desirable. A better approach might be to implement a hybrid data model, with one species defining the underlying data structure and the other structure using indices or pointers over top of that structure. Certain Cartesian products may be amebable to a direct implementation, something we will explore in the future.

Most of the generic programming concepts extend easily to species. Being functors, mapping over species is straightforward; it does depend on the storage model but is generic in the species description.

```
instance Functor Structure where
    fmap f (Stored s xs) = Stored s (fmap f xs)
instance Functor Storage where  fmap = mapStor
mapStor :: (a->b) -> (Storage a) -> (Storage b)
mapStor f StorNone        = StorNone
mapStor f (StorMark _ x)  = mapStor f x
mapStor f (Stor1 x)       = Stor1 (f x)
mapStor f (Storn x)       = Storn (map f x)
mapStor f (StorProd x y)  = StorProd (mapStor f x) (mapStor f y)
mapStor f (StorSum s x)   = StorSum s (mapStor f x)
mapStor f (StorComp xy)   = StorComp (mapStor (mapStor f) xy)
```

Equality provides several interesting examples of how programming with species works. Equality for sets is up to permutation of elements, and for cycles is up to rotation. This is complicated by the fact that these may be sets/cycles of structures. Equality over compositions complicates the whole process; the equality of $\mathcal{F} \circ \mathcal{G}$ structures is managed by testing for the equality of $\mathcal{F}$ structures using the equality of $\mathcal{G}$structures as the boolean test. This is a generic strategy for lifting functions over compositions of structures.

```
instance (Eq a) => Eq (Structure a) where
  (==) (Stored s1 e1) (Stored s2 e2) | (s1 == s2) = eqstruct s1 (==) e1 e2
  (==) (Stored s1 e1) (Stored s2 e2) | (s1 /= s2) = False
eqstruct :: (Eq a) => Spec->(a->a->Bool)->Storage a->Storage a->Bool
eqstruct Error _ _ _ = True
eqstruct Empty _ _ _ = True
eqstruct Single eq (Stor1 x) (Stor1 y) = eq x y
eqstruct Single _ _ _ = False
eqstruct Set eq (Storn xs) (Storn ys) = isPermutationBy eq xs ys
eqstruct Set eq (Stor1 x) ys = eqstruct Set eq (Storn (x:[])) ys
eqstruct Set eq xs (Stor1 y) = eqstruct Set eq xs (Storn (y:[]))
eqstruct Cycle eq (Storn x) (Storn y) = isCycleBy eq x y
eqstruct (FixSize n s) eq x y = eqstruct s eq x y
eqstruct (Prod s1 s2) eq (StorProd x1 x2) (StorProd y1 y2) =
    (eqstruct s1 eq x1 y1) && (eqstruct s2 eq x2 y2)
eqstruct (Sum _) eq (StorSum sx ex) (StorSum sy ey) | (sx == sy)
                                                    = (eqstruct sx eq ex ey)
eqstruct (Sum _) eq (StorSum sx ex) (StorSum sy ey) | (sx /= sy)
                                                    = False
eqstruct (Pt s) eq ex ey = eqstruct s eq ex ey
eqstruct (Comp f g) eq (StorComp xs) (StorComp ys) =
    eqstruct f (eqstruct g eq) xs ys
eqstruct (FuncComp f g) eq xs ys = eqstruct Set eq xs ys
eqstruct (RTree r tr) eq (StorProd (Stor1 xr) rbx)
           (StorProd (Stor1 yr) rby) =
    (eq xr yr) && (eqstruct (Comp r (RTree r tr)) eq rbx rby)
eqstruct _ _ _ _ = False
```

3.3 Generic Programming

There are a large number of approaches to generic programming in Haskell, as described in [20], a very active area of research. Species fits very well into these strategies, with the new structures providing the opportunity for enhanced capabilities while being compatible with the existing systems.

The generic programming approaches described there generally recognize the notion of sum and product as operators on type constructors, and provide generic methods to adapt methods to these constructed types. For example, Generic Haskell allows functions to be defined over sum and product type constructor operators.

We show a Generic Haskell style translation of an encode function. To support species, the main additions are more bit patterns for headers and length encoding for sets and cycles. In this example, encode translates a regular dataype of Ints into a bit list[4] encapsulating the structure. For brevity, we name the subfunctions of `encode` with a leading `e` only.

$$\begin{aligned}
&\mathit{encode}\{\!|\mathbf{0}|\!\} && = \texttt{error "impossible"}\\
&\mathit{encode}\{\!|\mathbf{1}|\!\} && = [\,]\\
&\mathit{encode}\{\!|\mathcal{X}|\!\}\ i && = \mathit{eInt}\ i\\
&\mathit{encode}\{\!|\mathcal{E}|\!\}\ i && = \mathit{eSetStart}\ +\!\!+\ \texttt{foldr}\ \mathit{eInt}\ i\ +\!\!+\ \mathit{eSetEnd}\\
&\mathit{encode}\{\!|\mathcal{C}|\!\}\ i && = \mathit{eCycleStart}\ +\!\!+\ \texttt{foldr}\ \mathit{eInt}\ i\ +\!\!+\ \mathit{eCycleEnd}\\
&\mathit{encode}\{\!|\mathcal{F}_n|\!\}\ i && = \mathit{eCard}\ n\ +\!\!+\ \mathit{encode}\{\!|\mathcal{F}|\!\}\ i\\
&\mathit{encode}\{\!|\mathcal{F}^{\bullet}|\!\}\ (\mathcal{F}x) && = \mathit{ePt}\ +\!\!+\ \mathit{encode}\{\!|\mathcal{F}|\!\}\ x\\
&\mathit{encode}\{\!|\mathcal{F}+\mathcal{G}|\!\}\ (\mathcal{F}x) && = \mathit{encode}\{\!|\mathcal{F}|\!\}\ x\\
&\mathit{encode}\{\!|\mathcal{F}+\mathcal{G}|\!\}\ (\mathcal{G}x) && = \mathit{encode}\{\!|\mathcal{G}|\!\}\ x\\
&\mathit{encode}\{\!|\mathcal{F}\cdot\mathcal{G}|\!\}\ (x*y) && = \mathit{encode}\{\!|\mathcal{F}|\!\}\ x\ +\!\!+\ \mathit{encode}\{\!|\mathcal{G}|\!\}\ y\\
&\mathit{encode}\{\!|\mathcal{F}\circ\mathcal{G}|\!\}\ xys && = \mathit{fold}\{\!|\mathcal{F}|\!\}\ ((+\!\!+).(\mathit{encode}\{\!|\mathcal{G}|\!\}))\ xys\\
&\mathit{encode}\{\!|\mathcal{A}_{\mathcal{R}}|\!\}\ xs && = \mathit{encode}\{\!|\mathcal{X}\cdot(\mathcal{R}\circ\mathcal{A}_{\mathcal{R}})|\!\}\ xs\\
&\mathit{encode}\{\!|\mathcal{F}\times\mathcal{G}|\!\}\ x && = \mathit{eCart}\ +\!\!+\mathit{encode}\{\!|\mathcal{F}|\!\}\ x\ +\!\!+\ \mathit{encode}\{\!|\mathcal{G}|\!\}x
\end{aligned}$$

Some care must be taken with composition, as this means generically extending a function over a structure of structures. This requires a generic fold function, which

[4] the standard example uses Ints and Chars, our implementation is currently single-sorted

unwraps the first layer of species into a list, applying the encoding generated by the interior species. This is a standard pattern for generic functions over composed species.

Generically extending functions over the new species and operators is somewhat novel. The definition of equality of structures provides a good example of the issues here; two sets of items are equal if one is a permutation of the other, and two cycles are equal if there is a rotation of the first so that it is identical to the second. This is a substantial departure from all other generic programmings schemes as each data element in the underlying type is part of a singleton (in species terminology). This is not the case in species, where elements could be grouped in a set without an externally accessible ordering.

3.4 Testing and random generation

A Haskell programmers who is now addicted to QuickCheck [8] may wonder if this generalization will make their life more difficult. Pleasantly the answer is a resounding "no". Both the theory and practice of enumeration and random generation of samples from structures defined as species has been fully worked out.

A Calculus for the random generation of labelled combinatorial structures [16] covers the theory in detail. An even more thorough treatment is in the forthcoming book [15]. From the point of view of testing and random generation, we can summarize their findings as

Theorem 3.2 *Any representable specie has a random generation routine that uses precomputed tables of size $O(n)$ and achieves $O(n \log n)$ worst-case time complexity for generating a structure of size n. The tables take time $O(n^2)$ to compute.*

The generation algorithm is naturally staged: given a representation of a specie (and a size), it returns a specialized routine for the generation. This is especially well-suited to testing of algorithms over combinatorial structures, as the structures tend to be stable but testing of algorithms needs to happen on a regular basis.

The `combstruct` package in the computer algebra system Maple® offers such facilities [14]. For example, using this package we can define ordered rooted planar trees (see section 2.8) and draw a random size 7 structure via

```
> orpt := {AL = Prod(Z, Sequence(AL))}:
> draw([AL, orpt, 'unlabelled'], size=7);
```

where `AL` is the name of the structure, Z is an atom (i.e. $\mathcal{X}$), and `Sequence` is a synonym for `List`. It is also possible (via `allstructs`) to get all structures of a particular size.

3.5 Counting

The rules for producing a system of equations from a system of functional equations for a species are algorithmic. But much, much more is true. Getting any number of terms from the resulting generating functions is also algorithmic. For very large classes of equations, solving in closed form (when it exists) is a decidable problem. All examples in section 2.8 (except the ones using functorial composition) were automatically verified using `combstruct`. For the example of the previous section, it is as simple as

```
> gfsolve(orpt,'unlabelled',x);
```

which returns

$$\{Z(x) = x, AL(x) = \frac{1-\sqrt{1-4x}}{2}\}.$$

But one can do even more. The theory of asymptotic analysis of coefficients of holonomic functions (a class that covers most of the generating functions which appear as solutions) is very well developped [13,15] and largely automated [11,14,31].

3.6 Quantifying efficiency

We don't have to stop at counting datastructures. By associating cost functions in a natural way to functional algorithms over species [11,12], tools also exist [14,31] for the automatic computation of asymptotic expansions of both the worst-case and even *average case* cost functions for such functions.

We are expecially enamoured with a particular example in [12] which shows that for a class of expressions (defined as a specie), a particular algorithm for differentiation (using term rewriting) applied to a random expression of size n has complexity

$$\sqrt{\frac{\pi}{12}}n^{3/2} + \frac{5}{6}n + O(n^{1/2})$$

while a variant of the algorithm which uses sharing of terms has complexity

$$\frac{4}{3}n + \frac{1}{6} + O(\frac{1}{n}).$$

We are working on linking such tools into our current work.

3.7 Compiler optimization

Theorem 2.7 has a rather unusual application as a compiler optimization. Suppose we have some general algorithm over a specie $\mathcal{F}$ and we know the first few terms of its expansion $\mathcal{F} = \mathcal{F}_0 + \mathcal{F}_1 + \mathcal{F}_2 + \ldots$. We can then emit specialized code, using aggressive inlining, for those small cases. If the run-time representation of that structure also gives us $O(1)$ access to knowing whether it is of size[5] ≤ 2 or > 2, we can take full advantage of these specialized versions which are frequently much simpler. This is especially advantageous for recursive operations; although this only gives us a constant speed-up, that can nevertheless be significant. To us the important part is that this can be done automatically.

4 Advanced Species

We give a quick overview of the more advanced theory of species, as we have only covered a small part of the theory. This is by no means exhaustive, but rather only

[5] note that we only need to decide those predicates, we do not need to know the actual size

covers those areas where we feel that applications to (functional) programming are likely. The reader is again directed to [5] for more details.

4.1 Multisort and Weighted Species

The theory of species is easily extended into two different directions: adding weights and adding sorts.

Weightings allow the classification and analysis of species according to one or more descriptive parameter(s). Weights are given by a function $w : A \to \mathbb{A}$ where $\mathbb{A}$ is a formal power series ring (in an arbitrary number of variables) over an integral domain $\mathbb{K}$. Weights are frequently taken to be monomials in $\mathbb{A}$. We define the weight of a set A to be

$$|A|_w = \sum_{\alpha \in A} w(a)$$

From pairs (U, w) of a set U and a weight $w : U \to \mathbb{A}$, one can make a category by taking weight-preserving bijections as arrows. The endofunctors of this category are then called weighted species. The formal power series associated is the exponential power series $\mathcal{F}_w$ with coefficients in $\mathbb{A}$

$$\mathcal{F}_w(x) = \sum_{n \geq 0} \mid \mathcal{F}[n] \mid_w \frac{x^n}{n!} \tag{1}$$

As is usual in combinatorics, this lets us count certain properties of structures. For example, we can count binary trees which have at least on path of length l. Another example is the specie $\mathcal{S}_w$ of permutations weighted by the number of cycles, i.e. where $w(\sigma) = \alpha^{cycles(\sigma)}$ and α is a formal variable (so that $\mathbb{A} = \mathbb{Z}[\alpha]$). We then obtain that $\mathcal{S}_w(x) = \exp(-\alpha \log(1 - x)) = (1 - x)^{-\alpha}$.

Multisort species are defined as functors from $\mathbb{B}^k$ to $\mathbb{E}$, where the arrows of $\mathbb{B}^k$ are component-wise bijections. Each component is regarded as a different sort. One of the more interesting 2-sorted species is $Fun[U, V] = \{f \mid f : U \to V\}$ of functions from set U to set V. The associated generating function satisfies $Fun(X, Y) = \mathcal{E}(\mathcal{E}(X) \cdot Y)$. There are similar functional, as well as differential, equations for the 2-sorted species of injective, surjective and bijective functions (see exercise 1 in Section 2.4 of [5]).

Weighting can be extended to multisort species through obvious means, the details of which are also available in [5].

4.2 Virtual Species

The semi-ring of species ($\mathfrak{Spe}$) covers a large class of structures, but it does not provide a solution to all implicitly defined species. This semi-ring can be extended to a full ring, in much the same way that the semi-ring of natural numbers can be extended to the integers. The set of *virtual* species is defined as the quotient set $\mathfrak{Virt} = (\mathfrak{Spe} \times \mathfrak{Spe}) / \sim$ where $(\mathcal{F}, \mathcal{G}) \sim (\mathcal{H}, \mathcal{K}) \iff \mathcal{F} + \mathcal{H} \simeq \mathcal{G} + \mathcal{K}$. $\mathfrak{Virt}$

can be made into a commutative ring using

$$\begin{aligned}(\mathcal{F},\mathcal{G})+(\mathcal{H},\mathcal{K}) &=(\mathcal{F}+\mathcal{H},\mathcal{G}+\mathcal{K}) && \text{addition}\\ (\mathcal{F},\mathcal{G})\cdot(\mathcal{H},\mathcal{K}) &=(\mathcal{F}\cdot\mathcal{H}+\mathcal{G}\cdot\mathcal{K})-(\mathcal{F}\cdot\mathcal{K}+\mathcal{G}\cdot\mathcal{H}) && \text{multiplication}\end{aligned}$$

When $\mathcal{G}$ is a subspecie of $\mathcal{F}$, we can interpret $\mathcal{F}-\mathcal{G}$ straightforwardly as the structures of species $\mathcal{F}$ excluding those that are in $\mathcal{G}$. In other cases $\mathcal{F}-\mathcal{G}$ is called *strictly virtual*, and we must rely on formal rules to manipulate these virtual species.

One important result [40], is that every virtual species Φ can be written in a unique reduced form

$$\Phi = \Phi^{+} - \Phi^{-} \text{ where } \qquad \Phi^{+}\cap\Phi^{-}\subset\{\mathbf{0},\mathbf{1}\}$$

where we call Φ^{+} (resp. Φ^{-}) the positive (resp. negative) part of the species.

Various concepts, like multiplicative inverse (a.k.a division), the inverse for substitution and even logarithm can be given a meaning for virtual species. Additionally many combinatorial and functional equations that do not allow solutions within $\mathfrak{Spe}$ can be solved within the ring of virtual species. All operations on species generalize to virtual species.

Strictly virtual species do not have a clear interpretation as type constructors. However it seems likely that these would be very useful in representing intermediate constructions. As long as one only ever attempts to instantiate positive species, virtual species could provide a very rich language for modular construction of type constructors.

This ring is not without further surprises. For example, for fixed $n \geq 2$, let us consider the (virtual) specie $\Phi = n\mathcal{C}_n - \mathcal{X}^n$. We compute that $\Phi' = n\mathcal{X}^{n-1} - n\mathcal{X}^{n-1} = \mathbf{0}$. Thus the equation $\mathcal{F}' = \mathbf{0}$ has an infinite number of solutions which we would not normally consider "constant".

4.3 Integration

We have just shown that integration in species seems problematic. Since the derivative is already known to be an important operation, we would really want to understand integration (if it exists). There are really two answers.

In our current context, Rajan [34] shows that the derivative has both a left and right adjoint, which are rather different from each other.

Up to now, we have been using species based on the category $\mathbb{B}$. There is however a very similar theory of *linear species* based on the category $\mathbb{L}$ of totally ordered sets and increasing bijections. When we need to be precise, we refer to $\mathbb{B}$-species or $\mathbb{L}$-species as necessary.

$\mathbb{L}$-species are useful in two ways:

(i) In defining data structures in which there is some essential order on the underlying data elements, such as a binary tree with nodes decreasing in value towards the root,

(ii) and modeling the natural ordering assigned by the physical addressing of a concrete (von Neumann style) computational system.

$\mathbb{L}$-species allow two new operations:

$$\mathcal{F} * \mathcal{G} = \mathcal{F} \cdot_{\mathcal{O}} \mathcal{X} \cdot_{\mathcal{O}} \mathcal{G} \qquad \text{Convolution} \qquad (2)$$

$$(\int \mathcal{F})[L] = \mathcal{F}[L \backslash \{\min L\}] \qquad \text{Integral} \qquad (3)$$

where L is a linearly ordered set and $\min L$ is its least element. We indicate by $\cdot_{\mathcal{O}}$ the *ordered product*.

In $\mathbb{L}$-species, differentiation is defined by addition of a new *minimal* element, so that we can see that integration is literally the inverse of differentiation. Furthermore, this extra structure allows us to solve many different kinds of combinatorial differential equations, as well as functional equations.

5 Related Work

There are several different strands of research which are related. We single out four areas: work on generalized types, work on combinatorial structures, work on species, and work on generic programming. We review the highlights in the sections below.

5.1 Functors as Type Constructors

Using categorical models of data types has a long and fruitful history, but we will assume that the reader is aware of this. We will thus restrict ourselves to citing papers which we find representative of each approach. We found [17] a particularly enlightening introduction to this topic.

We hasten to point out that species, being of a combinatorial nature, do not easily lend themselves to modeling co-algebraic structures. This is not to say that there is not a relation, which Rutten [36] also conjectures exists, but this area does not seem to have been explored yet.

5.1.1 Containers

Containers [1,2,3] and Species are very similar, they mainly differ in their emphasis. Containers emphasize *shape*, while species put more emphasis on combinatorics and explicit examples. Species and quotient containers with finite shape are intimately related, and we would not be surprised if they turn out to be isomorphic. In other words, a certain part of our work could have just as easily been done using containers. However, we wanted to leverage the combinatorial tools and the extensive and explicit theory of species, especially as it is already close to programming practice. Doing the same via containers would have required us to first ensure that a lot (more) theory 'worked', which was not our goal. We are reasonably convinced that once a combinatorial interpretation of quotient containers is firmly established, our work could be recast in that light.

5.1.2 Non-regular Functors

Malcolm in [29] showed that initial algebras also exist for all regular functors. In first thought that regularity was somehow needed[6], but Bird et al in [7] followed by Johann and Ghani in [35] firmly put that to rest. This last paper shows that parametrised inductive data types where recursion is non-uniform still leads to initial algebras. This is done by considering the type to be the fixed point of a higher order functor over the category of functors over initial algebras. This reverses the roles of the fixed point operator and lamba operator to create inductive families of types instead of families of inductive types.

5.1.3 Bifunctors and Higher Order Operators

In [18], Gibbons defines a polymorphic type using bifunctors. A type is then obtained via the least fixed point of a section of this bifunctor with a type argument, viz

$TA = \mathrm{DATA}(A\oplus)$
$in_{TA} :: A \oplus\ TA \rightarrow\ TA$

where in_{TA} is the constructor function. The type constructor T can then be described as a functor, with its action on a function f given by

$$Tf = \mathrm{fold}_{TA}(in_{TB} \circ\ (f \oplus\ id)) = \mathrm{map}\,T\ f$$

and denoted as $\mathrm{map}\,T\ f$.

This approach makes it easy to define properties of folds and other paramorphic higher order operators. In order to make the fold and unfold interoperate, the work is presented in the category of pointed complete partial orders (pcpo) and strict continuous functions $\mathfrak{CPO}_\perp$, which requires all underlying sorts be extened with a $\perp$ denoting undefinedness, and certain strictness conditions. Many programs can be written as combinations of folds and unfolds; [18] includes several examples such as quicksort and merge sort.

This clearly seems to be related to 2-sorted species and the implicit species theorem, as we noted earlier. We intend to pursue the exact relation in more detail in future work.

5.1.4 Shapely Functors

C. Barry Jay introduced the notion of shapely functors in [23] as a way of separating a function's action on structure from its action on data values. A shapely functor F is defined categorically as a functor over an extensive category that preserves all (stable) pullbacks and is equipped with a copyable natural transformation that embeds a list of data elements into F.

Lists, vectors and matrices and their standard operations are defined in terms of shapely functors. In [25] and [24], he introduces shape polymorphism, the dual to data polymorphism, where an operation changes the shape but not the values of the data, and shows that shapely type constructors are closed under the construction of initial algebras. Lists, trees, graphs and all other "algebraic" types are shown to

[6] see [35] for details on this

be shapely; arrays and vectors are also shapely, so that class is larger than regular functors. He also addresses the issues of building a type system using shapely functors.

This theoretical work is then applied in the language FISh [26] in which data types operations have both shape and data components. The compiler checks shapes statically, and can optimize based on shapes.

One of the most notable features here is the inclusion of vectors and arrays. At any given size, these are also clearly species. What is interesting is how much of the shape is visible to the type system and the compiler, and can be used for optimization.

5.2 Combinatorial structures

The best guide to the vast litterature is undoubtedly the Flajolet and Sedgewick book [15], but Stanley's 2-volume set on enumerative combinatorics [37,38] is also a good source. Note that a fair bit of the work described here can be generalized to attribute grammars [31].

As well as `combstruct` in Maple, there is also `MuPAD-Combinat` [21] and `Aldor-Combinat` [19]. The latter is noteworthy in two ways: it is implemented in a very strongly typed language, and it adheres very closely to the theory of species. There was also an earlier effort called Darwin [6] which does not seem to have survived to this day.

5.3 Species

There have been many generalizations of species to cover many different aspects of combinatorics – but an enumeration of these would take us too far afield. Recently, Fiore et al have begun to explore the generalisation of species of structures between small categories and the corresponding generalizsed analytic functors (between presheaf categories). This research is deeply categorical, but as part of [10], they have identified that a Kleisli-bicategory of generalised species of structures is cartesian closed, and therefore supports operations for abstraction and evaluation, in addition to the standard projection and pairing. This clearly makes this work relevant to the use of species to functional programming languages, but the exact decoding from category theory to practical implementation matters have not yet been done.

One should also mention the link with quantum physics through Baez and Dolan's *stuff types* [4] and expanded by Morton [33], which are also a generalization of species.

5.4 Datatype Generic Programming

Datatype generic programming is one name given to the study of formal (and informal) strategies to define functions which are naturally induced by the natural shape of a data structure.

For Haskell alone [20] takes 75 pages for a thorough comparison of the various approaches. We take it for granted that this is an important and fruitful methodology. Thus we must prove that it is possible to lift these ideas to the setting of species as well. We are evaluating all of the approaches as enumerated in [20] to see which one will generalize most elegantly.

In theory, the refinement to the implicit species theorem gives us exactly what we need: the existence of the appropriate initial algebra. This in turn gives us folds. Unfortunately, this is insufficient, as implementing a fold over a set or a cycle is considerably more difficult than over types obtained from a regular functor.

This is not an easy task, but Johann and Ghani [35] have shown how to describe folds over nested datatypes. They provide both fold and build definitions for nested data types arising from all rank 2 functors, and also provide generalized fold / build fusion rules. These are given both as theory and in a Haskell implementation. The extension of fusion laws to non-polynomial functors is a valuable guide in the evaluation of fusion laws to analytic functors.

We should also mention Zippers. Originally conceived as a means to store a data type with a distinguished element (i.e. a cursor), as well as a means to reach the neighbours of this element. Huet [22] describes zippers for trees. Over time, the link with derivatives was formalized. However, part of the original notion seems to have been lost: that of a zipper as a means of (local) traversal. For regular functors, it is relatively simple to reconstruct this information. However, it is not at all clear if there is such a clear notion for "set", for example.

6 Conclusion and Future Work

Species are a straightforward and powerful generalization of the theory of type constructors as currently exists in functional programming languages. It provides additional constructors and combinators for the construction of useful *abstract* types. The theory is already extrememly rich, and actually provides for features not yet common in current implementations of programming languages, such as random generation and enumeration (for testing), generation of counting functions (as equations, closed-form or in asymptotic form). It also enables some further functionality, like automatic worst and average case analysis of algorithms, especially those algorithms which "follow the structure" of the types.

It is not without some trepidation that we set out to write this paper, hoping to do for species what Wadler did for monads in functional programming.

To us, one of the most interesting parts of the theory of combinatorial species is its deep connections with generating functions, and thus analysis. It is remarkable how many theorems from classical analysis (about analytic functions) can be lifted to this setting. As Fiore et al note [10], "Analytic functors can be regarded as structural counterparts to exponential generating functions, and provide an equivalent view of species of structures as Taylor series." We are working on furthering this dictionary by linking concepts in analysis as directly as possible to idioms in functional programming. In particular, the connection with Huet's Zipper [22] is deeper than just being able to compute derivatives of type constructors. Some traversal strategies seem to have interpretations as integrals and integral transforms that we wish to make more precise.

Our implementation was strictly intended to explore the viability of species as type constructors, and to demonstrate that the main generic programming techniques can be applied to analytic functors. We intend to explore this link in further detail to see what new opportunities this affords. In particular, using one of the generic programming frameworks, we would like to experiment with extending Haskell's syntax and type system to create a deep embedding of species as type constructors.

Another area for investigation is to extend QuickCheck to be able to deal with these new constructors. The automation afforded to us by the current state of the theory should allow us to "derive" automated testing routines. If we can push that in the area of combinatorics as well, perhaps by first linking-in to Maple's `combstruct`, that would be interesting too. Eventually, we would like to include automated analysis of algorithms.

The attentive reader might have noticed the almost complete absence of folds from this paper. In theory, there is no issue with fold. In practice however, implementations of fold have to deal with ordering of computations, which is a rather thorny issue in the presence of sets and cycles. We have thought long and hard about this issue, and hope to report on our progress in a subsequent publication.

References

[1] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In Andrew D. Gordon, editor, *Foundations of Software Science and Computational Structures*, volume 2620 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[2] Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Constructing polymorphic programs with quotient types. *Mathematics of Program Construction*, pages 2–15, 2004.

[3] Michael Abott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. ∂ for data. *Fundamentae Informatica*, 65(1,2):1 – 28, March 2005. Special Issue on Typed Lambda Calculi and Applications 2003.

[4] John Baez and James Dolan. *From finite sets to Feynman diagrams*, pages 177–195. Springer-Verlag, Berlin, 2001.

[5] Francois Bergereon, Gilbert Labelle, and Pierre Laroux. *Combinatorial Species and Tree-Like Structures*, volume 67 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1998.

[6] F. Bergeron and C. Pichet. Darwin, a system designed for the study of enumerative combinatorics. In *EUROCAL '85*, volume 204 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1985.

[7] Richard Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 3:1–100, 1999.

[8] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ICFP 2000*, 2000.

[9] R.M. Corless, G.H. Gonnet, D.E.G. Hare, D.J. Jeffrey, and D.E. Knuth. On the Lambert W function. *Advances in Computational Mathematics*, 5:329–359, 1996.

[10] M. Fiore, N. Gambino, M. Hyland, and G. Winskel. The cartesian closed bicategory of generalised species of structures. *J. London Math. Soc.*, 77(1):203–220, 2008.

[11] P. Flajolet, B. Salvy, and P. Zimmermann. Lambda-Upsilon-Omega: The 1989 Cookbook. Research Report 1073, INRIA, August 1989. 116 pages.

[12] P. Flajolet, B. Salvy, and P. Zimmermann. Automatic average-case analysis of algorithms. *Theoretical Computer Science A*, 79(1):37–109, February 1991.

[13] Philippe Flajolet and Andrew M. Odlyzko. Singularity analysis of generating functions. *SIAM J. Discrete Math.*, 3(2):216–240, 1990.

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

[15] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, January 2008. Preliminary version of a book to be published by Cambridge University Press, 800p.+x, available electronically from Philippe Flajolet's home page.

[16] Philippe Flajolet, Paul Zimmermann, and Bernard Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theor. Comput. Sci.*, 132(2):1–35, 1994.

[17] Jeremy Gibbons. *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, chapter 5, pages 149–201. Number 2297 in LNCS. Springer, 2002.

[18] Jeremy Gibbons. Design patterns as higher-order datatype-generic programs. In *ACM SIGPLAN Workshop on Generic Programming*, 2006. part of ICPF 2006.

[19] Ralf Hemmecke and Martin Rubey. Aldor-combinat: An implementation of combinatorial species, 2006. Available at http://www.risc.uni-linz.ac.at/people/hemmecke/aldor/combinat/.

[20] Ralf Hinze, Johan Jeuring, and Andres Löh. *Comparing Approaches to Generic Programming in Haskell*, pages 72–149. Number 4719 in LNCS. Springer Verlag, 2007.

[21] F. Hivert and N.M Thiéry. MuPAD-Combinat, an open-source package for research in algebraic combinatorics. *Séminaire Lotharingien de Combinatoire*, 51, 2004. 70 pp.

[22] Gerard Huet. Functional pearl: The zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997.

[23] C. Barry Jay. Matrices, monads and the fast fourier transform. Technical Report UTS-SOCS-93.13, University of Technology, Sydney, August 1993.

[24] C. Barry Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.

[25] C.B. Jay and J.R.B. Cockett. Shapely types and shape polymorphism. In D. Sannella, editor, *Programming Languages and Systems - ESOP '94: 5th European Symposium on Programming, Edinburgh, U.K., April 1994, Proceedings*, volume 788 of *Lecture Notes in Computer Science*, pages 302–316. Springer Verlag, 1994.

[26] C.B. Jay and P.A. Steckler. The functional imperative: shape! In Chris Hankin, editor, *Programming languages and systems: 7th European Symposium on Programming, ESOP'98 Held as part of the joint european conferences on theory and practice of software, ETAPS'98 Lisbon, Portugal, March/April 1998*, volume 1381 of *LNCS*, pages 139–53. Springer Verlag, 1998.

[27] André Joyal. Une théorie combinatoire des séries formelles. *Advances in Mathematics*, 42:1–82, 1981.

[28] André Joyal. Foncteurs analytiques et especes de structures. *Combinatoire Enumerative, Springer Lecture Notes in Mathematics*, 1234:125–159, 1986.

[29] Grant Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Rijksuniversiteit Groningen, September 1990.

[30] Matías Menni. Combinatorial functional and differential equations applied to differential posets. *Discrete Mathematics*, 308(10):1864–1888, May 2008.

[31] Marni Mishna. Attribute grammars and automatic complexity analysis. *Advances in Applied Mathematics*, 30(1):189–207, 2003.

[32] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[33] Jeffrey Morton. Categorified algebra and quantum mechanics. *Theory and Applications of Categories*, 16(29):785–854, 2006.

[34] Dayanand S. Rajan. The adjoints to the derivative functor on species. *J. Comb. Theory, Ser. A*, 62(1):93–106, 1993.

[35] S. Ronchi Della Rocca, editor. *Initial Algebra Semantics is Enough!*, volume 4583. Typed Lambda Calculus and Applications, Springer-Verlag, 2007.

[36] J. J. M. M. Rutten. Behavioural differential equations: a coinductive calculus of streams, automata, and power series. *Theor. Comput. Sci.*, 308(1-3):1–53, 2003.

[37] Richard P. Stanley. *Enumerative combinatorics. Vol. 1*, volume 49 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, 1997. With a foreword by Gian-Carlo Rota, Corrected reprint of the 1986 original.

[38] Richard P. Stanley. *Enumerative combinatorics. Vol. 2*, volume 62 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, 1999. With a foreword by Gian-Carlo Rota and appendix 1 by Sergey Fomin.

[39] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In Steve Munchnik, editor, *Proceedings, 14th Symposium on Principles of Programming Languages*, pages 307–312. Association for Computing Machinery, 1987.

[40] Y N Yeh. *The Calculus of Virtual Species and K-Species*. Number 1234 in Lecture Notes in Mathematics. Springer Verlag, 1986.

A Tables of species (from [5])

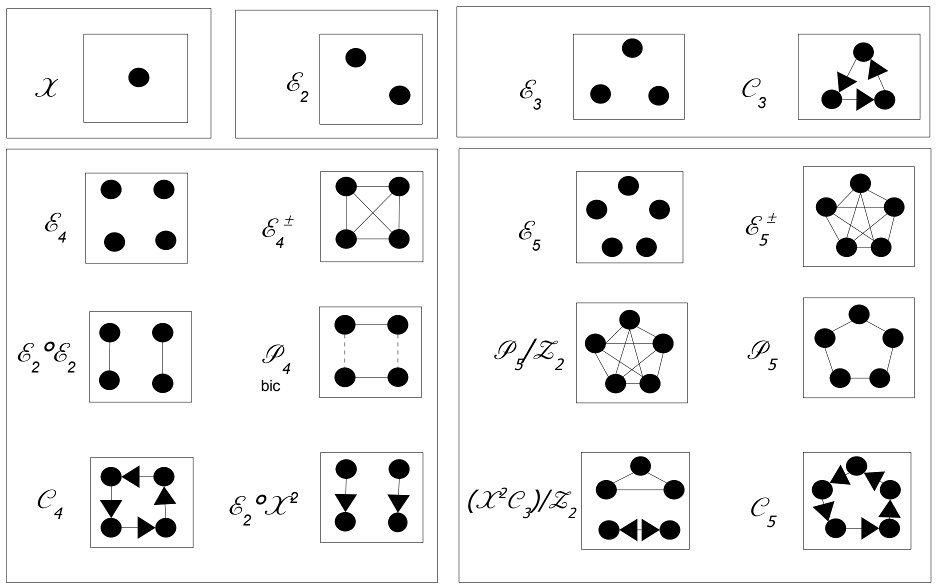

Fig. A.1. Atomic species for sets of cardinality $n \leq 5$.

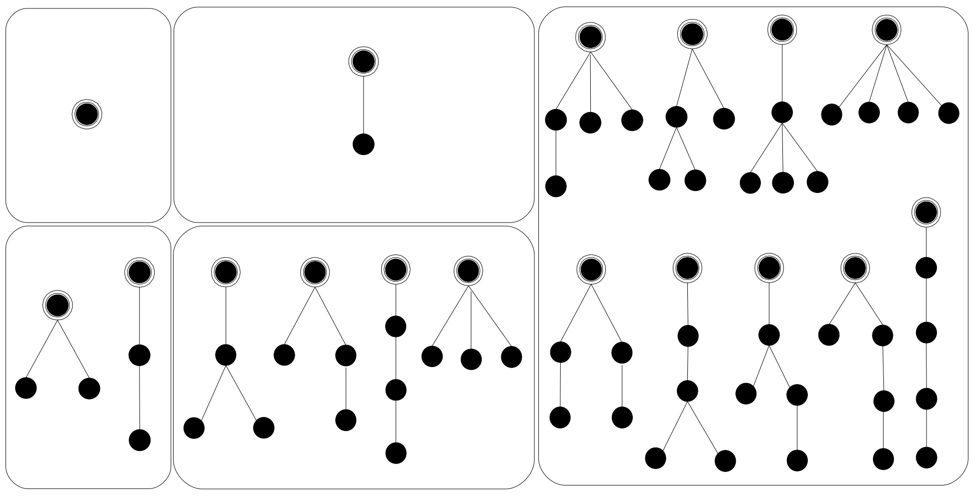

Fig. A.2. Unlabelled rooted trees of cardinality $n \leq 5$.