

Random testing—and beyond! with combinatorial species

Brent Yorgey
University of Pennsylvania

University of Kansas
November 24, 2009



Outline

Introduction/Motivation

Regular species

- As types

- As generating functions

Generating and selecting shapes

Further research

The theory of combinatorial species gives us a flexible, effective framework for thinking about data types and generic programming.

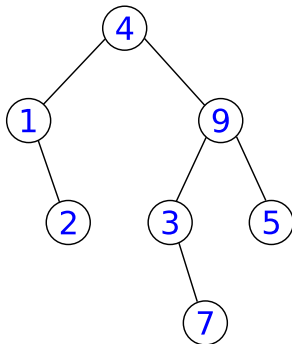
- ▶ Case study: test generation
- ▶ Hints at other applications

- ▶ J. Carette and G. Uszkay, Species: making analytic functors practical for functional programming

Testing binary trees

Consider the following Haskell type of polymorphic binary trees:

```
data Tree  $\alpha$  = Empty  
      | Node  $\alpha$  (Tree  $\alpha$ ) (Tree  $\alpha$ )
```



Testing binary trees

We can write functions that work on binary trees:

— *Test whether any element of a tree satisfies a predicate.*

$\text{anyElt} :: (a \rightarrow \text{Bool}) \rightarrow \text{Tree } a \rightarrow \text{Bool}$

$\text{anyElt } p \text{ Empty} = \text{False}$

$\text{anyElt } p (\text{Node } a \text{ l } r) = p \ a \ \vee \ \text{anyElt } p \ \text{l} \ \vee \ \text{anyElt } p \ \text{r}$

— *Compute the product of all the elements in a tree.*

$\text{treeProd} :: \text{Num } a \Rightarrow \text{Tree } a \rightarrow a$

$\text{treeProd Empty} = 1$

$\text{treeProd } (\text{Node } a \text{ l } r) = a * \text{treeProd l}$

Testing binary trees

We can write functions that work on binary trees:

— *Test whether any element of a tree satisfies a predicate.*

$\text{anyElt} :: (a \rightarrow \text{Bool}) \rightarrow \text{Tree } a \rightarrow \text{Bool}$

$\text{anyElt } p \text{ Empty} = \text{False}$

$\text{anyElt } p (\text{Node } a \text{ l } r) = p \ a \ \vee \ \text{anyElt } p \ \text{l} \ \vee \ \text{anyElt } p \ \text{r}$

— *Compute the product of all the elements in a tree.*

$\text{treeProd} :: \text{Num } a \Rightarrow \text{Tree } a \rightarrow a$

$\text{treeProd Empty} = 1$

$\text{treeProd (Node } a \text{ l } r) = a * \text{treeProd l}$

oops. . .

Testing binary trees

Testing with QuickCheck:

— *All trees containing zero should have a product of zero.*

`prop_prodZero :: Tree Int → Property`

`prop_prodZero t = anyElt ($\equiv 0$) t \implies treeProd t $\equiv 0$`

Testing binary trees

Testing with QuickCheck:

— *All trees containing zero should have a product of zero.*

`prop_prodZero :: Tree Int → Property`

`prop_prodZero t = anyElt ($\equiv 0$) t \implies treeProd t $\equiv 0$`

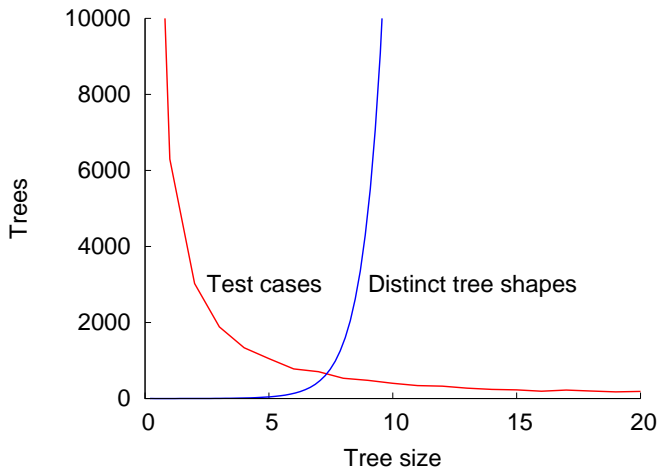
How to generate random instances of `Tree`? QuickCheck can't do it automatically...

Generating trees: state of the art!

instance Arbitrary $\alpha \Rightarrow$ Arbitrary (Tree α) **where**
 arbitrary = sized tree
 tree 0 = return Empty
 tree n = oneof [return Empty,
 liftM3 Node arbitrary subtree subtree]
 where subtree = tree $\lfloor n/2 \rfloor$

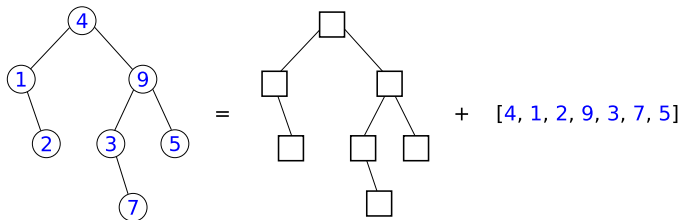
The problem

This gives a horrible distribution of test cases!



Shapes

A shape is the application of a type constructor to a distinguished unit type \square (a “hole”).



Data structure = shape + data.

- ▶ M. Abbott, T. Altenkirch, and N. Ghani, Categories of Containers, FoSSaCS 2003.

Testing by shape

By parametricity, for polymorphic uses we only need to test each shape exactly once.

- ▶ J. Voigtländer: Much Ado about Two: a pearl on parallel prefix computation. POPL 2008.
- ▶ J.-P. Bernardy, P. Jansson, K. Claessen: Testing Polymorphic Properties.

For monomorphic uses we can still decouple shape generation from content generation.

Goals

We'd like to...

- ▶ Test all shapes up to a given size (like SmallCheck)
- ▶ Then randomly generate larger instances (like QuickCheck)

Goals

We'd like to...

- ▶ Test all shapes up to a given size (like SmallCheck)
- ▶ Then randomly generate larger instances (like QuickCheck)

So we need...

- ▶ A way to generate all shapes of a given size
- ▶ A way to randomly select a shape of a given size

Goals

We'd like to...

- ▶ Test all shapes up to a given size (like SmallCheck)
- ▶ Then randomly generate larger instances (like QuickCheck)

So we need...

- ▶ A way to generate all shapes of a given size
- ▶ A way to randomly select a shape of a given size
- ▶ ... Generically for any data type!

Outline

Introduction/Motivation

Regular species

As types

As generating functions

Generating and selecting shapes

Further research

Species algebra, take 1

$$\begin{aligned} S ::= & 0 \\ & | 1 \\ & | X \\ & | S + S \\ & | S \cdot S \end{aligned}$$

Species algebra, take 1

$$\begin{aligned} S ::= & 0 \\ & | 1 \\ & | X \\ & | S + S \\ & | S \cdot S \end{aligned}$$

Intuition?

Species as polymorphic types

$$0[\alpha] = \text{Void}$$

$$1[\alpha] = \text{Unit}$$

$$X[\alpha] = \alpha$$

$$(S + T)[\alpha] = S[\alpha] + T[\alpha]$$

$$(S \cdot T)[\alpha] = S[\alpha] \times T[\alpha]$$

Species as polymorphic types

$$0[\alpha] = \text{Void}$$

$$1[\alpha] = \text{Unit}$$

$$X[\alpha] = \alpha$$

$$(S + T)[\alpha] = S[\alpha] + T[\alpha]$$

$$(S \cdot T)[\alpha] = S[\alpha] \times T[\alpha]$$

type Foo $\alpha = \text{Either (Maybe } (\alpha, \alpha)) \text{ Bool}$

$$\text{Foo} = (1 + X \cdot X) + (1 + 1)$$

$$= 3 + X^2$$

Species as polymorphic types

$$0[\alpha] = \text{Void}$$

$$1[\alpha] = \text{Unit}$$

$$X[\alpha] = \alpha$$

$$(S + T)[\alpha] = S[\alpha] + T[\alpha]$$

$$(S \cdot T)[\alpha] = S[\alpha] \times T[\alpha]$$

type Foo $\alpha = \text{Either (Maybe } (\alpha, \alpha)) \text{ Bool}$

$$\text{Foo} = (1 + X \cdot X) + (1 + 1)$$

$$= 3 + X^2$$

Recursive types...?

Species algebra, take 2

$$\begin{aligned} S ::= & 0 \\ & | 1 \\ & | X \\ & | S + S \\ & | S \cdot S \\ & | x \\ & | \mu x. S \end{aligned}$$

Species algebra, take 2

$$\begin{aligned} S ::= & 0 \\ & | 1 \\ & | X \\ & | S + S \\ & | S \cdot S \\ & | x \\ & | \mu x. S \end{aligned}$$

data Tree α = Empty | Node α (Tree α) (Tree α)

$$\text{Tree} = \mu t. 1 + X \cdot t \cdot t$$

Species as (ordinary) generating functions

For S a species, $\widetilde{S}(x) \in \mathbb{Z}[[x]]$:

$$\widetilde{0}(x) = 0$$

$$\widetilde{1}(x) = 1$$

$$\widetilde{X}(x) = x$$

$$\widetilde{S + T}(x) = \widetilde{S}(x) + \widetilde{T}(x)$$

$$\widetilde{S \cdot T}(x) = \widetilde{S}(x) \widetilde{T}(x)$$

$$\widetilde{\mu x. S}(x) = S[x \mapsto \mu x. S](x)$$

Species as (ordinary) generating functions

For S a species, $\tilde{S}(x) \in \mathbb{Z}[[x]]$:

$$\tilde{0}(x) = 0$$

$$\tilde{1}(x) = 1$$

$$\tilde{X}(x) = x$$

$$\widetilde{S + T}(x) = \tilde{S}(x) + \tilde{T}(x)$$

$$\widetilde{S \cdot T}(x) = \tilde{S}(x)\tilde{T}(x)$$

$$\widetilde{\mu x.S}(x) = S[x \mapsto \mu x.S](x)$$

$$\widetilde{(3 + X^2)}(x) = 3 + x^2$$

$$\widetilde{\text{Tree}}(x) = 1 + x + 2x^2 + 5x^3 + 14x^4 + \dots$$

The punchline

$$\tilde{S}(x) = s_0 + s_1x + s_2x^2 + s_3x^3 + \cdots = \sum_{n \geq 0} s_n x^n$$

The punchline

$$\tilde{S}(x) = s_0 + s_1x + s_2x^2 + s_3x^3 + \cdots = \sum_{n \geq 0} s_n x^n$$

s_n is the number of distinct S -shapes with n holes!

Combinatorial intuition

- ▶ $\tilde{0}(x) = 0$: no values of type $0[\square] = \text{Void}$.

Combinatorial intuition

- ▶ $\tilde{0}(x) = 0$: no values of type $0[\square] = \text{Void}$.
- ▶ $\tilde{1}(x) = 1$: one value of type $1[\square] = \text{Unit}$, with no holes.

Combinatorial intuition

- ▶ $\tilde{0}(x) = 0$: no values of type $0[\square] = \text{Void}$.
- ▶ $\tilde{1}(x) = 1$: one value of type $1[\square] = \text{Unit}$, with no holes.
- ▶ $\tilde{X}(x) = x$: one value of type $X[\square] = \square$, with one hole.

Combinatorial intuition

$$\begin{aligned}\widetilde{S + T}(x) &= \widetilde{S}(x) + \widetilde{T}(x) \\ &= \sum_{n \geq 0} s_n x^n + \sum_{n \geq 0} t_n x^n \\ &= \sum_{n \geq 0} (s_n + t_n) x^n\end{aligned}$$

An $(S + T)$ -shape with n holes is either an S -shape with n holes or a T -shape with n holes.

Combinatorial intuition

$$\begin{aligned}\widetilde{S \cdot T}(x) &= \widetilde{S}(x)\widetilde{T}(x) \\ &= \left(\sum_{n \geq 0} s_n x^n \right) \left(\sum_{n \geq 0} t_n x^n \right) \\ &= \sum_{n \geq 0} \left(\sum_{k=0}^n s_k t_{n-k} \right) x^n\end{aligned}$$

An $(S \cdot T)$ -shape with n holes is an S -shape with k holes paired with a T -shape with $n - k$ holes.

Outline

Introduction/Motivation

Regular species

As types

As generating functions

Generating and selecting shapes

Further research

Generating tree shapes

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

$$t_0 = 1$$

$$t_n = \sum_{k=0}^{n-1} t_k t_{n-1-k}$$

```
treeShapes :: Int → [Tree □]
treeShapes 0 = [Empty]
treeShapes n = [Node □ | r | k ← [0..n-1],
                      l ← treeShapes k,
                      r ← treeShapes (n-k-1)
                ]
```

Selecting shapes

For uniform random generation of shapes, it's enough to efficiently compute a bijection

$$S_n^! : \{0, \dots, s_n - 1\} \rightarrow S_n[\square].$$

We can again use intuition from generating functions to guide us.

Selecting shapes

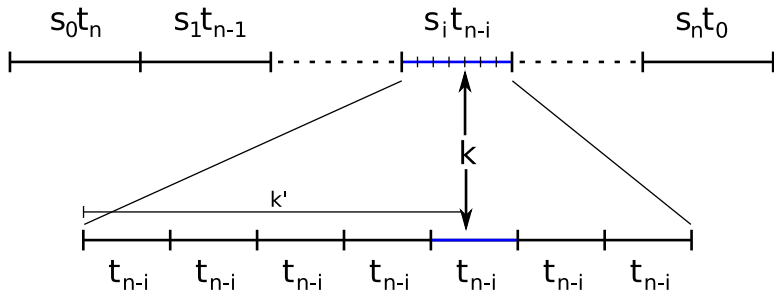
Define $S_n^! : \{0 \dots s_n - 1\} \leftrightarrow S_n[\square]$ by

$$1_0^! = 1$$

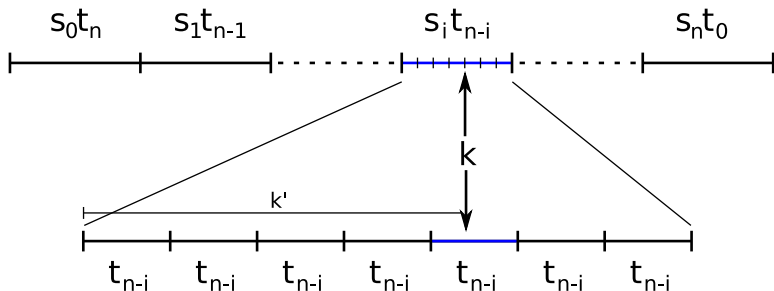
$$X_1^! = \square$$

$$(S + T)_n^!(k) = \begin{cases} S_n^!(k) & k < s_n \\ T_n^!(k - s_n) & k \geq s_n \end{cases}$$

$$(S \cdot T)_n^!(k) = (S_i^!(\lfloor k'/t_{n-i} \rfloor), T_{n-i}^!(k' \bmod t_{n-i}))$$

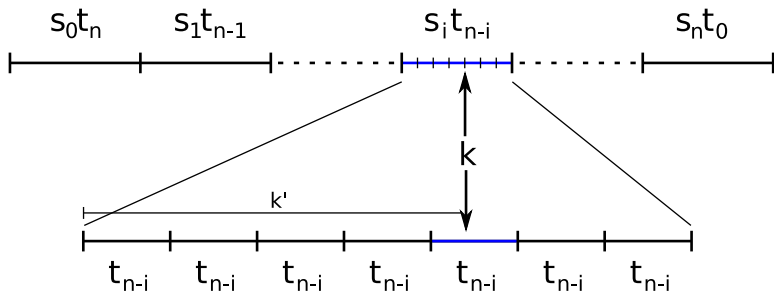


$$(S \cdot T)_n^!(k) = (S_i^!(\lfloor k'/t_{n-i} \rfloor), T_{n-i}^!(k' \bmod t_{n-i}))$$



Requires knowing s_n, t_n, \dots

$$(S \cdot T)_n^!(k) = (S_i^!(\lfloor k'/t_{n-i} \rfloor), T_{n-i}^!(k' \bmod t_{n-i}))$$



Requires knowing s_n, t_n, \dots which can be precomputed using generating functions.

Outline

Introduction/Motivation

Regular species

As types

As generating functions

Generating and selecting shapes

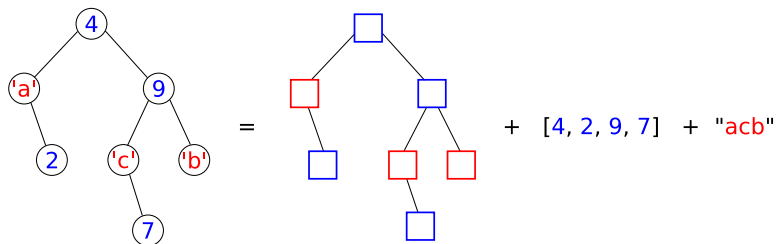
Further research

Multiply-polymorphic types?

Multi-sorted species.

Just add X_1, X_2, X_3, \dots and $\square_1, \square_2, \square_3, \dots$

data Tree2 a b = Empty
| Node2 (Either a b) (Tree2 a b) (Tree2 a b)



Monomorphic types?

- ▶ Add K_τ for each base type τ , with $K_\tau[\alpha] = \tau$.
- ▶ Count constructors (or more generally, assign each constructor a “weight”).
- ▶ P. Flajolet, P. Zimmerman, and B. Van Cutsem, A calculus for the random generation of labelled combinatorial structures, Theor. Comput. Sci. 132, no. 1-2, pp. 1-35.

Composition

Consider the type of rose/plane trees:

```
data RTree a = Empty  
            | RNode a [RTree a]
```

As a species, $\text{RTree} = \dots?$

Composition

Consider the type of rose/plane trees:

```
data RTree a = Empty
             | RNode a [RTree a]
```

As a species, $\text{RTree} = \dots?$

$\mu r.1 + X \cdot (\mu l.1 + r \cdot l)?$ Ugh!

Composition

Consider the type of rose/plane trees:

```
data RTree a = Empty
             | RNode a [RTree a]
```

As a species, $\text{RTree} = \dots?$

$$\mu r.1 + X \cdot (\mu l.1 + r \cdot l)? \text{ Ugh!}$$

$$\mu r.1 + X \cdot (L \circ r)$$

$$L = \mu l.1 + X \cdot l$$

Composition

For regular species,

$$\widetilde{S \circ T}(x) = \widetilde{S}(\widetilde{T}(x))$$

and we can make good combinatorial sense out of this! So we can generate and select shapes with composition, too.

Other species?

Extend from regular to analytic species.

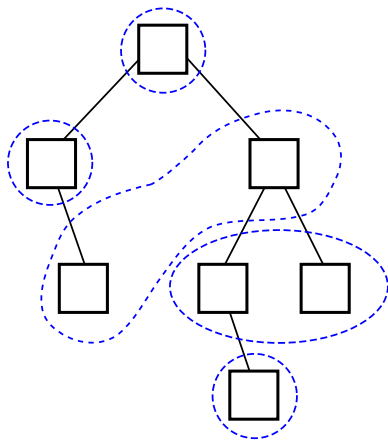
- ▶ Bags (E), cycles (C), ...
- ▶ Cartesian product, functor composition, differentiation...

$$T_G = 1 + X \cdot (E \circ T_G)$$

We can do generation and selection for these too. What would a programming language look like that allowed such data structures?

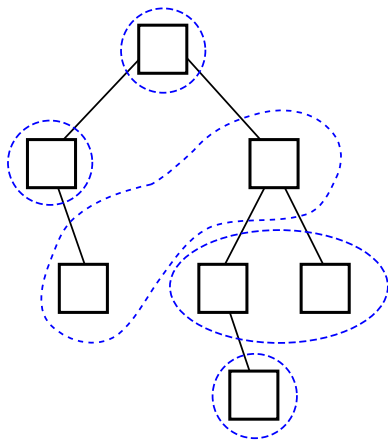
Constrained polymorphism?

Eq $\alpha \Rightarrow S \alpha$ corresponds to...



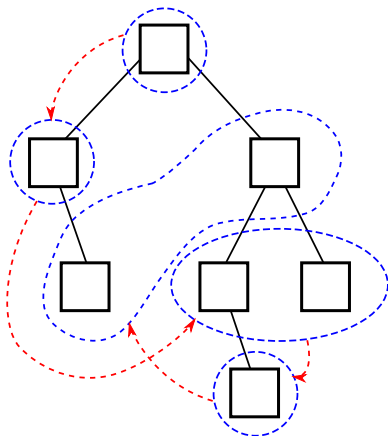
Constrained polymorphism?

Eq $\alpha \Rightarrow S \alpha$ corresponds to... $S \times Par = S \times (E \circ E_+)$



Constrained polymorphism?

Ord $\alpha \Rightarrow S \alpha$ corresponds to... $S \times Bal = S \times (L \circ E_+)$



Want to know more?

More about combinatorial species:

- ▶ <http://byorgey.wordpress.com/2009/07/24/introducing-math-combinatorics-species/>
- ▶ Bergeron, Labelle, and Leroux, Combinatorial Species and Tree-Like Structures.