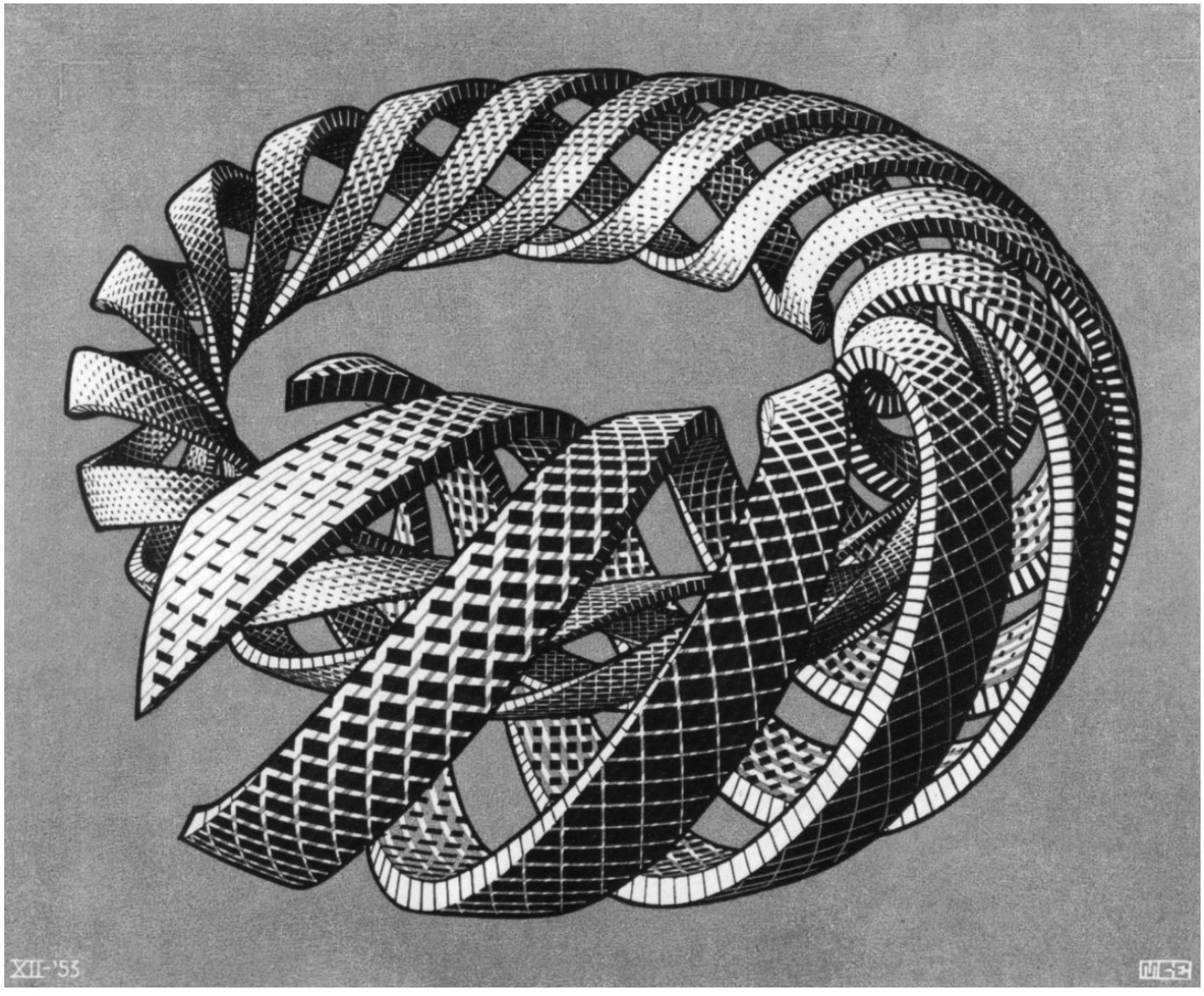
Coinductive Stream Calculus in Haskell

Joost Winter

May 27, 2020



Spirals (M.C. Escher, 1955)

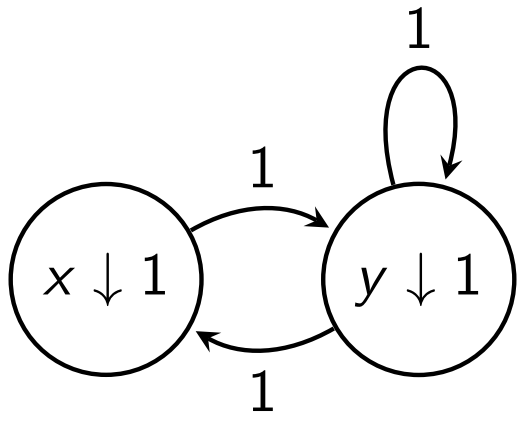
Outline

- We will explore connections between the (traditional and coinductive) theory of weighted automata, and streams as they can be coinductively defined in Haskell.
- This talk will be roughly half theory and half practice.
- Coinductive stream calculus in Haskell: pioneered by Douglas McIlroy and Ralf Hinze, making use of the principle of *lazy evaluation*.

Simple streams in Haskell

```
x = 1 : 2 : 3 : x -- a periodic stream
y = 0 : x -- an eventually periodic stream
```

Automata-theoretic setting



- As common in coinductive approaches to automata theory, we do not use specific input vectors. We will moreover fix $\mathbb{Q}$ as the underlying output structure, and fix a singleton input alphabet.
- Matrix view: $E = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$, $T = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$
- Behavioural differential equation view:

$$\begin{array}{rclcrcl} o(x) & = & 1 & & x' & = & y \\ o(y) & = & 1 & & y' & = & x + y \end{array}$$

Simple streams

Some useful definitions:

- A stream is defined as an element of the space $\mathbb{Q}^\omega$.
- The derivative σ' of a stream σ is defined as its tail.
- We call a set of streams $S \subseteq \mathbb{Q}^\omega$ *stable* whenever it is closed under derivative

Simple streams

Some useful definitions:

- A stream is defined as an element of the space $\mathbb{Q}^\omega$.
- The derivative σ' of a stream σ is defined as its tail.
- We call a set of streams $S \subseteq \mathbb{Q}^\omega$ *stable* whenever it is closed under derivative

Definition: A stream is called *simple* whenever it is an element of a stable finite subset of $\mathbb{Q}^\omega$.

Simple streams

Some useful definitions:

- A stream is defined as an element of the space $\mathbb{Q}^\omega$.
- The derivative σ' of a stream σ is defined as its tail.
- We call a set of streams $S \subseteq \mathbb{Q}^\omega$ *stable* whenever it is closed under derivative

Definition: A stream is called *simple* whenever it is an element of a stable finite subset of $\mathbb{Q}^\omega$.
Proposition: A stream is simple iff it is eventually periodic.

Towards algebra, or: streams as a `Num` type

(a.k.a. defining a ring structure on $\mathbb{Q}^{\mathbb{N}}$, coinductively)

```
instance Num a => Num [a] where

  (+)           = zipWith (+)
  (s:u)*(t:v) = s*t : map (t*) u + map (s*) v + (0:u*v)
  negate        = map negate
  fromInteger i = fromInteger i : repeat 0
  signum        = undefined
  abs           = undefined
```

The product here defined is the *convolution product*, here defined coinductively and traditionally defined as

$$(\sigma\tau)(n) = \sum_{i+j=n} \sigma(i)\tau(j)$$

Streams and power series

We regard streams and power series as equivalent, that is, we identify the power series given as

$$\sum_{i=0}^{\infty} s_n X^n$$

with the stream

$$(s_0, s_1, s_2, \ldots)$$

This can be done by regarding X as a concrete object corresponding to the stream $(0, 1, 0, 0, 0, \ldots)$. Note that X^n is the stream that has 1 in position n and 0 everywhere else.

Fibonacci in Haskell, coinductively

From the earlier system of behavioural equations

$$\begin{array}{lcllcl} o(x) & = & 1 & x' & = & y \\ o(y) & = & 1 & y' & = & x + y \end{array}$$

we can dirrectly derive the following Haskell code:

```
x = 1 : y
y = 1 : x + y
```

or alternatively and equivalently:

```
x = 1 : 1 : x + tail x
```

Towards algebra (2): defining division

```
instance Fractional a => Fractional [a] where
  fromRational i = fromRational i : repeat 0
  recip x = recip (head x) :
     map ((negate$recip$head x)*) (tail x)*recip x
```

Note that the inverse of a stream is defined if and only if the inverse of its initial value is defined.

From coinductive specifications to generating functions

We start from the coinductive specification of the Fibonacci stream:

$$\sigma = 1 : 1 : \sigma + \sigma'$$

First replace the cons operators with the formal variable X, and use $\sigma' = \frac{\sigma - 1}{X}$:

$$\begin{aligned}\sigma =& 1 + X + X^2(\sigma + \frac{\sigma - 1}{X}) \\ =& 1 + X\sigma + X^2\sigma\end{aligned}$$

Now we get:

$$\sigma = \frac{1}{1 - X - X^2}$$

From coinductive specifications to generating functions

We start from the coinductive specification of the Fibonacci stream:

$$\sigma = 1 : 1 : \sigma + \sigma'$$

First replace the cons operators with the formal variable X, and use $\sigma' = \frac{\sigma - 1}{X}$:

$$\begin{aligned}\sigma =& 1 + X + X^2(\sigma + \frac{\sigma - 1}{X}) \\ =& 1 + X\sigma + X^2\sigma\end{aligned}$$

Now we get:

$$\sigma = \frac{1}{1 - X - X^2}$$

And a new Haskell expression:

```
fibs = 1 / (1 : (-1) : (-1))
```

Characterization of recognizable streams

- We can now state Gérard Jacob's (1975) characterization of recognizable series (and use it as definition):

Characterization of recognizable streams

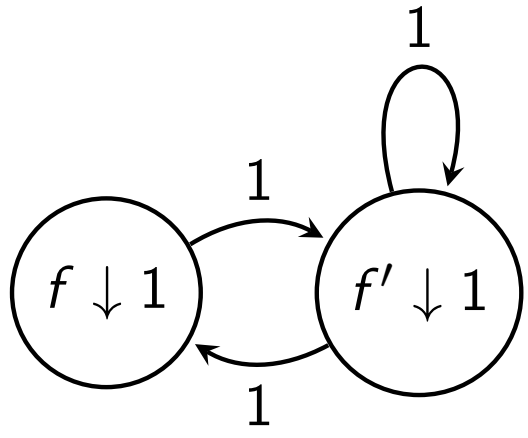
- We can now state Gérard Jacob's (1975) characterization of recognizable series (and use it as definition):
  Definition: A stream is called *recognizable* whenever it is an element of a linear subspace of $\mathbb{Q}^\omega$ that is
  1. stable, and
  2. finite-dimensional.

Characterization of recognizable streams

- We can now state Gérard Jacob's (1975) characterization of recognizable series (and use it as definition):
  Definition: A stream is called *recognizable* whenever it is an element of a linear subspace of $\mathbb{Q}^\omega$ that is
  1. stable, and
  2. finite-dimensional.
- Proposition: A stream is recognizable iff either of the two following equivalent conditions holds:
  1. It is recognized by a weighted automaton.
  2. It is generated by a finite system of behaivoural differential equations.

Representations of recognizable streams

1. Recurrence:
   $f(0) = f(1) = 1, f(n+2) = f(n) + f(n+1)$
2. b.d.e.:
   $o(f) = o(f') = 1, f'' = f + f'$
3. Weighted automaton:



4. Haskell:

```
fibs = 1 : 1 : fibs + d fibs
```

Characterization of algebraic streams

- We call a vector space over $\mathbb{Q}$ an algebra iff it is also a ring, compatible with the addition of the vector space.
- This allows us to present Wechler's (1983) characterization of algebraic series, in the same spirit as Jacob's characterization:

Characterization of algebraic streams

- We call a vector space over $\mathbb{Q}$ an algebra iff it is also a ring, compatible with the addition of the vector space.
- This allows us to present Wechler's (1983) characterization of algebraic series, in the same spirit as Jacob's characterization: Definition: A stream is called *algebraic* whenever it is an element of a linear subspace of $\mathbb{Q}^\omega$ that is
  1. stable, and
  2. an algebra, and
  3. finitely generated as an algebra.

Characterization of algebraic streams

- We call a vector space over $\mathbb{Q}$ an algebra iff it is also a ring, compatible with the addition of the vector space.
- This allows us to present Wechler's (1983) characterization of algebraic series, in the same spirit as Jacob's characterization: Definition: A stream is called *algebraic* whenever it is an element of a linear subspace of $\mathbb{Q}^\omega$ that is
  1. stable, and
  2. an algebra, and
  3. finitely generated as an algebra.
- Example: the following stream, the stream of Catalan numbers, is algebraic:

```
catalans = 1 : catalans^2
```

Some more examples

- Pascal's triangle (rational):

```
pascal n = 1 : sum [ pascal i | i <- [1..n] ]
```

or

```
pascal = (repeat 1^)
```

- Stirling numbers of the 2nd kind (rational):

```
s2 n = 1 : sum [ i * s2 i | i <- [1..n] ]
```

- Number of lambda terms with n free De Bruijn indices (transcendental):

```
lambda n = n : lambda n^2 + lambda (n+1)
```

The number of *m*-ary search trees on *n* keys

Haskell:

```
searchtrees m = take (m - 1) ones ++ searchtrees m ^ m
```

Generating function:

$$A(X) = X^{m-1}A^m(X) + \frac{1 - X^{m-1}}{1 - X}$$

Explicit formula:

Theorem 1 *The number of m-ary search trees on n keys is given by*

$$\tau_{m,n} = \sum \binom{k_+}{k_0, \ldots, k_{m-2}} \frac{[(m/(m-1))(k_+ - 1)]!}{[(k_+ - 1)/(m-1)]! k_+!}, \qquad (1)$$

where the sum is over all $(m-1)$*-tuples* $(k_0, \ldots, k_{m-2})$ *such that:* (i) $k_i \geq 0$ *for* $0 \leq i \leq m-2$, (ii) $m-1$ *divides* $k_+ - 1$, *and* (iii) $\sum_{j=0}^{m-2}(j+1)k_j = n+1$.

(from Fill/Dobrow: *The number of m-ary searchtrees on n keys*)

Exercise 1: Lazy caterer's sequence

Given a natural number n, let $l(n)$ denote the maximum number of pieces that can be obtained by cutting a pancake n times (without rearranging pieces).
Find a Haskell specification of l.

Exercise 1: Lazy caterer's sequence

Given a natural number n, let $l(n)$ denote the maximum number of pieces that can be obtained by cutting a pancake n times (without rearranging pieces).
Find a Haskell specification of l.
Recurrence:

$$l(0) = 1 \quad l(n+1) = l(n) + n$$

Exercise 1: Lazy caterer's sequence

Given a natural number n, let $l(n)$ denote the maximum number of pieces that can be obtained by cutting a pancake n times (without rearranging pieces).
Find a Haskell specification of l.
Recurrence:

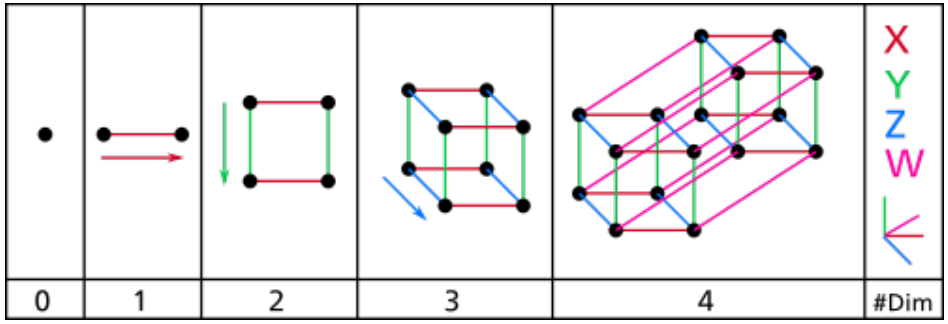
$$l(0) = 1 \quad l(n+1) = l(n) + n$$

$$\begin{array}{lcccccc} \sigma' & = & l(1) & l(2) & l(3) & \ldots \\ \sigma & = & l(0) & l(1) & l(2) & \ldots \\ \tau & = & 0 & 1 & 2 & \ldots \end{array}$$

This gives: $\sigma' = \sigma + \tau$
Solution:

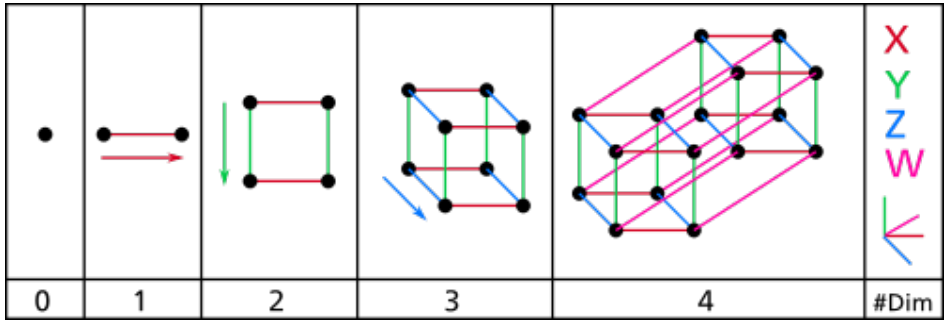
```
lazy = 1 : lazy + [1..]
```

Exercise 2: Counting edges of hypercubes



Let $h(n)$ be the number of edges in a n-dimensional hypercube. Find a Haskell specification of h.

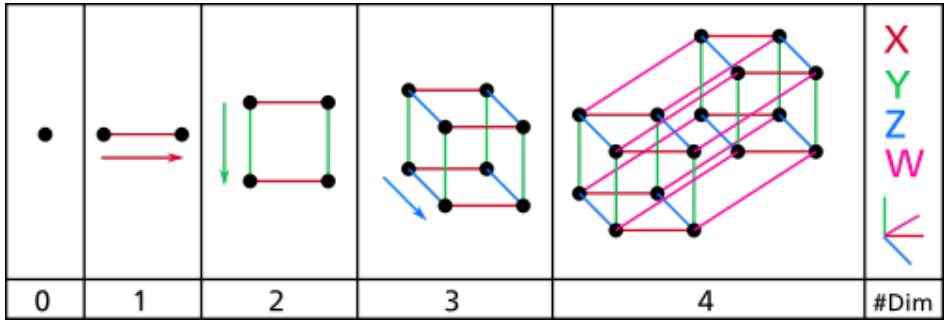
Exercise 2: Counting edges of hypercubes



Let $h(n)$ be the number of edges in a n-dimensional hypercube.
Find a Haskell specification of h.
Recurrence:

$$h(0) = 0 \quad h(n+1) = 2h(n) + 2^n$$

Exercise 2: Counting edges of hypercubes



Let $h(n)$ be the number of edges in a n-dimensional hypercube.
Find a Haskell specification of h.
Recurrence:

$$h(0) = 0 \quad h(n+1) = 2h(n) + 2^n$$

Solution:

```
powers2 = 1 : 2 * powers2
hypercube = 0 : 2 * hypercube + powers2
```

Using different products (and product rules)

We can vary the definition of the product, to obtain different types of behaviours:

- Hadamard product:

```
(*) = zipWith (*)
```

- Convolution product:

```
(s : u) * r @ (t : v) = s * t : u * r + s * v
```

- Shuffle product:

```
q @ (s : u) * r @ (t : v) = s * t : u * r + v * q
```

Selected references

If you are interested in learning more. . .

- M. Douglas McIlroy — *The Music of Streams*
- M. Douglas McIlroy — *Functional Pearls: Power Series, Power Serious*
- Jan Rutten — *Coinductive Counting with Weighted Automata*
- Joost Winter — *Coalgebraic Characterizations of Automata-Theoretic Classes*

Also:

- `http://oeis.org/`
- `http://www.mimuw.edu.pl/~jwinter/qstream/`