Monsters: programming and reasoning

A study and implementation of monadic streams

Chris Purdy and Venanzio Capretta

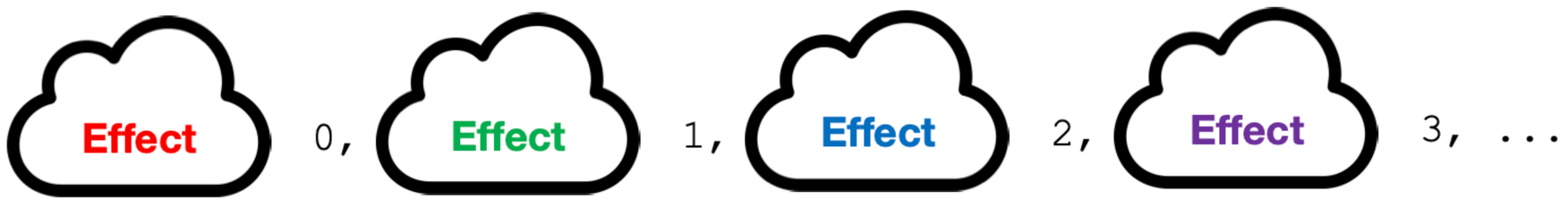
What is a monadic stream?

A **stream** is an infinite sequence of values. For example, the stream of natural numbers.

```
0, 1, 2, 3, 4, 5, …
```

To get to the n^{th} element (value) in the stream, you need to ‘traverse’ all of the previous elements.

A **monadic stream** is a stream where to access the next element, you need to **evaluate** an **effect**.



A **pure** (normal) stream is a monadic stream **without effects**, or where the effects don’t do anything.

To get to the n^{th} element in a monadic stream, you need to **traverse** **all previous elements and effects**.

Key concepts

Monadic streams are **potentially infinite streams** of **effectful computations**. We call these '**monsters**' for short.

$$\begin{aligned}&\mathbf{codata}\ \mathbb{S}_M\ A : \mathsf{Set}\\ &\quad \mathsf{mcons}_M : M\,(A \times \mathbb{S}_M\ A) \to \mathbb{S}_M\ A\end{aligned}$$

A **monad** is a **functor** M (a map from types to types) along with two **natural transformations** (polymorphic structure preserving maps):

$$\eta_A : A \to M\,A$$

$$\mu_A : M\,(M\,A) \to M\,A$$

Monads (rather, Kleisli arrows) can be thought of as representing effectful computations (*Moggi 1991*)

One example is the **Maybe** monad, whose corresponding effect is partiality:

$$\mathbf{Maybe}\,A = 1 + A$$

$$\eta_A = \mathsf{inr}$$

$$\mu_A\,(\mathsf{inl}\,\star) = \mathsf{inl}\,\star$$

$$\mu_A\,(\mathsf{inr}\,r) = r$$

Maybe-monster

By instantiating our type of monadic streams with the **Maybe** monad, we get the type of **Maybe**-monsters.

$$\mathbf{List} \cong \mathbb{S}_{\mathbf{Maybe}}$$

By expanding an element of $\mathbb{S}_{\mathbf{Maybe}}\,\mathbb{N}$, we can see that the type is isomorphic to that of possibly infinite **lists** (sometimes called **colists**)

$$\begin{aligned} &\mathsf{inr}\,(1, \mathsf{inr}\,(2, \mathsf{inr}\,(3, \mathsf{inr}\,(4, \mathsf{inl}\,\star)))) \\ &\qquad\cong \\ &\texttt{[1, 2, 3, 4]} \end{aligned}$$

Reader-monster

Another example is the **Reader**-monster.

$$\mathbf{Reader}_E\, A = E \to A$$

Kleisli arrows for this monad represent computations that read from a shared environment of type E.

The type $\mathbb{S}_{\mathbf{Reader}_E}$ corresponds to that of **possibly-infinite-state automata**.

This can (almost) be seen by unfolding the type definition, dropping constructors to reduce clutter.

$$E \to (A \times (E \to (A \times (E \to (A \times \ldots$$

The first function, when evaluated, returns a result of type A and a continuation. Both of these can depend on the choice of element in E. The function at the head of a **Reader**-monster is its current ‘state’.

Why this is interesting

Different kinds of coinductive data structures are represented by instantiations of monsters with different monads.

Maybe monad gives us **lazy lists**, Reader monad gives us **state machines**, List monad gives us **trees**, IO monad gives us **interactive processes**, and so on.

> "We should try to define **functions on monsters** that are **polymorphic in the monad**, that instantiate to operations on each of these data types."

With this in mind, we have developed a **library of generalised operations** **on monadic streams**, called `monster`.

Why this is interesting

Many are **liftings of common operations** on lists and streams (zips, scans, filtering, etc.).

Correspondences between instances of monsters and other data structures inspired more **context specific functions** (included in the `Examples` module of our library). Future work will focus on these specific applications.

We also define operations on monsters where the underlying functor is not a monad, and instead has different structure (applicative, traversable, foldable, representable, etc.).

Example: `scan`

The Haskell standard library version:

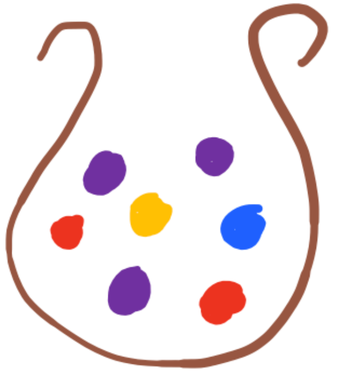
```
scan (+) 0 [1, 2, 3, 4, 5] = [0, 1, 3, 6, 10, 15]
```

Using the correspondence between lists and Maybe-monsters, we can generalise **`scan`** to any monadic stream.

Take a basic state machine (Reader-monster) that performs **one-dimensional edge detection**. Let's suppose you want the **running total of the edges detected**. You can do this using the generalised **`scan`** function.

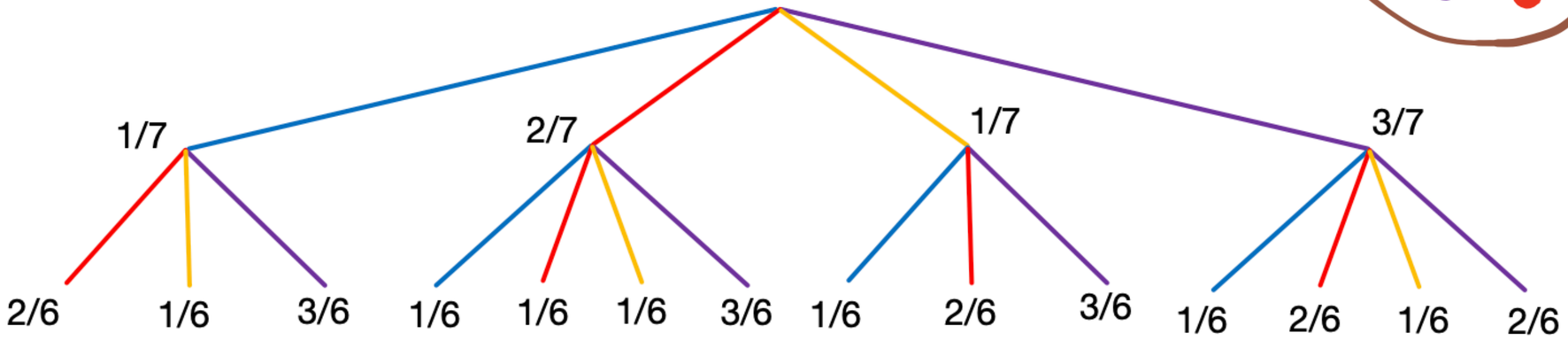
```
runSM edgeDetector [I,I,O,O,I,O,O,O,I,O] =
    ["No edge", "No edge", "Edge detected", "No edge", "Edge detected",
"Edge detected", "No edge", "No edge", "Edge detected", "Edge detected"]

edgeCounter = scan {add 1 if "Edge detected"} 0 edgeDetector

runSM edgeCounter [I,I,O,O,I,O,O,O,I,O] = [0,0,1,1,2,3,3,3,3,3,4,5]
```

`scan` use cases



Another use case is calculating sequential probabilities in probability trees.

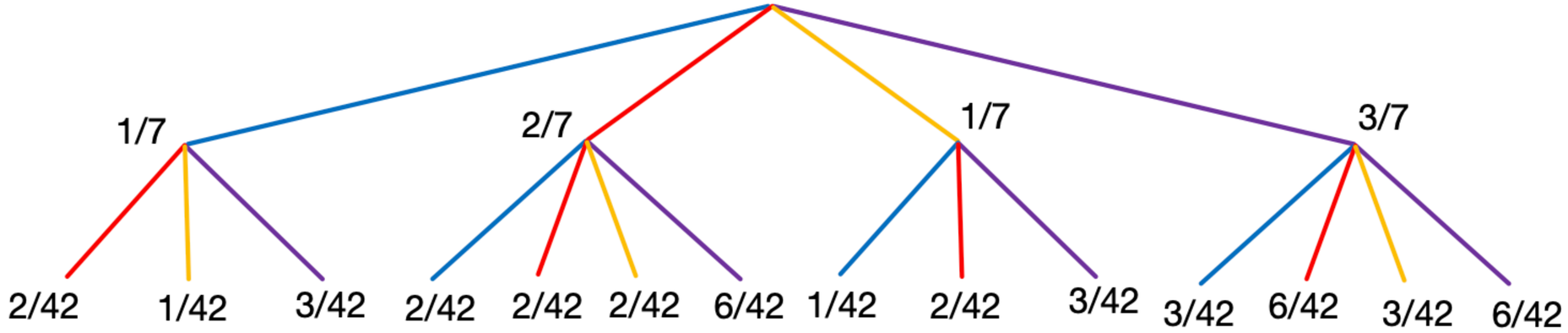
`probTree =`



Suppose you wanted to calculate the probabilities of blue then red, yellow then red, purple then blue etc.

`f` `probTree =`

`f = ???`



`scan` use cases

This can be done again using **`scan`**.

We can do this because (branch-labelled) **trees** correspond to **List-monsters**.

$$\mathbf{Tree}\ \text{(branch labelled)} \cong \mathbb{S}_{\mathbf{List}}$$

```
data BLTree a = MonStr [] a
```

By scanning the tree with multiplication, we get a tree where the probability at each branch is the probability *given all of the choices preceding it*.

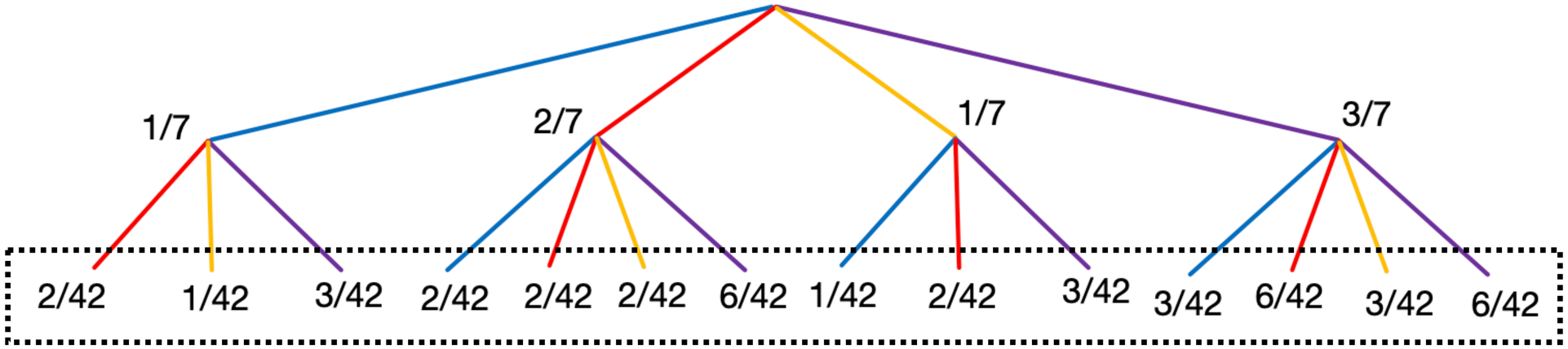
We can flatten the tree down to its second level (indexed from 0) to inspect it before and after the operation.

```
probTree !! 1 =  [              Red: (2,6), Yellow: (1,6), Purple: (3,6),
                  Blue: (1,6), Red: (1,6), Yellow: (1,6), Purple: (3,6),
                  Blue: (1,6), Red: (2,6),                Purple: (3,6),
                  Blue: (1,6), Red: (2,6), Yellow: (1,6), Purple: (1,3)]
```

scan use cases

```
sProbTree = scan' (*) probTree
```

```
sProbTree !! 1 =  [              R|B: (2,42), Y|B: (1,42), P|B: (3,42),
                   B|R: (2,42), R|R: (2,42), Y|R: (2,42), P|R: (6,42),
                   B|Y: (1,42), R|Y: (2,42),              P|Y: (3,42),
                   B|P: (3,42), R|P: (6,42), Y|P: (3,42), P|P: (6,42)]
```



Minimal structure

Depending on the function, we require different structure on the underlying functor.

It is interesting to try and find the minimal structure required for a particular function:

- `zip`, turns a pair of monsters into a monster of pairs - the functor must be **applicative**
- `tail`, discards the first element of the stream (and ‘absorbs’ the effect) - the functor must be a **monad**
- `scan` (with a starting element) - the functor must be **applicative**
- `scan'` (without a starting element) - the functor must be a **monad**

All monads are applicative in Haskell, but not all applicative functors are monads. In that sense, applicative functors have “less extra structure” than monads.

Finding the minimal extra structure required for a particular function could give a **characterisation of coinductive types that support that operation***.

Key results and insights

Our Haskell library **`monster`** that implements many functions to operate on general monadic streams.

Agda formalisations of two proofs with the **`agda-categories`** library:

- $A \mapsto \nu X \,.\, A \odot X$ is a **functor** for arbitrary bifunctor $\odot : \mathbf{D} \times \mathbf{C} \to \mathbf{C}$ *
- $A \mapsto \nu X \,.\, F(A \otimes X)$ is a **monodial endofunctor** on $(\mathbf{C}, \otimes)$ where $F : (\mathbf{C}, \otimes) \to (\mathbf{C}, \otimes)$ and $(\mathbf{C}, \otimes)$ is equipped with a "four middle interchange" $(X \otimes Y) \otimes (A \otimes B) \cong (X \otimes A) \otimes (Y \otimes B)$

These two proofs imply that the type constructor $\mathbb{S}_M$ is:

- An endofunctor, by instantiating $\odot = M(- \times -)$
- A cartesian monoidal endofunctor, by instantiating $\otimes = \times$ and $F = M$ (only when M is cartesian monoidal)

Key results and insights

It is **not the case that monadic streams are monads** if the underlying functor is an arbitrary **monad**. However, they are if the underlying functor is a **representable monad**.

If the underlying functor is a **comonad**, then the corresponding monadic stream is itself a **comonad**.

Monadic streams relate closely to **Functional Reactive Programming** (FRP), where they model streams of **input values** with **side-effects** (Perez et al.). Our library can be used in this context to manipulate these streams without fixing the side-effect (the monad).

Other FRP constructs also relate to specific cases of monadic streams - **monadic stream functions** (see Dunai) are monadic streams instantiated with the `ReaderT` monad transformer.

`github.com/venanzio/monster`

Thank you for listening and to the organisers for a great conference!