Programming with Monsters

Venanzio Capretta and Christopher Purdy

March 23, 2021

Abstract

A monadic stream is a potentially infinite sequence of values in which a monadic action is triggered at each step before the generation of the next element. A monadic action consists of some functional *container* inside which the unfolding of the stream takes place. Examples of monadic actions are: executing some input/output interactions, cloning the process into several parallel computations, executing transformations on an underlying state, terminating the sequence.

We develop a library of definitions and universal combinators to program with monadic streams and we prove several mathematical results about their behaviour.

We define the type of Monadic Streams (MonStr), dependent on two arguments: the underlying monad and the type of elements of the sequence. We use the following terminology: a monadic stream with underlying monad m is called an m-monster. The definition itself doesn't depend on the fact that the underlying operator is a monad. We define it generally: some of the operators can be defined without any assumptions, some others only need the operator to be a Functor or Applicative Functor. A different set of important combinators and theoretical results follow from the assumption that the operator is a Co-Monad rather than a Monad.

We instantiate the abstract MonStr type with several common monads (Maybe, List, State, IO) and show that we obtain well-known data structures. Maybe-monsters are lazy lists, List-monsters are non-well-founded finitely branching trees, IO-monsters are interactive processes. We prove equivalences between the traditional data types and their MonStr versions: the MonStr combinators instantiate to traditional operations.

Under some assumptions on the underlying functor/monad, we can prove that the MonStr type is also a Functor, and Applicative, or a Monad, giving us access to the special methods and notations of those type classes.

1 Introduction

Intuitive explanation of Monadic Streams and motivation.

2 Monadic Streams

A monadic stream is a sequence of values in which every stage is obtained by triggering a monadic action. If σ is such a stream, it will consist of an action for a certain monad M that, when executed, will return a head (first element) and a tail (continuation of the stream). This process can be continued in a non-well-founded way: streams constitute a coinductive type.

Formally the type of streams over a monad M (let's call them M-monsters) is defined as:

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

Categorically, we can see this type as the final coalgebra of the functor $F\,X - == M\,(A \times X)$. (This final coalgebra does not necessarily exists for every M, we will discuss this issue below.)

Instantiating M with some of the most well-known monads leads to versions of known data types or to interesting new constructs.

If we instantiate M with the identity monad, we obtain the type of pure streams. Its usual definition is the following:

$$\begin{aligned}&\textbf{codata}\ \mathbb{S}_A : \mathsf{Set}\\ &\quad (\lhd) : \mathbb{N} \to \mathbb{S}_{\mathbb{N}} \to \mathbb{S}_{\mathbb{N}}.\end{aligned}$$

(The type of the constructor has been curried, as is common.) An element of $\mathbb{S}_A$ is an infinite sequence of elements of A: $a_0 \lhd a_1 \lhd a_2 \lhd \cdots$.

If we instantiate M with the Maybe monad we obtain the type $\mathbb{S}_{\mathsf{Maybe},A}$, equivalent to the type of lazy lists $\mathsf{List}(A)$. The Maybe monad is a functor that adds an extra element to the argument type: $\mathsf{Maybe}\,X \cong X + 1$, so the single constructor $\mathsf{mcons}_{\mathsf{Maybe}} : \mathsf{Maybe}\,(A \times \mathbb{S}_{\mathsf{Maybe},A}) \to \mathbb{S}_{\mathsf{Maybe},A}$ is equivalent to two constructors (for $\mathsf{Nothing}$ and Just):

$$\begin{aligned}&\textbf{codata}\ \mathsf{List}(A) : \mathsf{Set}\\ &\quad (\lhd) : A \times \mathsf{List}(A) \to \mathsf{List}(A)\\ &\quad \mathsf{nil} : \mathsf{List}(A).\end{aligned}$$

This means that an element of $\mathsf{List}(A)$ is either an empty sequence nil or a non-empty sequence $a \lhd \sigma$ where $a : A$ and σ is recursively an element of $\mathsf{List}(A)$. Since this is a coinductive type, the constructor $(\lhd)$ can be applied an infinite number of times. Therefore $\mathsf{List}(A)$ is the type of finite and infinite sequences.

[**Venanzio's comment:** *Add example with* $M = \mathsf{List}$ *(maybe introduce different notations for lazy and strict lists) and give some simple example of functions on monsters.*]

It is important to make two observations about M.

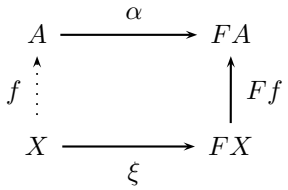
First, M does not need to be a monad for the definition to make sense. In fact we will obtain several interesting results when M satisfies weaker conditions, for example being just a functor. So we will take M to be any type operator (but see second observation) and we will explicitly state what properties we assume about it. The most important instances are monads and it is convenient

to use the facilities of monadic notation in programming and monad theory in reasoning.

The second observation is that it is not guaranteed in general that the **codata** type is well-defined. Haskell will accept the definition when M is any functor, but mathematically the type is well defined only when the functor $F\,X = M\,(A \times X)$ has a final coalgebra.

Definition 1. *For any functor F, a* coalgebra *for F is pair $\langle A, \alpha\rangle$ consisting of a type A and a function $\alpha : A \to FA$.*

We say that $\langle A, \alpha\rangle$ is a final F-coalgebra *if, for every coalgebra $\langle X, \xi : X \to FX\rangle$, there is a unique coalgebra morphism between the the two coalgebras: $f : \langle X, \xi\rangle \to \langle A, \alpha\rangle$. Such a morphism is a function between the types that commutes with the coalgebra functions:*



$$\alpha \circ f = Ff \circ \xi.$$

This definition means that we can define function into a coinductive type by giving a coalgebra. For example, consider the data type of pure streams of natural numbers: The type $\mathbb{S}_A$ is the final coalgebra of the functor $F(X) = A \times X$.

We can define a function into $\mathbb{S}_A$ by defining a coalgebra on the domain type. For example, if we want to define a function that maps any natural number n to the stream of numbers starting from n, $n \lhd (n+1) \lhd (n+2) \lhd (n+3) \lhd \cdots$, we can do it by the following coalgebra on $\mathbb{N}$:

$$\begin{aligned} &\xi : \mathbb{N} \to \mathbb{N} \times \mathbb{N} \\ &\xi\, n = \langle n, n+1\rangle \end{aligned}$$

(Note that the target type of the coalgebra is $F(\mathbb{N}) = \mathbb{N} \times \mathbb{N}$: The first $\mathbb{N}$ is the parameter of the functor, while the second $\mathbb{N}$ is the carrier of the coalgebra.)

In practical programming, we often let the coalgebra ξ be implicit by directly defining the function f recursively. For example, the function above would be defined as:

$$\begin{aligned} &f : \mathbb{N} \to \mathbb{S}_{\mathbb{N}} \\ &f\, n = n \lhd f\,(n+1) \end{aligned}$$

Here the presence of the parameter n and the argument of the recursive call $n+1$ implicitely give the coalgebra $n \mapsto \langle n, n+1\rangle$.

[Venanzio's comment: *Talk about definition by guarded recursion. Show specific examples of definition by guarded recursion for monsters.*]

The definition of $\mathbb{S}_{M,A}$ is not meaningful for all Ms, because the final coalgebra may not exist or not be unique. A useful result is that a functor has a

final coalgebra if it is a container [1], and F is a container if M is [2]. This is the case for all the instances that we consider (but there are well known counterexamples, like the powerset functor and the continuation functor).

Therefore we will silently assume that M is a container and that the final coalgebra exists. Cofinality means that we can define functions into the coalgebra by *corecursion* and we can prove properties of its elements by *coinduction*.

3 Examples

Instantiations with Identity (Pure Streams), Maybe (Lazy Lists), List (Trees), IO (Processes), State (Stateful Streams).

Definition of the *standard* function on those data types in a general way.

4 Instances of Functor, Applicative, Monad

Show that monadic streams are an instance of these three classes, under some assumptions about the underlying monad.

So far we determined that we think that MonStr satisfies the monad laws if the underlying monad is commutative and idempotent [3]. (Proof?)

What about Applicative?

Give conterexamples where it isn't a monad (using State).

Also examples when it is a monad even if the underlying monad is not commutative and idempotent: with List as underlying monad we obtain Trees, which are a monad. Can this be generalized.

4.1 Functor instance in Haskell

```
unwrapMS :: MonStr m a → m (a, MonStr m a)
unwrapMS (MCons m) = m

transformMS :: Functor m ⇒ (a → MonStr m a → (b, MonStr m b)) →
                                  MonStr m a → MonStr m b
transformMS f s = MCons $ fmap (λ(h,t) → f h t) (unwrapMS s)

instance Functor m ⇒ Functor (MonStr m) where
   -- fmap :: (a → b) → MonStr m a → MonStr m b
   fmap f = transformMS (λa s → (f a, fmap f s))
```

4.1.1 Functor proof

Proof that `fmap` **id** = **id**

```
fmap id = transformMS (λa s → id a, fmap id s)
-- definition of 'transformMS'
= λs → MCons $ fmap (λ(h,t) → (λa s → (id a, fmap id s)) h t) (unwrapMS s)
```

```
-- application of `(λa s → (id a, fmap id s))` to `h t`
= λs → MCons $ fmap (λ(h,t) → (id h, fmap id t)) (unwrapMS s)
-- take as assumption that `fmap id t == id t` – coinductive hypothesis?
= λs → MCons $ fmap (λ(h,t) → (id h, id t)) (unwrapMS s)
-- definition of `id`
= λs → MCons $ fmap (λ(h,t) → (h,t)) (unwrapMS s)
-- definition of `id`
= λs → MCons $ fmap id (unwrapMS s)
-- `unwrapMS` returns an element in a functor, so `fmap id == id` in this case
= λs → MCons $ id (unwrapMS s)
-- application of `id`
= λs → MCons $ (unwrapMS s)
-- MCons and unwrapMS are inverses
= λs → id s
= id
```

Proof that `fmap (f . g) == (fmap f) . (fmap g)`

To simplify notation in the proof, we introduce the notation `tr(f)` for the expression `\(h,t) -> (f h, fmap f t)`.

We're going to use the fact that the underlying operator `m` is a functor and so satisfies the functor laws, and also that the pairing operator `(,)` is functorial in both arguments.

By definition of `fmap` for monsters, we have that

```
fmap f s = transformMS (λ a s → (f a, fmap f s)) s
         = MCons $ fmap (λ (h,t) → (λa s → (f a, fmap f s) h t)) (unwrapMS s)
         = MCons $ fmap (λ (h,t) → (f h, fmap f t)) (unwrapMS s)
         = MCons $ fmap tr(f) (unwrapMS s)
```

We also use the fact that the monster constructor `MCons` and the function `unwrapMS` that removes it are inverse of each other.

```
fmap (f ∘ g) = transformMS (λa s → (f ∘ g) a, fmap (f ∘ g) s)
-- definition of `transformMS`
= λs → MCons $ fmap (λ(h,t) → (λa s → ((f ∘ g) a, fmap (f ∘ g) s)) h t) (unwrapMS s)
-- application of `(λa s → ((f ∘ g) a, fmap (f ∘ g) s))` to `h t`
= λs → MCons $ fmap (λ(h,t) → ((f ∘ g) h, fmap (f ∘ g) t)) (unwrapMS s)`
-- take as assumption that `fmap (f ∘ g) t == (fmap f ∘ fmap g) t` – coinductive hypothesis?
= λs → MCons $ fmap (λ(h,t) → ((f ∘ g) h, (fmap f ∘ fmap g) t)) (unwrapMS s)
-- By fuctoriality of pairing in both components
= λs → MCons $ fmap (tr(f) ∘ tr(g)) (unwrapMS s)
-- By functoriality of the underlying operator m
= λs → MCons $ (fmap tr(f)) ∘ (fmap tr(g)) (unwrapMS s)
-- MCons and unwrapMS are inverses
= λs → MCons $ fmap tr(f)) $ unwrapMS $ MCons $ fmap tr(g) (unwrapMS s)
= λs → MCons $ fmap tr(f)) $ unwrapMS $ fmap g s
= λs → fmap f $ fmap g s
= fmap f ∘ fmap g
```

4.2 Comonad instance in Haskell

```
instance Comonad w ⇒ Comonad (MonStr w) where
  —extract :: MonStr w a → a
  extract = extract ∘ headMS

  —duplicate :: MonStr w a → MonStr w (MonStr w a)
  duplicate s = MCons $ fmap (λ(h,t) → (s, duplicate t)) (unwrapMS s)
```

`headMS` returns the first element of the first pair in the monster, wrapped in the underlying functor. Since this functor is required to be a comonad, extract can be used to return the wrapped values inside.

We introduce `mm(s)` for the "monster matrix" formed by a monadic stream `s` - this is defined as a monster where the first element is `s`, the second is the tail of `s`, the third is the tail of the tail of `s`, and so on.

The `duplicate` instance for monadic streams forms this "monster matrix": `duplicate s = mm(s)`

4.2.1 Comonad proof

Proof that `extract . duplicate == id`

```
extract ∘ duplicate = λs → (extract ∘ headMS) ∘ duplicate $ s
= λs → (extract ∘ headMS) (duplicate s)
— duplicate forms a monster matrix
= λs → (extract ∘ headMS) mm(s)
— the first element of the monster matrix is the original stream, as per the definition above.
— taking the head returns the original stream guarded by the operator m, and extract then retrieves
— s due to m being comonadic
= λs → s
= id
```

Proof that `fmap extract . duplicate == id`

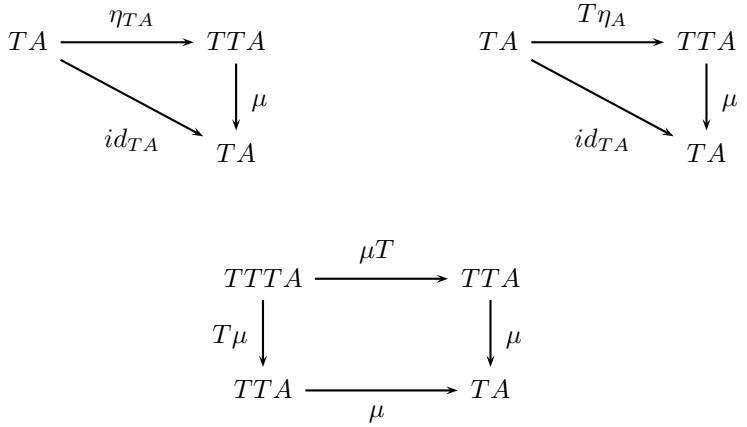
```
fmap extract ∘ duplicate = λs → fmap (extract ∘ headMS) mm(s)
— by functoriality of underlying operator m
= λs → (fmap extract) ∘ (fmap headMS) $ mm(s)
— fmapping headMS replaces the monster at each step in the monster matrix with its
— head, inside the operator. extract then removes these comonadic operators, resulting
— in the same monster that we started with
= λs → s
= id
```

Proof that `duplicate . duplicate == fmap duplicate . duplicate`

5 Monad proof

Definition 2. *For any endofunctor T on a category C, a* monad *is a triple (T, μ, η), of T and two natural transformations $\mu : TT \Rightarrow T$ and $\eta : 1_C \Rightarrow T$*

(1_C being the identity functor on C) such that these diagrams commute:



For the functor $\mathbb{S}_M$ to be a monad, the monad laws (these commuting diagrams) need to be satisfied. However, assuming a monad $(\mathbb{S}_M, \mu, \eta)$, the left identity law (top-left diagram) does not hold in general, given an arbitrary underlying monad M. This is due to a correct μ not being constructible.

Here, subscripts of η (and μ) indicate their monad, and not components of the natural transformation - these are instead indicated using function application: $\eta_M(x)$ where $x : A$ implies that we are using the component of η_M at A.

First, note that $\eta_{\mathbb{S}_M}$ has to be defined as:

$$\eta_{\mathbb{S}_M}(x) = mcons_M(\eta_M((x, \eta_{\mathbb{S}_M}(x)))$$

There is one choice of A for the first element of the pair, and only one way of lifting pairs into the inner monad M, which is by using η_M. The second element of the pair has to be a $\mathbb{S}_{M,A}$, and the only way to produce this here is by a recursive call.

We introduce the notation $|^i_M$ to denote a monadic stream constructor guarded by a monadic action $m_i : A \to MA$. Since each monadic action in M may be different, i is just a label to indicate when they are the same. That is, for any i, and a fixed $a : A$:

$$s_0 : \mathbb{S}_{M,A} = \overset{i}{\underset{M}{|}}\, a, s_0$$
$$s_1 : \mathbb{S}_{M,A} = \overset{i}{\underset{M}{|}}\, a, s_1$$
$$s_0 = s_1$$

If $i = \eta$, this monadic action is the monad M's unit, the natural transformation η_M. For any other label i, the monadic action is arbitrary. If there is no label, whether the monadic actions are equal is not being considered.

Proof. If $(\mathbb{S}_M, \mu, \eta)$ is a monad, then it needs to obey the left identity law, which in this case is

$$\mu_{\mathbb{S}_M}(\eta_{\mathbb{S}_M}(s)) = s \qquad s : \mathbb{S}_{M,A}$$

For an arbitrary steam s, defined as:

$$s = \overset{0}{\underset{M}{|}}\, a_0, \overset{1}{\underset{M}{|}}\, a_1, \overset{2}{\underset{M}{|}}\, a_2, \dots$$

First we construct the M-monster of M-monsters (termed an M-matrix) as per this requirement:

$$\eta_{\mathbb{S}_M}(s) : \mathbb{S}_{M,\mathbb{S}_{M,A}} = \overset{\eta}{\underset{M}{|}}\, s, \overset{\eta}{\underset{M}{|}}\, s, \overset{\eta}{\underset{M}{|}}\, s, \dots$$

To take the diagonal, you have to take the first element of the first stream, the second of the second stream, and so on. The law says that $\eta_{\mathbb{S}_M}(s)$ needs to be somehow manipulated to form s.

We reason that $\mu_{\mathbb{S}_M}$ cannot be constructed by noting that there is no function out of a monad in general (no function $f : MA \to A$ for an arbitrary functor M over which a monad is defined), and then traversing the M-matrix to see which monadic actions *necessarily* have to guard each element in the M-monster resulting from an application of $\mu_{\mathbb{S}_M}$ to $\eta_{\mathbb{S}_M}(s)$.

Observe that the first element of the first stream a_0 in $\eta_{\mathbb{S}_{M,A}}(s)$ is guarded by two monadic actions, one of them η, the other an arbitrary action, which we'll call $m_0 : A \to MA$. By the left identity on the monad M, $\eta(m_0(a_0)) = m_0(a_0)$, which indicates that in the absolute best case, the first element of the M-monster is guarded by m_0.

$$\eta_{\mathbb{S}_M}(s) : \mathbb{S}_{M,\mathbb{S}_{M,A}} = \overset{\eta}{\underset{M}{|}}\, (\overset{0}{\underset{M}{|}}\, \overset{\downarrow}{a_0}, \overset{1}{\underset{M}{|}}\, a_1, \dots), \overset{\eta}{\underset{M}{|}}\, s, \dots$$

Now that we know the first element of the resulting stream a_0 will be, in the best case, guarded by m_0, we look at what actions guard the second element a_1. WLOG, there are two ways to extract a_1, either by taking the second element of the second M-monster, or by taking the second element of the first M-monster. This gives two to consider.

Case 1: In the first case: if we take the second element of the second M-monster, then we have to consider what actions guard the element a_1 - these in order are η, η, m_0, m_1, obtained by traversing along $\eta_{\mathbb{S}_M}(s)$ to the second s, and then along s to a_1. By using the left identity again, this could reduce in the best case to $m_0(m_1(a_1))$.

$$\eta_{\mathbb{S}_M}(s) : \mathbb{S}_{M,\mathbb{S}_{M,A}} = \overset{\eta}{\underset{M}{|}} s, \overset{\eta}{\underset{M}{|}} (\overset{0}{\underset{M}{|}} a_0, \overset{1}{\underset{M}{|}} \overset{\downarrow}{a_1}, \dots), \dots$$

Now realise that in s, the element a_0 comes before a_1. This means that any monadic actions guarding a_0, have to also guard a_1. Crucially, the m_0 actions that guard them *are not the same*, one comes from the first s, and the other from the second s. This means a_1 has to be guarded by *two copies* of m_0 - one from the s in the first index of $\eta_{\mathbb{S}_M}(s)$ (which is required to guard a_0), and one from the second.

As you cannot get rid of this extra monadic action in general, s is not constructible from $\eta_{\mathbb{S}_M}(s)$ in this way. The problem with a duplicated monadic action can be demonstrated with the State monad - if you have an action that adds 3 to an underlying state, duplicating this would produce an overall action of adding 6.

Case 2: The second case to consider was to take the second element from the first stream. To show this doesn't work generally either, we have to instead consider the right monad identity law. This states that:

$$\mu_{\mathbb{S}_M}(\mathbb{S}_M \eta_{\mathbb{S}_M}(s)) = s \qquad s : \mathbb{S}_{M,A}$$

Where $\mathbb{S}_M \eta_{\mathbb{S}_M}$ lifts $\eta_{\mathbb{S}_M}$ to a function on monadic streams, which effectively applies it to each of the elements (this is our `fmap` definition from earlier). We redefine the M-matrix accordingly:

$$\begin{aligned}
&a_i : A \\
&\eta_{\mathbb{S}_M}(a_i) : \mathbb{S}_{M,A} = \overset{\eta}{\underset{M}{|}} a_i, \overset{\eta}{\underset{M}{|}} a_i, \overset{\eta}{\underset{M}{|}} a_i, \dots \\
&\mathbb{S}_M \eta_{\mathbb{S}_M}(s) : \mathbb{S}_{M,\mathbb{S}_{M,A}} = \overset{0}{\underset{M}{|}} \eta_{\mathbb{S}_M}(a_0), \overset{1}{\underset{M}{|}} \eta_{\mathbb{S}_M}(a_1), \overset{2}{\underset{M}{|}} \eta_{\mathbb{S}_M}(a_2), \dots
\end{aligned}$$

Using similar logic to before, you can show that a_0 is guarded by at least m_0. However, if you pick the second element from the second stream, you get another a_0. The $\mu_{\mathbb{S}_{M,A}}$ operation has to cross over to the next 'sub-monster' $\eta_{\mathbb{S}_M}(a_1)$ to get the value a_1. This loops us back to the first case, where we show that doing this causes a duplication of monadic actions, considering the left identity.

□

6 Applications

[**Chris's comment:** *Show a few applications and motivate the usefulness of Monadic Streams.*]

One application from monadic streams is they can be seen as a generalised form of Yampa's [4] signal functions. For example, one can implement the integral signal function using the Reader functor. First you define the type of monadic stream - we use the same names as for the same constructs in Yampa, to make the correspondence clear:

```
type DTime = Double

type SignalFunc a b = MonStr ((→) (DTime, a)) b
```

```
integral :: SignalFunc Double Double
integral = MCons integralAuxF

integralAuxF :: (DTime, Double) → (Double, SignalFunc DTime Double)
integralAuxF (_, a) = (0 , integralAux 0 a)
   where integralAux igrl a_prev = MCons (λ(dt, a') →
      (igrl' dt, integralAux (igrl' dt) a'))
      where igrl' dt' = igrl + (dt' ∗ a_prev)
```

7 Conclusions

Discussion of related literature.

Open problems.

Future Work.

References

[1] Michael Abott, Thorsten Altenkirch, and Neil Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.

[2] Venanzio Capretta and Jonathan Fowler. The continuity of monadic stream functions. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12. IEEE Computer Society, 2017.

[3] John Clark and Robert Wisbauer. Idempotent monads and *-functors. *Journal of Pure and Applied Algebra*, 215(2):145 – 153, 2011.

[4] Courtney, Antony and Nilsson, Henrik and Peterson, John The Yampa Arcade *Association for Computing Machinery*, 10.1145/871895.871897, 2003