Programming with Monsters

Venanzio Capretta and Christopher Purdy

February 11, 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:

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

Categorically, we can see this type as the final coalgebra of the functor $F\,X = M\,(A \times X)$.

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 explicitely 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 functor $F\,X = M\,(A \times X)$ has a final coalgebra. So the definition of $\mathbb{S}_{M,A}$ is not meaningful for all Ms. 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 (\(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 (\a s → (f a, fmap f s))
```

4.1.1 Functor proof

Proof of $fmap\ id == id$

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

□

Proof of $fmap\ (f \circ g) == (fmap\ f) \circ (fmap\ g)$

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 satisfied 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 (\ a s -> (f a, fmap f s)) s
         = MCons $ fmap (\ (h,t) -> (\a s -> (f a, fmap f s) h t)) (unwrapMS s)
         = MCons $ fmap (\ (h,t) -> (f h, fmap f t)) (unwrapMS s)
         = MCons $ fmap tr(f) (unwrapMS s)
```

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

```
fmap (f . g) = transformMS (\a s → (f . g) a, fmap (f . g) s)
-- definition of ‘transformMS‘
= \s → MCons $ fmap (\(h,t) → (\a s → ((f . g) a, fmap (f . g) s)) h t) (unwrapMS s)
-- application of ‘(\a s → ((f . g) a, fmap (f . g) s))‘ to ‘h t‘
= \s → MCons $ fmap (\(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?
= \s → MCons $ fmap (\(h,t) → ((f . g) h, (fmap f . fmap g) t)) (unwrapMS s)
-- By fuctoriality of pairing in both components
= \s → MCons $ fmap (tr(f) . tr(g)) (unwrapMS s)
-- By functoriality of the underlying operator m
= \s → MCons $ (fmap tr(f)) . (fmap tr(g)) (unwrapMS s)
-- MCons and unwrapMS are inverses
= \s → MCons $ fmap tr(f)) $ unwrapMS $ MCons $ fmap tr(g) (unwrapMS s)
= \s → MCons $ fmap tr(f)) $ unwrapMS $ fmap g s
= \s → fmap f $ fmap g s
= fmap f . fmap g
```

5 Applications

Show a few application and motivate the usefulness of Monadic Streams.

6 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.