Programming with Monsters

Submitted May 2021, in partial fulfilment of
the conditions for the award of the degree **BSc Computer Science**.

14326637
School of Computer Science
University of Nottingham

I hereby declare that this dissertation is all my own work, except as indicated in the text:

Signature 

Date 09/05/2021

Programming with Monsters

Christopher Purdy and Venanzio Capretta

May 9, 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. Monadic actions are a wide class of side effects, modes of execution, evaluation paradigms; they are realised as functional *containers* 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 (the kind of possible actions) 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 Comonad rather than a Monad.

We instantiate the abstract MonStr type with several common monads (Maybe, List, Reader, 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, Reader-monsters are finite state machines, 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 begin to prove that the MonStr type is also a Functor, an Applicative, a Monad, or a Comonad, giving us access to the special methods and notations of those type classes.

1 Introduction

Monadic streams are interesting and powerful ways of programming computational processes that generate effects.

They encapsulate the notion of (potentially infinite) streams of values, where traversing a stream from one element to the next can trigger, for example, the splitting of the process into several streams or the termination of the stream.

Traversing the stream could also require extra inputs, or cause I/O events, such as printing to the screen or asking for user interaction. These extra computations, among others, come under the umbrella term *computational effects* or simply *effects*.

We will also call these effects *monadic actions* or just *actions*, referring to monads; these, in functional programming, are type constructors whose elements encapsulate the concept of effectful computations.

To understand monadic streams, we must first look at pure streams.

A *pure stream* is an infinite sequence of values, for example (using the Haskell notation that we will introduce later), the stream of all natural numbers:

```
nats = 0 <: 1 <: 2 <: 3 <: 4 <: ...
```

The type of pure streams with elements of type `a` is denoted by `Stream a`. The cons operation `(<:)` is used to append an element in front of an existing stream. Streams are infinite, so we must use recursion to define them.

For example, the constant stream of ones is defined as

```
ones = 1 <: ones
```

The list of natural numbers can be defined as

```
nats = fromNat 0
  where fromNat n = n <: fromNat (n+1)
```

This kind of self-referential definition is called *corecursion*. It differentiates from the standard inductive recursion in three ways: first, the recursive calls are not required the be applied to a smaller argument (in the example we apply `fromNat` to $n+1$, which is larger than n); second, there is no need for a base case where the recursion will terminate; third, the recursive calls must be *guarded*, that is, we must ensure that at least part of the result is produced before the recursive call (here `n <:`).

For this to work in practice, corecursive structures (like our stream of natural numbers above) must be evaluated *lazily*, meaning 'only as needed'. The program will not try to generate the whole stream, but will only produce one element at a time when required. The lazy evaluation strategy allows definitions like this in Haskell and other similar programming languages.

A *monadic stream* is a sequence of values in which every constructor `(<:)` is enclosed in a monadic action: to obtain the head (first element) and tail

(continuation) of the stream, we must execute the monadic action.
Their type constructor is called `MonStr`, so we call them *monsters* for short.

For example a Maybe-*monster* is a sequence of elements that is either nothing (meaning the sequence is finished) or some head element followed by a tail. This allows us to define both finite and infinite sequences. The definitions of `ones` and `nats` are still valid Maybe-monsters. In addition, we can now define finite sequences:

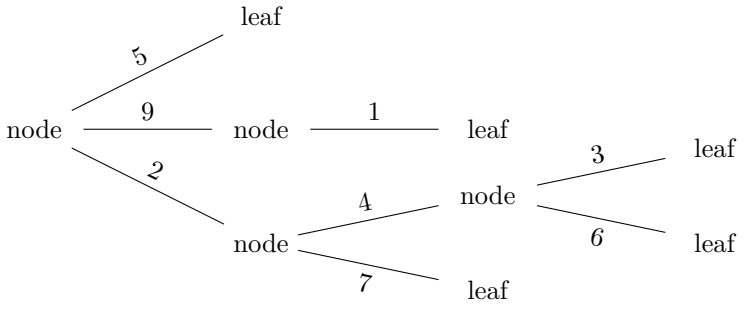
```
1 <: 2 <: 3 <: empty
```

where `empty` is the Maybe-monster given by the 'nothing' action.

Another example is List-*monsters*, where the effect is to produce a list of values, which can be thought of as a non-deterministic computation. An element consists of a list of heads and tails: there may be many (zero or more) branches, each with its own head and tail. List-monsters are actually arbitrarily branching trees, with labels (elements) on the branches instead of the nodes:

```
node [ 5 <: leaf
     , 9 <: node [ 1 <: leaf
                 ]
     , 2 <: node [ 4 <: node [ 3 <: leaf
                             , 6 <: leaf
                             ]
                 , 7 <: leaf
                 ]
     ]
```

`leaf` is a tree with no branches (`leaf = node []`). This example is the List-monster that corresponds to the tree below. Each branch has its own tail, or collection of subtrees in this case.



This is a finite tree, but the trees are allowed to be infinite, in both the width of the tree (the number of branches at one level), and the depth of the tree (the length of the longest path from the root to a leaf).

In these two examples, we have used the Maybe and List monads to construct monsters. The Maybe monad gives the effect of a computation that might

return Nothing; in the case of a Maybe-monster, returning Nothing terminates the stream. The List monad embodies computations that can return any number of values; in the case of List-monsters this gives us our arbitrary number of branches at each level.

One important feature of monads is that consecutive effects can be 'flattened', 'joined' or 'sequenced' into a single one: a list of lists can be concatenated into a single list.

Some other types of monsters include IO-*monsters*, which represent interactive processes, and Reader-*monsters* which represent finite state automata. All of these variations of monadic stream we have introduced are explained and expanded on in section 4.

All of our examples will be written in Haskell, and sometimes pseudo-Haskell. In section 6, where we prove various properties of monsters, we will use Haskell for equational reasoning, alongside more type theoretic notation for the same purpose. These two notations are interwoven throughout the article, but the distinction between them should be clear: all Haskell code is surrounded with a black bounding box, and displayed in monospace font.

The primary purpose of this paper is to motivate the library we have developed: an extensive set of combinators and operations on monadic streams, along with some concrete examples, written in Haskell. This library is envisaged as motivation for the potential of monadic streams as a general model of intensional effectful computations.

This paper acts as a first draft to two separate papers that we are looking to publish: one addressing the mathematical properties of monadic streams, and another on how monsters can be used in practice, utilising our Haskell library. Once finalised and properly documented, the library will be released on Hackage.

To briefly explain the layout: the next two sections precisely define monadic streams, and the coinduction principle, the means by which we can prove properties of monadic streams (and, more generally, corecursive structures).

Section 4 and 5 focus on developing concrete examples, and how specific functions from our library specialise in the different contexts presented by various kinds of monsters.

In section 6 we begin to prove categorical properties of monadic streams.

In the final two sections we talk about related work, specifically in Functional Reactive Programming (FRP), and conclude with open problems and general reflections.

2 Monadic Streams

A monadic stream is a sequence of values in which every subsequent element 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*) with elements of type A 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 $F_M\,X = M\,(A \times X)$. We give definitions of (final) coalgebras and a summary of their properties below. The final coalgebra does not necessarily exist for every M, but it does for most of the commonly used monads; we discuss this issue below.

The monadic streams definition is a type operator that maps a type A to the type of M-monsters with elements of type A; we may indicate the operator by $\mathbb{S}_M$ and the type by the slightly different notation $(\mathbb{S}_M\,A)$. This notation will be useful when we prove properties of the operator, for example that it is a functor, applicative functor, monad, or comonad, which we do in Section 6.

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:

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

(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: $a_0 \triangleleft a_1 \triangleleft a_2 \triangleleft \cdots$.

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: $\mathsf{Maybe}\,X$ contains copies of each element $x : X$, denoted by $\mathsf{Just}\,x$, plus a *empty* element $\mathsf{Nothing}$. So $\mathsf{Maybe}\,X \cong X + 1$. 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):

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

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

Another example is when the underlying monad is $M = \mathsf{List}$ itself. In this case each entry in the stream is a list of pairs of heads and tails. This is equivalent trees of arbitrary branching degrees (finitely branching if we use only finite lists, but also countably infinite branches if we use lazy lists). Since the type is coinductive, the trees can be non-well-founded, that is, they may be infinitely deep.

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 operator, but mathematically the type is well defined only when $F_M\, X = M\,(A \times X)$ is a functor with 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: $\mathsf{ana}_\xi : \langle X, \xi\rangle \to \langle A, \alpha\rangle$. Such a morphism is a function between the types that commutes with the coalgebra functions:*

$$\begin{array}{ccc} A & \xrightarrow{\alpha} & FA \\ {\scriptstyle \mathsf{ana}_\xi}\,\vdots & & \uparrow\, {\scriptstyle F\,\mathsf{ana}_\xi} \\ X & \xrightarrow[\xi]{} & FX \end{array} \qquad \alpha \circ \mathsf{ana}_\xi = F\,\mathsf{ana}_\xi \circ \xi.$$

The function is called ana_ξ because it is sometimes called the anamorphism *for the coalgebra ξ.*

This definition means that we can define a function into a coinductive type by giving a coalgebra. For example, consider the data type of a pure stream 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 A, which in this case is $\mathbb{N}$. 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 using a 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.) The anamorphism $\mathsf{fr} = \mathsf{ana}_\xi : \mathbb{N} \to \mathbb{S}_\mathbb{N}$ maps n to the stream starting at n.

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

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

Here the presence of the parameter n and the argument of the recursive call $n+1$ implicitly give the coalgebra $n \mapsto \langle n, n+1 \rangle$. This is a general programming pattern: we specify a function by equations that directly give the head of the resulting stream, and indirectly determine the tail by applying recursively the function at the top of the second argument of the constructor $\lhd$. We say that the recursive call is *guarded by the constructor*. When this happens we can always find a coalgebra that justifies the definition.

In practical programming we can use a more liberal methodology, coding programs by equations that are not strictly guarded by constructors, but can be reduced to that form (or directly to coalgebra form) by some standard transformation. For example, a different way of defining the function fr is:

$$\begin{aligned} &\mathsf{fr}' : \mathbb{N} \to \mathbb{S}_\mathbb{N} \\ &\mathsf{fr}'\, n = n \lhd (\mathsf{fr}'\, n \oplus \bar{1}) \end{aligned}$$

where $\oplus$ is the pointwise addition of streams and $\bar{1}$ is the constant stream of ones:

$$\begin{aligned} &(\oplus) : \mathbb{S}_\mathbb{N} \to \mathbb{S}_\mathbb{N} \to \mathbb{S}_\mathbb{N} && \bar{\cdot} : A \to \mathbb{S}_A \\ &\sigma_1 \oplus \sigma_2 = (\mathsf{head}\, \sigma_1 + \mathsf{head}\, \sigma_2) \lhd (\mathsf{tail}\, \sigma_1 \oplus \mathsf{tail}\, \sigma_2) && \bar{x} = x \lhd \bar{x} \end{aligned}$$

Notice that, while the definitions of $\oplus$ and $\bar{\cdot}$ are correctly guarded by constructors, the definition of fr' is not strictly guarded: the recursive call doesn't occur immediately at the top of the second argument of the constructor $\lhd$, but instead as an argument of the $\oplus$ operator. However, it is relatively easy to modify the definition to make it comply with the strict guardedness condition [3]. We will allow ourselves to use this more lax definition style.

A class of functors for which the existence of a final coalgebra is guaranteed is that of *containers* [1]: they are a generalization of tree constructors in which any type can be used for branching; this leads to generalized types of non-well-founded trees.

Guardedness by constructors is common good criterion for the acceptability of recursive definitions with a final coalgebra as its codomain: it accepts any recursive equations in which the right-hand side is a term with a constructor on top and recursive calls occurring only as direct arguments of that constructor.

See previous survey work [3] for an overview of the theory of final coalgebras, coinductive types, and corecursive definitions.

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_M is a container if M is [4]. 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).

The idea of a container functor F is that, given a type X, the elements of $F\,X$ are built by constructors that specify a *shape* together with *positions* inside the shape where elements of X can be inserted. For example, the functor Maybe has two shapes: Nothing, with not positions at all, and Just, with a single position. Another example is the List functor: we can see it as having an infinite number of shapes: one shape for every natural number n, with n positions:

$$\begin{array}{ll} \text{shape } n: & [\ \bullet,\ \cdots,\ \bullet\ ] \\ \text{positions:} & \quad\uparrow \qquad\quad \uparrow \end{array}$$

In our case, if M is a container, the functor F_M is a container where the shapes are extended with an element of A paired to every positions. For example, if M is List, the general form of shapes and positions is this:

$$\begin{array}{ll} \text{shape } n: & [\ \langle a_0, \bullet\rangle,\ \cdots,\ \langle a_{n-1}, \bullet\rangle\ ] \\ \text{positions:} & \quad\uparrow \qquad\qquad\quad \uparrow \end{array}$$

The coinductive types we use are always final coalgebras of containers. From now on we 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*. This means that we can define a function by equations that recursively apply the function itself to the elements in the positions of the shapes, and we can prove properties of elements by invoking the statement we want to prove on the elements in the positions.

One final observation is about the distinction between *inductive* and *coinductive* types. We have defined pure streams, lazy lists, monadic streams as coinductive types, using the keyword **codata**. Every container functor also has an inductive type, defined similarly but with the keyword **data**. For example, if we change the keyword in the definition of lazy lists, we obtain the type of finite lists:

$$\begin{array}{l} \textbf{data}\ \mathsf{FList}(A) : \mathsf{Set} \\ \quad (\triangleleft) : A \times \mathsf{FList}(A) \to \mathsf{FList}(A) \\ \quad \mathsf{nil} : \mathsf{FList}(A). \end{array}$$

This change requires every element of $\mathsf{FList}(A)$ to be well-founded, that is, to hit the base case nil after a finite number of steps. Inductive types are much more well known that the coinductive ones, and have been a staple of computer science for decades. The field of coinductive types is instead much younger and still under development.

Inductive types can be characterized dually to coinductive ones as initial algebras of functors. The pattern of definition of function on them is different: we use the common *inductive recursion* style, rather than guardedness.

An inductive type is always a subset of the coinductive type for the same functor (finite lists are contained in lazy lists), but the coinductive type contains extra infinite elements.

It is strictly inadmissible to confuse inductive and coinductive types: they have different and inconsistent ways of defining functions and of reasoning about them (see the next section on *coinduction*). However, Haskell doesn't distinguish between the two: it has no **codata** keyword and the **data** definitions define types with potentially infinite elements. So **data** in Haskell really means **codata**. However, Haskell allows us to define functions on this types by inductive recursion (or by using folds), which would strictly be allowed only for well-founded data.

This liberality works well for programming, but may create inconsistencies when we want to prove properties of our programs. Monsters are defined in Haskell as:

```
data MonStr m a = MCons (m (a , MonStr m a))
```

We will treat this as a coinductive definition and only define functions on them using the guardedness-by-constructors paradigm.

3 The Coinduction Principle

Inductive types comply with an *induction principle*, which states that we can prove statements about them by bottom-up recursion on their structure. Dually, coinductive types comply with a *coinduction principle*, which states that we can prove equalities of their elements by top-down *co-recursion* on their structure.

Let us illustrate the idea with the example of pure streams: Suppose that we want to prove that two streams σ_0 and σ_1 are equal. Since streams are infinite sequences of elements, that will require proving equality of their entries in corresponding positions: if $\sigma_0 = a_0 \lhd a_1 \lhd a_2 \lhd \cdots$ and $\sigma_1 = b_0 \lhd b_1 \lhd b_2 \lhd \cdots$, we must prove $a_0 = b_0$, $a_1 = b_1$, $a_2 = b_2$, and so on (we assume equality on streams is extensional). This requires an infinite sequence of equalities to prove, and clearly we cannot produce all of these explicitly. However, the proof of equality can be seen itself as a stream of proofs of equalities of each pair of elements in the same positions. We can use the same principle of guarded recursion to generate all the proofs: we recursively assume that we can prove the equality of the tail streams and we only need to give explicitly the equality of the heads.

More specifically, we often need to prove that two *functions* produce equal streams. Suppose $f, g : X \to \mathbb{S}_A$ are the two functions, and they are both defined by guarded recursion (that is, defined by coalgebras):

$$f\,x = h_f(x) \lhd f\,(t_f(x)) \qquad g\,x = h_g(x) \lhd f\,(t_g(x))$$

So f is defined by the coalgebra $\langle h_f, t_f \rangle : X \to A \times X$ and g is defined by the coalgebra $\langle h_g, t_g \rangle : X \to A \times X$. We want to prove that the two functions are extensionally equal; let us give a name to the proof of that statement:

$$H : \forall x : X, f\,x = g\,x$$

To prove this statement we can: first, show directly that the heads must be equal, $p_h : h_f(x) = h_g(x)$; then invoke the *coinduction hypothesis* H for the tails. However, since the recursive calls are applied to potentially different elements of X ($t_f(x)$ and $t_g(x)$), we can't apply H directly.

We can get around this problem by generalizing our goal: instead of proving that the functions give the same result when applied to the same input, we aim for the stronger statement that they give the same result when applied to inputs related by a certain binary relation $\sim$:

$$H : \forall x\, y : X, x \sim y \to f\, x = g\, y.$$

If $\sim$ is reflexive, this will imply our original goal.

Two other properties are necessary to make things work. First, we need the equality of the heads of the output: if $x \sim y$, then $h_f(x) = h_g(y)$. Second, we need that $\sim$ is preseved under taking the tail functions for f and g: if $x \sim y$, then $t_f(x) \sim t_g(y)$; so that we can now apply H to obtain that $f(t_f(x)) = g(t_g(y))$.

A relation $\sim$ satisfying these two properties is called a *bisimulation* between the coalgebras $\langle h_f, t_f\rangle$ and $\langle h_g, t_g\rangle$. The principle of coinduction states that bisimilar elements generate equal streams.

This notion can be generalized to coalgebras of any functor F. To formulate it we need to define the *lifting* of a relation by the functor F: if $\sim$ is a relation on a type X, we want to lift it to a relation $\overset{F}{\sim}$ on $F\,X$. A simple way to do it is to apply F to the set-theoretic characterization of the relation: the set of all pairs that satisfy it, $R = \{\langle x, y\rangle \mid x \sim y\}$. We can then use the functorial lift of the projections $\pi_0, \pi_1 : R \to X$ to say that two elements $u, w : F\,X$ are related, $u \overset{F}{\sim} w$, if there exists an element $s : F\,R$ such that $F\,\pi_0\,s = u$ and $F\,\pi_1\,s = w$. This can be further generalized by allowing R to be any type: a collection of *receipts* that certify that pairs of elements are related. In category theory, this notion is called a *span*.

Definition 2. *Let A be a type; a* span *on A is a triple $\langle R, r_1, r_2\rangle$ where R is a type and r_1, r_2 are functions $R \to A$. If F is a functor, the* lifting of the span R by F *is the span $\langle F\,R, F\,r_1, F\,r_2\rangle$ on $F\,A$.*

Definition 3. *Let $\langle A, \alpha\rangle$ be a coalgebra. A span $\langle R, r_1, r_2\rangle$ is a* bisimulation *if there exists a morphism $\rho : R \to FR$ such that both r_1 and r_2 are coalgebra morphisms from $\langle R, \rho\rangle$ to $\langle A, \alpha\rangle$:*

$$\begin{array}{ccc} & \overset{r_1}{\longrightarrow} & \\ R & \rightrightarrows & A \\ & \underset{r_2}{\longrightarrow} & \\ \rho \downarrow & & \downarrow \alpha \\ & \overset{Fr_1}{\longrightarrow} & \\ FR & \rightrightarrows & FA \\ & \underset{Fr_2}{\longrightarrow} & \end{array} \qquad \begin{aligned} \alpha \circ r_1 &= Fr_1 \circ \rho \\ \alpha \circ r_2 &= Fr_2 \circ \rho. \end{aligned}$$

(The diagram here is used only to declare the type of the morphisms: we don't assume that it commutes. The only equalities are the ones stated on the right.)

The idea is that a relation is a bisimulation if, whenever two elements of A are related by it, then their images through α are related by the lifting. If F is a container and you think of α as giving the structure of an element this says: if two elements are related, then they must have the same shape, with components in corresponding positions also related. This notion of bisimulation was first introduced by Park [14] and Milner [13] as a way of reasoning about processes. Similar concepts were developed earlier in other fields and substantial previous work prepared the background for its appearance. The survey article by Sangiorgi [20] tells the history of the idea. Aczel [2] adopted it as the appropriate notion of equality for non-well-founded sets. There are subtle differences between several notions of bisimulation that are not equivalent in full generality: recent work by Staton [22] investigates their correlations.
On $\mathbb{S}_A$ a bisimulation is a binary relation $\sim$ such that:

$$\forall \sigma_1, \sigma_2 : \mathbb{S}_A . \sigma_1 \sim \sigma_2 \Rightarrow \mathsf{head}\, \sigma_1 = \mathsf{head}\, \sigma_2 \wedge \mathsf{tail}\, \sigma_1 \sim \mathsf{tail}\, \sigma_2$$

Notice that $\sigma_1 \sim \sigma_2$ guarantees that corresponding elements in the infinite sequences defined by σ_1 and σ_2 are equal, that is, σ_1 and σ_2 are extensionally equal. In fact, by repeatedly applying the above property, we have that $\mathsf{head}\, (\mathsf{tail}^n\, \sigma_1) = \mathsf{head}\, (\mathsf{tail}^n\, \sigma_2)$ for every n.

Definition 4. *A coalgebra $\langle A, \alpha \rangle$ is said to satisfy the* coinduction principle *if for every bisimulation $\langle R, r_1, r_2 \rangle$ we have that $r_1 = r_2$ extensionally. That is, all pairs related by the bisimulation are equal.*

Intuitively, the coinduction principle states that the elements of A are completely characterised by their structure, which can be infinite. There is a well-known connection between finality of a coalgebra and the coinduction principle.

Theorem 5. *Final coalgebras satisfy the coinduction principle.*

In practice the coinduction principle is applied by a corecursive proof that can invoke the statement to be proved under certain structural restrictions. When proving the equality of two given terms, we can appeal to the statement that we want to prove, as a *coinduction hypothesis*, as long as it is *guarded by constructors* in the sense that it is only deployed to prove the equality of direct components of the given terms.

As an example, let us use the coinduction principle to prove that the two versions of the function fr defined in Section 2 are equal, that is, $\mathsf{fr}\, n = \mathsf{fr}'\, n$ for all natural numbers n. We will first need a lemma about fr', basically saying that it satisfies the tail equaltion of fr, which we also prove by coinduction.

Lemma 6. *For all natural numbers n, $\mathsf{tail}\, (\mathsf{fr}'\, n) = \mathsf{fr}'\, (n + 1)$.*

Proof. We unfold the two sides of the equality, simplifying according to the

definitions of fr', $\oplus$, and $\bar{1}$:

$$\begin{aligned}
\mathsf{tail}\,(\mathsf{fr}'\,n) &= \mathsf{tail}\,(n \lhd (\mathsf{fr}'\,n \oplus \bar{1})) \\
&= \mathsf{fr}'\,n \oplus \bar{1} \\
&= (\mathsf{head}\,(\mathsf{fr}'\,n) + 1) \lhd (\mathsf{tail}\,(\mathsf{fr}'\,n) \oplus \bar{1}) \\
&= (n+1) \lhd (\mathsf{tail}\,(\mathsf{fr}'\,n) \oplus \bar{1}) \\
\mathsf{fr}'\,(n+1) &= (n+1) \lhd (\mathsf{fr}'\,(n+1) \oplus \bar{1})
\end{aligned}$$

The two expressions have the same head, $n+1$, so we just need to prove that the tails are equal. For this we can simply invoke the *coinductive hypothesis*, that is, apply the statement of the lemma $\mathsf{tail}\,(\mathsf{fr}'\,n) = \mathsf{fr}'\,(n+1)$. □

To readers unfamiliar with this style of reasoning, this proof may seem hopelessly circular: we invoke the statement that we're trying to prove in its own proof. However, the circular appeal to the statement is restricted to proving equality of the tails: it is guarded by constructors. This constraint on the structure of the proof makes it ameanable to be reformulated more rigorously in the form of a bisimulation. (Strictly speaking, the coinductive hypothesis is applied to terms that are arguments of $\oplus$, so it is not technically guarded; but as in the case of the definition of fr', this lax use can be readily justified.)

We can now use the lemma to prove that both versions of the function are equal.

Theorem 7. *For all natural numbers n, $\mathsf{fr}\,n = \mathsf{fr}'\,n$.*

Proof. We unfold the left-hand side of the equation, then use the coinductive hypothesis and the previous lemma to reduce it to the right-hand side.

$$\begin{aligned}
\mathsf{fr}\,n &= n \lhd \mathsf{fr}\,(n+1) \\
&= n \lhd \mathsf{fr}'\,(n+1) && \text{by Coinductive Hypothesis} \\
&= n \lhd \mathsf{tail}\,(\mathsf{fr}'\,n) && \text{by Lemma 6} \\
&= \mathsf{fr}'\,n
\end{aligned}$$

The last step is justified because $n = \mathsf{head}\,(\mathsf{fr}'\,n)$ by definition and trivially any stream σ is equal to $\mathsf{head}\,\sigma \lhd \mathsf{tail}\,\sigma$. □

When working with monadic streams, we can similarly use proof by coinduction by invoking as coinductive hypothesis the statement we are proving. Uses of the coinductive hypothesis are justified if they are applied to the *subcomponents* of the stream.

We will illustrate this when we prove statements about some of the instantiations of monadic streams. For example, when using List-monsters (monadic streams where the underlying monad M is List), we are allowed to apply the coinductive hypothesis to the tails of all elements in the list.

4 Examples

We now shift our view to concrete examples of monsters, and their potential uses.

By instantiating monadic streams with different monads, you can form well-known data types, or variations of these. This has provided good insight into what kinds of operations are possible to perform on monadic streams in general.

For each one of these instances, we look at how they correspond to other related data structures. In the next section, we also investigate how a few generic functions on monadic streams act on each of these instances, with respect to how they transform the data structures that these monsters represent.

All of these instances of monadic streams, and some of their related operations, are implemented in the attached library. For the most part we will consider operations that are polymorphic on the monad, as these are more novel in their behaviour.

4.1 Identity monad

To recap, when instantiating $\mathbb{S}_{M,A}$ with the identity monad, we obtain the type of pure streams.

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

All standard operations on streams can be implemented for this type.

Interestingly, pure streams are also comonads, and this is true in general for any monadic stream where the underlying functor has a comonad structure. This is discussed at the end of this section, where we look at the example of a Store-monster.

4.2 Maybe monad

When the underlying functor is Maybe, we get the type of lazy lists

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

In Haskell, this type is implemented as the standard list type `[a]` for any type `a`. The nil constructor is just the empty list `[]`, and the $\triangleleft$ constructor is represented by the operator `(:)`. So the list $a_0 \triangleleft a_1 \triangleleft a_2 \triangleleft \mathsf{nil}$ is written in Haskell as either `a0:a1:a2:[]` or, with the nicer list notation, as `[a0,a1,a2]`.

The type $\mathbb{S}_{\mathsf{Maybe},A}$ of Maybe-monsters is isomorphic to the type $\mathsf{List}(A)$ of lazy lists, with every operation possible on lists also possible on Maybe-monsters. And indeed most operations can be generalized to polymorphic functions independent of the monad.

This isomorphism is witnessed by the two functions:

```
fromL :: [a] → MonStr Maybe a
fromL [] = Nothing
fromL (x:xs) = MCons (Just (x, fromL xs))

toList :: MonStr Maybe a → [a]
toList (MCons Nothing) = []
toList (MCons Just (x, xs)) = x : (toList xs)
```

It is clear from the definitions that these functions are the exact inverse of one another.

This isomorphism has provided a useful benchmark against which to test monad-polymorphic functions in our library. For each function in `Data.List` (a collection of list operations included in Haskell's standard library) that has been generalized to monadic streams, we can test this generalized function with a Maybe-monster, and compare the output to that of the corresponding function in `Data.List`. For this, we use the QuickCheck [5] library:

```
prop_drop :: Property
prop_drop = forAll (genListMonStr >*< chooseInt (0,1000)) $
              λ((l, ms), n) → drop n l == toList (dropM n ms)
```

This function builds a QuickCheck property, which validates whether the drop function (removal of the first n elements) on lists and Maybe-monsters are (extensionally) equivalent. This makes use of the isomorphism, allowing us to freely convert between these types in order to check for equality.

Included with the library is a whole suite of tests in this style, showing the correspondence between functions on Maybe-monsters and functions on lazy lists defined in the Haskell standard library.

All standard (and many non-standard) operations on lists are implemented for generic monads, sometimes with the requirement that the functor is an instance of `Foldable` or `Alternative`.

4.3 List monad

When instantiating monsters with the list monad, we get the type of branch-labelled trees. These are a variation on the usual type of Rose trees, where in this case the *edges*, rather than the nodes, of the trees are labelled by values.

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

(Note that this type has an empty element $\mathsf{leaf} : \mathsf{BLTree}(A)$, obtained by using the empty list, $\mathsf{leaf} = \mathsf{node}\,\mathsf{nil}$.)

Branch-labelled trees with one extra 'root' element are isomorphic to Rose trees, as shown below - this corresponds to the idea that every node in a tree has a unique branch from its parent, with the exception of the root node.

```
data RoseTree a = RNode a [RoseTree a]

phi :: (a, MonStr [] a) → RoseTree a
phi (a , MCons ts) = RNode a (fmap phi ts)

psi :: RoseTree a → (a, MonStr [] a)
psi (RNode a ts) = (a , MCons (fmap psi ts))
```

Branch-labelled trees in general are more useful for applications where you care only about *paths* down a tree, or *transitions* between states rather than the states themselves.

As an example, we could either model a game of noughts and crosses as a Rose tree where each node stores a game state, or with a List-monster where each branch stores the move taken. It is clear from this that you can have an empty branch-labelled tree (a single node with no branches/moves), but not an empty Rose tree (there is always an initial game state, even if no move has been made yet).

Another example is probability trees - the branches represent choices, and the labels are the probabilities of those choices occurring. This corresponds nicely to the non-determinism semantics of the list monad. Included in the attached code are some operations on and examples of List-monsters, with this concept in mind.

Some standard operations on trees don't work for this variation, but traversal operations return the branch labels in the expected order, with the convention of traversing left-to-right.

4.4 Reader monad

Yet another interesting type is that of Reader-monsters. The reader monad is simply an operator that generates the function type from a fixed input type I:

$$\mathsf{Reader}_I\, A = I \to A$$

We can see Reader_I-monsters, elements of $\mathbb{S}_{\mathsf{Reader}_I}\, A$, as trees with edges labelled by elements of I and internal nodes labelled by elements of A.

Reader-monsters correspond to the type of Mealy machines.

These are a type of finite state transducer, with a set of states S, input alphabet Σ, output alphabet Δ, an initial state $s_0 \in S$, and a transition function $\delta : S \times \Sigma \to S \times \Delta$. They are typically used to model computations where the outputs depend on both the internal state and an input.

$$\begin{aligned} &\textbf{record}\ \mathsf{Mealy}(S, \Sigma, \Delta) : \mathsf{Set} \\ &\quad \delta : S \times \Sigma \to S \times \Delta \\ &\quad s_0 : S \end{aligned}$$

There are mutual transformations between these two types. These form an equivalence: when transforming a Mealy machine to a Reader-monad and then

back to a Mealy machine, we don't get the same machine (the type of states has changed), but we get a Mealy machine with the same dynamic behaviour.

The direction from Reader-monsters to Mealy machines requires us to define the state type as a hierarchical recursive type from input to output.

```
type SMStr i o = MonStr ((→) i) o

data Mealy st inA outA = Mealy { initState :: st
                               , transf :: (st , inA) → (st , outA)
                               }

data StateFunc i o = SF { getSF :: i → (StateFunc i o, o) }

mealyToMonStr :: Mealy s i o → SMStr i o
mealyToMonStr (Mealy s tf) = MCons (λe → let (s', a) = tf (s, e)
  in (a, mealyToMonStr (Mealy s' tf)))

monStrToMealy :: SMStr i o → Mealy (StateFunc i o) i o
monStrToMealy (MCons f) = Mealy (aux f) (λ(g, e) → (getSF g) e)
  where
    aux :: (e → (a, MonStr ((→) e) a)) → StateFunc e a
    aux f = SF (λe → let (a, g) = f e in (aux (uncons g), a))
```

To clarify, `StateFunc` is equivalent to `SMStr` - the distinction is that we use `StateFunc` when we want to talk in context about the state of the Mealy machine that a `SMStr` corresponds to, and not just a Reader-monster in isolation.

The two functions defined produce extensionally equivalent Mealy machines, which when given the same inputs, produce the same outputs. A Reader-monster can be thought of as a Mealy machine where each state *is* the transition function, more specifically the transition functions partially applied to each state of the implied Mealy machine. From now on we talk about Reader-monsters, finite state machines (FSM) and Mealy machines, all referring to the same data structure.

Interestingly, from Jacobs [9], the type of Mealy machines is a way of defining a coalgebra: the corresponding monster is then the *behaviour* of this coalgebra. See the coinductive treatment of finite state machines in [9]. This particular work could be very relevant in our continued study on monadic streams and their connection to automata.

One function that works well with Reader-monsters is `zipWithA`. This uses a binary operation to combine the elements of two monsters, producing a new monster with these combined elements. In the case of state machines, this amounts to having a pair of state machines where the same inputs are fed to both, and then the outputs are combined with some arbitrary function. Each FSM still changes state independently.

As a trivial example, you could have a FSM that outputs the maximum of the last 3 inputs (where the set of inputs is ordered), and another that

outputs the minimum. If you zip these together with a function that checks for equality, then you have a FSM that outputs 'true' when the last 3 inputs were identical, and 'false' otherwise. This particular example is included in `Examples.StateMachines`.

This function, and others such as `interleaveReadM` (discussed in section 5), give useful ways of building complex FSMs out of smaller ones.

Another type that Reader-monsters correspond to, when the domain of the reader arrows are pairs of time values and another input type, is the type of signal functions as presented in [15]. Perez et al. discuss this correspondence, and in-fact show that monadic streams in general can play a useful role in Functional Reactive Programming (FRP), by modelling streams of inputs into reactive systems. This role, and other applications, are discussed further in section 7.

4.5 State monad

Instantiation of monsters with the state monad gives a type that is similar again to that of Mealy machines, but one that supplies its own input to each computation (after the first) - you can either 'restart' the machine by supplying a fresh state, or let it run with the states that it produces itself. This could be seen as a Mealy machine with feedback, one who's transition function returns a new input. We refer to this as a feedback machine (FBMachine for short):

$$\begin{aligned}&\textbf{record}\ \mathsf{FBMachine}(S, \Sigma, \Delta) : \mathsf{Set}\\ &\quad \delta : S \times \Sigma \to S \times (\Sigma \times \Delta)\\ &\quad s_0 : S\end{aligned}$$

We represent the data type this way to underline that the type of our 'feedback machines' $\mathsf{FBMachine}(S, \Sigma, \Delta)$ is equivalent to $\mathsf{Mealy}(S, \Sigma, \Sigma \times \Delta)$, that of a special case of Reader-monsters.

In this sense, most of what can be said about State-monsters was said in the Reader monad section above. The main difference is in how the stream is traversed, or executed - the state monad's join operation threads the state though each nested computation, whereas a Reader-monster can't do this with the reader monad operations alone.

Another way to think about a State-monster, separate from their interpretation as Mealy machines, is as an intensional unfold of a pure stream of values. Defining a State-monster amounts to defining a function that takes a state, and returns a value, a new state, and a continuation function (another State-monster). When the new state is continuously applied to the next continuation, this calculates an infinite stream of values from an initial 'seed' state - quite similar to the unfold function, also called the stream *co-iterator*.

```
data FBMachine s a = FBM { runFBM :: s → ((a, s), FBMachine s a) }
```

```
unfoldFBM :: FBMachine s a → s → Stream a
unfoldFBM fbm s0 = h <: (unfoldFBM cont s1)
    where ((h, s1), cont) = runFBM fbm s0

unfold :: (s → (a, s)) → s → Stream a
unfold f s0 = h <: (unfold f s1)
    where (h, s1) = f s0
```

The key difference is that the State-monster can contain many *different* functions that produce new states and outputs. This gives a richer language by which to calculate outputs, where each function defines what the next function should be, depending on an input state. This makes them analogous to state machines, where S is the set of functions the monster contains, and Σ is the state values given as inputs.

4.6 IO monad

A monadic stream of IO actions corresponds to a non-terminating process. Due to the nature of the IO monad, these operate quite differently to other monsters.

One interesting property is that an IO-monster is the only kind of non-well founded (infinite) monster, as far as we know, where collapsing the whole stream into one monadic action is possible and meaningful. This is done using the `runProcess` family of functions:

```
type Process a = MonStr IO a

runProcess :: Process a → IO [a]
runProcess (MCons s) = do (a,s') ← s
                          as     ← runProcess s'
                          return (a:as)
```

This is because IO actions can continuously interact with the outside world, and call other functions, *without* having to terminate. The IO action produced by collapsing an IO-monster is the (possibly infinite) sequence of all IO operations in the stream.

However in this sense, it is better not to 'run' a process (IO-monster) at all, and instead just unfold each IO action one at a time. This allows you pass it around, taking of layers of computation as needed:

```
consumeOne :: Show a ⇒ Process a → Process a
consumeOne p = absorbM $ do (a, cont) ← uncons p
                            putStrLn (show a)
                            return cont
```

This intensional representation of processes gives an advantage over just defining a single repeating action with just the IO monad, because you can take intermediate results from an infinite process as needed, whereas you cannot define a single IO action that does this without using lazy IO.

```
consumeOne' :: Show a ⇒ IO [a] → IO [a]
consumeOne' p = join $ do a ← fmap head p
                          putStrLn (show a)
                          return (fmap tail p)
```

The first line of the `do` statement won't return if the IO action `p` is non-terminating and strict.

The following is a technical worked example, presenting alternative ways of running an IO-monster in different contexts:

Given a process (an IO-monster), you may want to use some of the output values in another computation. To access these values, you need to run the IO actions that the monster consists of. However, since the first action will never return (as all of the subsequent actions are nested inside), this is not immediately possible with strict IO. Instead, you have to use lazy IO, so that values that are needed outside of the process are only calculated as required. This requires `unsafeInterleaveIO` [21], which defers execution of IO actions until evaluation of their result is forced by another computation.

In these functions, we are trying to output the first value computed by running a process `proc0`:

```
proc0 :: Process a

run0 :: IO ()
run0 = do as ← runProcess proc0
          putStrLn $ show (as !! 0)

run1 :: IO ()
run1 = do as ← unsafeRunProcess proc0
          putStrLn $ show (as !! 0)
```

Here, `run0` will run the process `proc0`, and the line that prints the first element will never be reached since `runProcess proc0` doesn't terminate.

In `run1`, we use `unsafeRunProcess`, which is defined as:

```
unsafeRunProcess :: Process a → IO [a]
unsafeRunProcess (MCons s) =
  do (a,s') ← s
     as     ← unsafeInterleaveIO (unsafeRunProcess s')
     return (a:as)
```

Each of the IO actions in the process are run with `unsafeInterleaveIO`, allowing the process to return 'early', and the `putStrLn $ show (as !! 0)` statement to be reached. This print statement forces the first IO action in the process to run, since the first element of `as` is needed (`as !! 0` returns the first element of the accumulated values from running the process).

Using `unsafeInterleaveIO` introduces concurrency problems into otherwise relatively pure Haskell programs. For some IO-monsters, such as those that simply output values to the terminal, this is not a big issue, but more complicated

ones may cause problems (for example ones that read and write to a file).

This problem informed the development of a more general operation on monadic streams, `interleaveReadM`, which allows for interleaving two monsters, where each element in the second depends on elements in the first.

```
interleaveReadM :: Monad m ⇒ MonStr m a → MonStr (ReaderT a m) b
                           → MonStr m b
interleaveReadM (MCons ma) (MCons f) = MCons $
  do (a, ma') ← ma
     (b, f')  ← runReaderT f a
     return (b, interleaveReadM ma' f')
```

This is useful for sequencing IO-monsters, without resorting to possibly unsafe methods. It relies on the `ReaderT` monad transformer, monad transformers being something we haven't yet discussed. They are essentially a method of stacking different monads to combine their effects. Here, we are using the `ReaderT a IO` monad, which represents an IO action that is computed from some environment of type `a`.

Shown below is an example of combining two processes with this operation.

```
inputProc :: Process Char
inputProc = MCons $ do c ← getChar
                       return (c, inputProc)

outputProc :: Show a ⇒ MonStr (ReaderT a IO) ()
outputProc = MCons $ do a ← ask
                        liftIO $ putStrLn (show a)
                        return ((), outputProc)

testProc :: IO ()
testProc = runVoidProcess (interleaveReadM inputProc outputProc)
```

Using this function lets you define dependent processes separately and then combine them afterwards. 'Dependent' refers to the use of the `ReaderT` monad transformer: the actions of the dependent process (`outputProc`) are specified by the elements in another process (`inputProc`).

Variations on this function give lots of options for modifying processes. For example, we could write a version of `run1` from earlier using `insertActReadM`, which inserts a dependent monadic action at a given index.

```
run1 :: IO [Int]
run1 = runProcess (insertActReadM 0 (putStrLn ∘ show) proc0)
```

This won't terminate, but prints the first element of the IO-monster like we wanted, by inserting the print function directly into the process.

To stop this process (without resorting to `unsafeInterleaveIO`), we define a variation of `runProcess` which stops unfolding the IO-monster when a given predicate is true:

```
stopAtPred :: (a → Bool) → Process a → IO a
stopAtPred p (MCons s) = do (a, s') ← s
                            if p a then (return a) else (stopAtPred p s')
```

As a side-note, `runProcess` can now be written as:

```
runProcess :: Process a → IO a
runProcess = stopAtPred (λ_ → False)
```

With this, it is now possible to terminate the process given a particular predicate. It is also possible to terminate a process at a particular index in the IO-monster, thus giving exactly the behaviour we wanted for `run1`, without resorting to unsafe IO (this example is included in the attached code, as it is a bit more verbose).

As demonstrated, IO-monsters are tricky, but provide an interesting intensional way of modelling processes, allowing the computations that they consist of to be modified and interleaved after their definition.

4.7 Store comonad

The store comonad (see section 6.3 for the definition of a comonad) is a pair of a state (called the store) and a function to extract a value from that state:

```
data Store s a = Store (s → a) s
```

A Store-monster, a monadic stream using the store comonad, is a stream where extracting from the store gives a value, and a *new store*. This encapsulates the idea of an environment that can be used to modify itself.

(We again stress that, contrary to the name, the underlying functor in a monadic stream doesn't have to be a monad, and can in-fact be other types of functor, in this case a comonad)

The concept of cellular automata [11] fits this type of monadic stream perfectly, where the state of the environment is calculated from the previous using a rule.

The comonadic interpretation of cellular automata [19] is a good introduction to the motivation behind and uses of comonads. The difference when using monadic streams is that a Store-monster already 'contains' every future state of its environment, so extra functions can be applied lazily to these before they are actually evaluated. By representing the structure intensionally rather than extensionally, you gain an extra level of control.

To show this, there is an implementation of Conway's game of life included in the attached code. This demo uses a kind of Zipper comonad [8] instead of store for some more efficiency, but the type used is isomorphic to `Store (Int,Int) Bool`.

5 General functions and operations

In light of the interpretations of various types of monadic stream, some of the general operations defined have interesting uses in all or some of the contexts presented.

Consider the `dropM` function.

```
dropM :: Monad m ⇒ Int → MonStr m a → MonStr m a
dropM n ms
  | n == 0    = ms
  | n > 0     = dropM (n - 1) (tailM ms)
  | otherwise = error "MonadicStreams.dropM: negative argument."
```

This drops the first n elements of a monster. Doing so requires the joining of the first n monadic actions, which in the case of State-monsters for example will keep the first n state modifications. The *monadic tail function* `tailM` automatically joins the first monadic action into the rest of the monster. This is equivalent to ignoring the output of the feedback machine for its first n feedbacks, so this operation could be seen as moving forwards in time n steps.

This 'moving forwards' interpretation is similar in every kind of monadic stream - in a List-monster, you get the list of choices after n previous choices were made. In an IO-monster you get the output after the first n computations. In a Reader-monster (state machine) you get the output and next state after giving the same input n times. In a Maybe-monster (lazy list), you drop the first n items from the list.

Another interesting function is `scanM`.

```
scanA :: Applicative m ⇒ (a → b → a) → a → MonStr m b → MonStr m a
scanA f z s = z <: scanH f z s
              where scanH f z s = MCons $ (λ(a, s') →
                  (f z a, scanH f (f z a) s')) <$> uncons s

scanM :: Monad m ⇒ (a → a → a) → MonStr m a → MonStr m a
scanM f s = absorbM $ (λ(a, s') → scanA f a s') <$> uncons s
```

It combines each pair of elements in the stream together with a given function, resulting in a stream of the intermediate values. `scanA` also does this, but leaves an extra monadic action at the beginning of the stream, which `scanM` joins with the next action in the monster to remove it. `scanA` can also generate a monster with a different type to the input, which can be seen from the definitions: this is useful when you want to scan using a function who's return type differs from that of its arguments.

In the context of List-monads used as probability trees, `scanM (*)` will (when applied to a tree), return the tree of sequential probabilities. That is, given two consecutive branches of the tree l_1 and l_2 (which represent probabilities), this scan instance calculates $l_1 * l_2$, the probability of traversing the first branch *and*

then the second. The scan carries this result forward, so over three sequential branches l_1, l_2 and l_3, the function would produce the probability $l_1 * l_2 * l_3$ at the third branch in the new tree.

In the same way, `scanM (+)` calculates the disjoint unions of probabilities across each combination of branches in the tree.

A final interesting function is `interleave`. This (unsurprisingly) lets you interleave the actions of two monadic streams.

```
interleave :: Functor m ⇒  MonStr m a → MonStr m a → MonStr m a
interleave mas mbs = transform (λh t → (h, interleave mbs t)) mas
```

In the context of List-monsters again, this function lets you interleave choices - for two List-monsters L_0 and L_1, representing decision trees, interleaving them results in a tree $L_{0,1}$ where the first decision in $L_{0,1}$ is the first in L_0, the second decision is the first in L_1, and so on. This can describe situations where different kinds of choices need to be made cyclically.

In IO-monsters, this function lets you interleave the actions of two process, effectively implementing a naïve kind of user-level multithreading.

In Reader-monsters and State-monsters, a variation of this function, `interleaveReadM` is better in the context of state machines (this function is discussed in the IO monad portion of section 4). Here, this function lets you combine two state machines such that the state and outputs of one depend on the outputs of the other.

```
interleaveReadM :: Monad m ⇒ MonStr m a → MonStr (ReaderT a m) b
                       → MonStr m b
interleaveReadM (MCons ma) (MCons f) =
  MCons $ do (a, ma') ← ma
             (b, f')  ← runReaderT f a
             return (b, interleaveReadM ma' f')
```

This could be extended so that both monsters are dependent on each other, which is left as an exercise to the reader.

The main aim of this library was to define general functions that operate on generic data structures (defined using monadic streams), such that the functions have interesting, or even novel, uses in the contexts presented by different monads or functors. We hope that the functions presented above, alongside the example in section 4, give readers a general sense of this.

6 Instances of Functor, Applicative, Comonad

This section outlines the Functor, Applicative, and Comonad instances, proving that the functions defined on monadic streams to implement these satisfy the relevant laws. We will prove these with a mixture of equational reasoning (using Haskell), and categorical reasoning, in each case making particular assumptions about the underlying functor.

6.1 Functor Instance

To show that $\mathbb{S}_M$ is a functor whenever M is, we have to define its behaviour on morphisms: if $f : A \to B$, then we must define how f maps on monadic streams:

$$\mathbb{S}_M\, f : \mathbb{S}_{M,A} \to \mathbb{S}_{M,B}$$
$$\mathbb{S}_M\, f\, (\mathsf{mcons}\, m) = \mathsf{mcons}\, (M\, (f \times \mathbb{S}_M\, f)\, m)$$

This definition complies with the *guarded-by-constructors* discipline: the recursive call to $(\mathbb{S}_M\, f)$ is mapped to the recursive substreams by the functorial application of the functor $M\, (A \times -)$. That is: $(\mathbb{S}_M\, f)$ will be recursively applied only at the recursive positions inside the shape of m. There are also applications of f to the first element (of type A) of the pairs in the M-position: this is non-recursive, and therefore not problematic.

The Haskell version of the functorial mapping uses a general stream transformer at the top level: `transformM` maps between `MonStr m a` and `MonStr m b` by mapping through `m` a function on the components:

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

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

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

We can now prove that the functor laws are satisfied by $\mathbb{S}_M$: it's functorial mapping preserves identities and composition. The proofs are straightforward applications of definitions and the functoriality of M and $\times$, except for the use of coinduction; we are allowed to invoke the laws themselves in their proofs, as long as we use them only in the direct recursive subterms of the mcons constructor, that is, in the *positions* for the container $M\, (A \times -)$.

Lemma 8. *The identity functor law holds for monadic streams:*

$$\mathbb{S}_M\, \mathsf{id}_A = \mathsf{id}_{\mathbb{S}_{M,A}}$$

Proof. We apply the left-hand side function to an M-monster in constructor form:

$$\begin{array}{ll}
\mathbb{S}_M\,\mathsf{id}_A\,(\mathsf{mcons}\,m) & \\
= \mathsf{mcons}\,(M\,(\mathsf{id}_A \times \mathbb{S}_M\,\mathsf{id}_A)\,m) & \text{by definition} \\
= \mathsf{mcons}\,(M\,(\mathsf{id}_A \times \mathsf{id}_{\mathbb{S}_{M,A}})\,m) & \text{by coinduction hypothesis} \\
= \mathsf{mcons}\,(M\,(\mathsf{id}_{A\times\mathbb{S}_{M,A}})\,m) & \text{by functoriality of } \times \\
= \mathsf{mcons}\,(\mathsf{id}_{M\,(A\times\mathbb{S}_{M,A})}\,m) & \text{by functoriality of } M \\
= \mathsf{mcons}\,m &
\end{array}$$

□

Lemma 9. *The composition functor law holds for monadic streams:*
If $f : A \to B$ and $g : B \to C$, then

$$\mathbb{S}_M\,(g \circ f) = (\mathbb{S}_M\,g) \circ (\mathbb{S}_M\,f)$$

Proof. Let's again apply the left-hand side function to an M-monster in constructor form:

$$\begin{array}{ll}
\mathbb{S}_M\,(g \circ f)\,(\mathsf{mcons}\,m) & \\
= \mathsf{mcons}\,(M\,((g \circ f) \times \mathbb{S}_M\,(g \circ f))\,m) & \text{by definition of } \mathbb{S}_M \text{ mapping} \\
= \mathsf{mcons}\,(M\,((g \circ f) \times ((\mathbb{S}_M\,g) \circ (\mathbb{S}_M\,f)))\,m) & \text{by coinduction hypothesis} \\
= \mathsf{mcons}\,(M\,((g \times \mathbb{S}_M\,g) \circ (f \times \mathbb{S}_M\,f))\,m) & \text{by functoriality of } \times \\
= \mathsf{mcons}\,((M\,(g \times \mathbb{S}_M\,g) \circ M\,(f \times \mathbb{S}_M\,f))\,m) & \text{by functoriality of } M \\
= \mathsf{mcons}\,(M\,(g \times \mathbb{S}_M\,g)\,(M\,(f \times \mathbb{S}_M\,f)\,m)) & \text{by definition of composition} \\
= \mathbb{S}_M\,g\,(\mathsf{mcons}\,(M\,(f \times \mathbb{S}_M\,f)\,m)) & \text{by definition of } \mathbb{S}_M \text{ mapping} \\
= \mathbb{S}_M\,g\,(\mathbb{S}_M\,f\,(\mathsf{mcons}\,m)) & \text{by definition of } \mathbb{S}_M \text{ mapping} \\
= ((\mathbb{S}_M\,g) \circ (\mathbb{S}_M\,f))\,(\mathsf{mcons}\,m) & \text{by definition of composition}
\end{array}$$

□

We can sum up these results by stating that the monster operator is a functor if the underlying 'monad' is (remember that we are not actually assuming that M is a monad yet, but just a type operator).

Theorem 10. *If M is a functor, $\mathbb{S}_M$ is also a functor.*

6.2 Applicative instance

Applicative functors [12] extend the mapping operation by allowing function sequencing under the functor. The Applicative class has two methods: pure, that injects single values into the functor, and $\circledast$, that applies functions under the functor.

We assume that the type operator M is an applicative functor, that is, it has methods:

$$\begin{array}{l}
\mathsf{pure} : A \to M\,A \\
(\circledast) : M\,(A \to B) \to M\,A \to M\,B
\end{array}$$

satisfying the applicative laws.

A typical use of applicative functors is to apply a function of many arguments to several applicative values. If $g : A_0 \to A_1 \to \cdots \to A_n \to B$ and $m_0 : M\,A_0, m_1 : M\,A_1, \ldots, m_n : M\,A_n$, then:

$$(\mathsf{pure}\,g) \circledast m_0 \circledast m_1 \circledast \cdots \circledast m_n : M\,B$$

In particular, if g is an infix binary operator $(\oplus) : A \to B \to C$, then we use the notation:

$$m_a \mathbin{\widehat{\oplus}} m_b = (\mathsf{pure}\,(\oplus)) \circledast m_a \circledast m_b$$

We will show that $\mathbb{S}_M$ is also applicative. In order to define the methods, we need some auxiliary functions on applicative monsters. First of all, a simplified version of mcons that appends a single value in front of a monster. This in turn uses a similar operator for functors, which appends an M-action to the front of a monster

$$\begin{aligned}&(\triangleleft\!\!:\!:) : M\,A \to \mathbb{S}_M\,A \to \mathbb{S}_M\,A\\ &m \triangleleft\!\!:\!: \sigma = \mathsf{mcons}\,(M\,(\lambda a.\langle a, \sigma\rangle)\,m)\end{aligned}$$

$$\begin{aligned}&(\triangleleft\!\!:) : A \to \mathbb{S}_M\,A \to \mathbb{S}_M\,A\\ &a \triangleleft\!\!: \sigma = (\mathsf{pure}\,a) \triangleleft\!\!:\!: \sigma\end{aligned}$$

In Haskell:

```
(<::) :: Functor m ⇒ m a → MonStr m a → MonStr m a
ma <:: s = MCons (fmap (λa → (a,s)) ma)

(<:) :: Applicative m ⇒ a → MonStr m a → MonStr m a
a <: s = pure a <:: s
```

The pure method for monsters then consists in repeating the same element forever:

$$\begin{aligned}&\mathsf{pure} : A \to \mathbb{S}_M\,A\\ &\mathsf{pure}\,a = a \triangleleft\!\!: \mathsf{pure}\,a\end{aligned}$$

We define the function application method by mapping straight function application on the heads and recursive calls on the tails through functorial and applicative lifting:

$$\begin{aligned}&(\circledast) : \mathbb{S}_M\,(A \to B) \to \mathbb{S}_M\,A \to \mathbb{S}_M\,B\\ &(\mathsf{mcons}\,m_f) \circledast (\mathsf{mcons}\,m_a) = \mathsf{mcons}\,(m_f \mathbin{\widehat{\circledcirc}} m_a)\\ &\qquad \mathsf{where}\ \langle f, \phi\rangle \circledcirc \langle a, \sigma\rangle = \langle f\,a, \phi \circledast \sigma\rangle\end{aligned}$$

This definition recursively applies $\circledast$ indirectly in the second components of the arguments of the $\circledcirc$ operator. This is lifted to $\mathbin{\widehat{\circledcirc}}$, which distributes down through the components of the applicative values m_f and m_a, and finally guarded by the constructor mcons. This guarantees the soundness of the definition according to the *guardedness by constructors* criterion.

Here is the definition of the Applicative instance for monsters in Haskell:

```
transformA :: Applicative m ⇒
              (a → MonStr m a → b → MonStr m b → (c, MonStr m c)) →
              MonStr m a → MonStr m b → MonStr m c
transformA f as bs = MCons $ (λ(a,as') (b,bs') → f a as' b bs')
                             <$> uncons as <*> uncons bs

instance Applicative m ⇒ Applicative (MonStr m) where
  pure a = a <: pure a
  (<*>) = transformA (λf fs a as → (f a, fs <*> as))
```

An applicative instance must satisfy four general laws regulating the interaction of sequencing and pure values and requiring associativity of application. Preliminary investigation and analysis of specific instances suggest that this is the case for monsters. A full proof is one of the goals of future work.

Conjecture 11. *If M is an applicative functor, $\mathbb{S}_M$ is also an applicative functor. That is, the following laws are satisfied, for every $a : A$, $f : A \to B$, $\sigma_a : \mathbb{S}_M\, A$, $\sigma_f : \mathbb{S}_M\, (A \to B)$, $\sigma_g : \mathbb{S}_M\, (B \to C)$:*

$$
\begin{aligned}
&(\mathsf{pure}\,\mathsf{id}) \circledast \sigma_a = \sigma_a \\
&\mathsf{pure}\,(f\,a) = (\mathsf{pure}\,f) \circledast (\mathsf{pure}\,a) \\
&\sigma_f \circledast (\mathsf{pure}\,a) = (\mathsf{pure}\,(\lambda f.f\,a)) \circledast \sigma_f \\
&\sigma_g \circledast (\sigma_f \circledast \sigma_a) = (\sigma_g \mathbin{\widehat{\circ}} \sigma_f) \circledast \sigma_a
\end{aligned}
$$

6.3 Comonad instance in Haskell

Comonads (in cojoin form) are functors with two additional operations: η that extracts an element from the functor, and μ, that 'duplicates' a functor:

$$
\begin{aligned}
&\eta : W\,A \to A \\
&\mu : W\,A \to W\,(W\,A)
\end{aligned}
$$

For this to constitute a comonad, these operations need to satisfy the comonad laws.

These operations are expressed in Haskell with the `extract` and `duplicate` functions of the `Comonad` type class:

```
class Functor w ⇒ Comonad w where
  extract :: w a → a
  duplicate :: w a → w (w a)
```

We believe that monadic streams are comonads, when the underlying functor is a comonad.

In the following Haskell equations, the function `head` returns the first element of the first pair in the monster, wrapped in the underlying functor. We require the underlying functor to be a comonad, so that we can use `extract` to remove the functor wrapping.

The proposed `Comonad` instance for monadic streams is as follows.

```
head :: Functor m ⇒ MonStr m a → m a
head = fmap fst ∘ uncons

instance Comonad w ⇒ Comonad (MonStr w) where
  -- extract :: w a → a
  extract = extract ∘ head
  -- duplicate :: MonStr w a → MonStr w (MonStr w a)
  duplicate ms = MCons $ fmap (λ(h,t) → (ms, duplicate t)) (uncons ms)
```

The `duplicate` function for monadic streams forms a "monster matrix". This is a monadic stream where each element is a monadic stream. In the following proof sketches, we represent this as `mm(ms)` where `mm(ms) = duplicate ms`.

Intuitively, `mm(ms)` is a monster where the first element is `ms`, the second is the tail of `ms`, the third is the tail of the tail of `ms`, and so on.

6.3.1 Comonad law proof sketches

Sketch of proof that `extract . duplicate == id`

```
extract ∘ duplicate
= λms → extract (MCons $ fmap (λ(h,t) → (ms, duplicate t))
  (uncons ms))
= λms → extract (head (MCons $ fmap (λ(h,t) → (ms, duplicate t))
  (uncons ms)))
= λms → ms
= id
```

Sketch of proof that `fmap extract . duplicate == id`

```
fmap extract ∘ duplicate = λms → fmap extract (MCons $ fmap (λ(h,t) →
  (ms, duplicate t)) (uncons ms))
= λms → transformM (λa s → (extract a, fmap extract s)) (MCons $
  fmap (λ(h,t) → (ms, duplicate t)) (uncons ms))
  [transformM definition]
= λms → MCons $ fmap (λ(h,t) → (extract h, fmap extract t))
  (fmap (λ(h,t) → (ms, duplicate t)) (uncons ms))
= λms → MCons $ fmap (λ(h,t) → (extract ms, fmap extract (duplicate t)))
  (uncons ms)
  [coinductive hypothesis]
= λms → MCons $ fmap (λ(h,t) → ((extract ∘ head) ms, id t)) (uncons ms)
  [(extract ∘ head) ms = a, first element of ms]
= λms → MCons $ fmap (λ(h,t) → (a, id t)) (uncons ms)
  [a is defined as the first element of ms, so h = a under the fmap]
= λms → ms
= id
```

Sketch of proof that `duplicate . duplicate == fmap duplicate . duplicate`

```
duplicate ∘ duplicate = λms → duplicate (duplicate ms)
= λms → MCons $ fmap (λ(h,t) → (duplicate ms, duplicate t)) $
```

```
  (fmap (λ(h,t) → (ms, duplicate t)) (uncons ms))
= λms → MCons $ fmap ((λ(h,t) → (duplicate ms, duplicate (duplicate t)))
  (uncons ms)

fmap duplicate ∘ duplicate = λms → fmap duplicate (MCons $ fmap (λ(h,t) →
  (ms, duplicate t)) (uncons ms))
= λms → transformM (λa s → (duplicate a, fmap duplicate s)) $
  (MCons $ fmap (λ(h,t) → (ms, duplicate t)) (uncons ms))
= λms → MCons $ fmap (λ(h,t) → (λa s →
  (duplicate a, fmap duplicate s)) h t) (uncons (MCons $ fmap (λ(h,t) →
  (ms, duplicate t)) (uncons ms)))
= λms → MCons $ fmap (λ(h,t) → (duplicate h, fmap duplicate t)) $
  (fmap (λ(h,t) → (ms, duplicate t)) (uncons ms))
= λms → MCons $ fmap (λ(h,t) →
  (duplicate ms, fmap duplicate (duplicate t))) (uncons ms)
  [coinductive hypothesis]
= λms → MCons $ fmap ((λ(h,t) →
  (duplicate ms, duplicate (duplicate t))) (uncons ms)
= duplicate ∘ duplicate
```

As with the applicative functor laws, we are looking to improve these proofs, and also to formulate them in more categorical terms.

6.4 Monad Counter-examples

It's natural to ask, given the chosen name for this data structure, whether monsters themselves are monads, at least when the underlying functor is a monad. This is also a natural question because pure streams (with no extra functor guarding the elements) are monads. However, this turns out not to be the case in general (we strongly believe), and we suspect that more constraints need to be placed on the underlying monad for the monster itself to be a monad. Namely, we believe the underlying monad should be idempotent.

A monad (in join form) is a functor with two additional operations: return that trivially injects an element into the functor, and join that flattens a doubly nested monadic functor into a single layer:

$$\begin{aligned} &\mathsf{return} : A \to M\ A \\ &\mathsf{join} : M\ (M\ A) \to M\ A \end{aligned}$$

For this to constitute a monad, these operations need to satisfy the monad laws (we introduce these as needed). In Haskell, a monad can be defined with the **Monad** type class:

```
class Functor m ⇒ Monad m where
  return :: a → m a
  (>>=) :: m a → (a → m a) → m a
```

For monadic streams to be a monad, we have to be able to inject a pure value into a monster. This is done using **return**. It is clear that the only possible implementation of **return** for monadic streams is the same as that of pure in the applicative functor definition from earlier.

```
return :: Monad m ⇒ a → MonStr m a
return a = MCons $ fmap (λa → (a, return a) (return a))
```

We also need to be able to 'flatten' a monster of monsters into a single monster, with respect to certain laws - this is done in Haskell using a function >>=, pronounced 'bind', its type defined above. This flattening is done explicitly in the function **join** :: MonStr m (MonStr m a) → MonStr m a: it is always the case that (ma >>= f) = **join** (fmap f ma) in Haskell. This identity lets us prove properties of bind by proving properties of join, and visa-versa, as **join** ma = ma >>= **id**.

In these following counter-examples, we are looking to show that three naïve ways of defining the join operation do not work. These cases may or may not cover all possibilities - this is something we have yet to prove.

We will omit the type signature for join in the examples: it is given in the previous paragraph. We also refer to the monsters which are elements of a monster of monsters, as *inner* monsters.

6.4.1 Case 1

In this case, we look at the definition of join as taking the 'horizontal' - the first element of each inner stream:

```
join (MCons mma) = MCons $ do (mas, mma') ← mma
                              fmap (λa → (a, join mma')) (head mas)
```

However, this definition of join violates the left identity monad law, which states that the following equality must hold:

```
a :: a
f :: a → MonStr m a
return a >>= f == f a
```

A witness to this is the monadic stream:

```
fromStep :: Int → MonStr (State Int) Int
fromStep n = MCons (state (λx → ((x, fromStep (n+1)), x+n)))
```

This State-monster, when run (see section 4.4 and 4.5) with an initial number n, generates the stream of integers where the first two differ by n, the next two differ by $n + 1$, and so on.

Running fromStep 1 with 1 produces the stream:

```
runFBStr (fromStep 1) 1 = 1 <: 2 <: 4 <: 7 <: 11 <: ...
```

Running **return** 1 >>= fromStep with 1 produces the stream:

```
runFBStr (return 1 >>= fromStep) 1 = 1 <: 2 <: 3 <: 4 <: 5 <: ...
```

These are different, so this `join` instance doesn't satisfy the left identity monad law.

Intuitively, we cannot take just the head of each stream, because the first action in each inner monster is 'adding 1' - we end up with a stream where 1 is added to the the previous element to get the next, which wasn't the definition of `fromStep 1`.

6.4.2 Case 2

The next possible definition takes the 'vertical' - just the first inner monster:

```
absorbM :: Monad m ⇒ m (MonStr m a) → MonStr m a
absorbM = MCons ∘ join ∘ fmap uncons

join = absorbM ∘ head
```

The function `absorbM` is defined in the library, and simply absorbs a monadic action from outside of a monadic stream. The `join` function used to define `absorbM` is that of the underlying monad.

This definition violates the right identity monad law, which states:

```
ma :: MonStr m a
ma >>= return == ma
```

A witness to this is the pure stream `from n`, defined as

```
from :: Monad m ⇒ Int → MonStr m a
from n = n <: from (n+1)
```

`from 0` gives the stream of natural numbers.

```
from 0 = 0 <: 1 <: 2 <: 3 <: 4 <: ...
```

`from 0 >>= return` using this definition of `join` (taking the first inner monster) gives us a stream of constant zeros:

```
from 0 >>= return = 0 <: 0 <: 0 <: ...
```

Clearly the `join` operation cannot be defined as taking the first element of the monster, or the 'vertical', since `return 0` $\neq$ `from 0`.

6.4.3 Case 3

The final case we consider is taking the 'diagonal'. We define this taking the first element of the first inner monster, the second element of the second inner monster, and so on.

This is defined in Haskell as follows:

```
tailM :: MonStr m a → MonStr m a
tailM = absorbM ∘ tail

join (MCons mma) = MCons $ do (mas, mma') ← mma
                              let (ma, ts) = (head mas, fmap tailM mma')
                                 in fmap (λa → (a, join ts)) ma
```

Similarly to case 1, this definition does not satisfy the left identity monad law, to reiterate:

```
a :: a
f :: a → MonStr m a
return a >>= f == f a
```

The monster `fromStep n` is a witness to this.

Running `fromStep 1` with `1` produces the stream:

```
runFBStr (fromStep 1) 1 = 1 <: 2 <: 4 <: 7 <: 11 <: ...
```

Running `return 1 >>= fromStep` with `1` produces the stream:

```
runFBStr (return 1 >>= fromStep) 1 = 1 <: 3 <: 8 <: 17 <: 31 <: ...
```

The problem seems to be when taking the diagonal, that when we take the nth element from the nth inner stream, we need to join together *all of the n monadic actions in the inner stream up to that element*. This generates different behaviour because the sequencing of the first n actions in a monster is not in general the same as the nth action by itself.

One solution to this could be to only allow the underlying monad to be idempotent [6]. This is a monad that 'squares to itself' - the `join` operation is a natural isomorphism. This isn't the case for this particular instance of the state monad, for example: sequencing two 'add 3' actions results extensionally in an 'add 6' action.

Taking the diagonal should work as a `join` operation for monadic streams with idempotent monads, since any order in which you join the actions results in the same action. However, this is quite a strong constraint, and it would be better if there exists a weaker one.

All of these cases and code testing the left and right identity monad laws are included in the `Test.MonadCounterExamples` module, which can be used to validate the reasoning presented.

These three cases may not be exhaustive, so this section exists as an initial sketch (to be completed at a later date) of why monadic streams are not monads (when the underlying functor is a monad). However, we can say for sure that the `join` operation is not one of the cases considered, which will certainly inform the direction of a concrete proof.

We also seek to prove that if the monad is idempotent, then the corresponding monadic stream is itself a monad.

7 Related work and applications

The general application of this library is hinted at in the examples section: data structures that can be represented as monadic streams (with the underlying functor being a monad) give meaningful semantics to the general operations we have developed.

Perhaps the most interesting correspondence is that of Reader-monsters and finite state transducers. In this context, the functions in our library provide a set of useful combinators to work with and define state machines. A few of these have been mentioned in sections 4 and 5, and some more uses are demonstrated in the attached example code.

7.1 FRP application

One further application of monadic streams is that they can be seen as a generalised form of Yampa's [7] signal functions. This is expanded on by Perez et al. in [15]. For example, you can implement an integral signal function using the Reader monad. 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)
```

Additionally, combinators defined in our library work similarly to combinators in Yampa. For example, their (internal) signal function composition combinator `compPrim` is conceptually the same as our function `interleaveReadM`, which was covered in the IO monad examples section.

Yampa's concept of signal functions is generalised further in Dunai, a library with more direct relationship to monadic streams.

7.2 Relation to Dunai

Dunai [15] [18] is an FRP library implemented using monadic stream functions (MSFs), which Yampa has been implemented on top of (see BearRiver [17]).

The library we've developed in this article allows for a wide variety of operations on monadic streams. Since the MSFs used in Dunai are a special case

of monsters (monsters with the underlying functor `ReaderT a m`, where `m` is a monad), many functions in our library could be incorporated into Dunai, giving a set of potentially useful generalised transformations for MSF networks.

For example, `scanM` could be used to calculate a stream of intermediate velocity values from a stream of position values, by taking the difference of each consecutive position:

```
positions :: SignalFunc Env Double

speeds :: SignalFunc Env Double
speeds = scanM (λa b → b - a) positions
```

Here, `Env` is just a type representing some environment the signal function is operating on.

On the other hand, there are many interesting concepts in Dunai which could be incorporated into new functions in this library, such as their concept of FRP constructs.

These encapsulate the idea of changing `MonStr (ReaderT a (t m)) r` where the underlying monad `m` is lifted with some monad transformer `t`, into `MonStr (ReaderT a m) r` where the monad transformer `t` was 'run' in some way, leaving only the monad `m`. If the monad transformer were `ListT` for example, it would give rise to the concept of parallelism with broadcasting.

Monad transformers were only briefly explored in the library, but this indicates that there may be many more constructs we can form when taking them into account.

Additionally, monadic streams could be a useful extension to MSF based FRP - as shown in [15], monadic streams can already model an input stream of data into a reactive program. Using monadic streams instantiated with comonads, you could model input streams where the value of the input (and the rest of the stream) depends on some environment.

Monadic streams where the underlying functor is a comonad can be seen as the progressing timeline of an environment under some iterated operation. The comonad provides the notion of environment, and the stream imbues it with a notion of its progression in time. This idea only encapsulates current and future values of the environment, with past states being discarded.

This could be useful to model systems which have some kind of self referential data store, or environments that can be used to modify themselves.

One possible barrier to using this in practice is that often inputs into a program, especially in FRP applications, are wrapped in some kind of IO action. IO is not comonadic, as there is no way to extract a pure value from it (without the use of `unsafePerformIO`, but this is unsafe, as the name suggests). In that sense, monadic streams with comonads seem to have limited applicability when

it comes to streams of context-dependent inputs, when those inputs come from outside of the program.

7.3 Dataflow programming

Another purpuse monadic streams with comonads could be used for is extending the comonadic approach to dataflow programming, developed in [23], with the context of other comonads. The idea is that any comonad `w` can be extended with the notion of past, present, and future values with `MonStr w`. Functions out of the type `MonStr m a` are context-sensitive computations that are dependent on arbitrary numbers of future environments of type `w a`.

7.4 Properties of monadic streams

Many properties of monadic stream functions have been proven in the appendix [16] to Bärenz, Perez, and Nilsson's 2016 paper on MSF based FRP [15]. Depending on their formulation, these proofs may also show that the same properties are true for monadic streams, because monadic streams are just MSFs that ignore their inputs.

One of our goals is to investigate further how this work can be applied to the context of monadic streams.

8 Conclusions

This project developed a full library for monadic streams (*monsters*) in Haskell.

We generalised common operations, functions, applications from the standard `List` library and libraries on pure streams to work on all monsters. Applications of our work encompass not only a higher-order version of those libraries, but also new applications to non-well-founded trees, interactive processes, transition systems and automata, functional reactive programming, and more.

We proved several important abstract theorems about monsters and outlined strategies for further mathematical insights. These proofs required original use of the technique of *coinduction*, which is a very recent and active area of research.

At this point in the project, a few new problems have come to light, some related to implementation details of the library, and some to the theory behind monadic streams and corresponding data structures.

Future work on this library will include a review of the efficiency of many functions we defined - this project was initially an exercise to see what could be done with monadic streams, but if this library could be used for larger applications then efficiency would be an issue. Also, whether all of the functions work exactly as intended is still uncertain. The functions that are analogous to those in `Data.List` have been tested, but we have not yet defined some concrete semantics to interpret all of the other functions with, meaning we can only be

sure that their actions on monadic streams are approximately the intended ones.

We discovered, contrary to our initial assumptions, that monadic streams do not in general form a monad, even if the underlying functor does. We have sketched a proof of this, but a concrete argument has yet to be developed. We investigated the causes of this shortcoming and we proposed a conjecture that that monadic streams are themselves monads when the underlying functor is an idempotent monad [6]. This seems evident from how our counter-example sketch is constructed, and it seems to be the case experimentally, but a convincing proof showing that all of the monad laws hold has yet to be developed.

Additionally, our proofs of the applicative and comonad laws need to be developed further, and the comonad proofs formalized.

Another angle of interest is to look further into exactly how monadic streams are different operationally from the cofree comonad over a functor. This is defined as the final coalgebra of $F_M\ X = A \times M\ X$, where M is a functor, which is very similar to the definition of monadic streams in section 2.

The difference is only that there is no first 'unguarded' element in a monadic stream, but this is enough make them function quite differently when instantiated with the same monads.

Finally, whether monadic streams instantiated with comonads could have a good role in functional reactive programming is still an open problem. It may also be interesting to look at a dual of Dunai's monadic stream functions [18]. The dual of monadic streams are monadic lists:

```
data MonLst m a = m (Either a (MonLst m a))
```

Since the type of monadic stream functions is isomorphic to monadic streams with the reader monad transformer, the dual of monadic stream functions might be the following:

```
data MSFDual w e a = MonLst (EnvT e w) a
```

which is essentially a nested tuple, with each layer after the first guarded with a comonadic action.

```
data EnvT e w a = EnvT e (w a)
```

Evaluating the comonad with `extract` or a co-Kleisli arrow will either result in a final tuple of type `(e, a)`, or another layer of type `(e, MonLst (EnvT e w) a)`.

This is like an environment from which new values can be repeatedly extracted from, but at some point the environment will become exhausted, returning a single `a` instead of a new environment.

The purpose of investigating this is to see whether a monad-comonad interaction law [10] between the underlying monad in a MSF and the underlying comonad in a `MSFDual` instance could be used to define a functor-functor interaction law between these two types, and whether this would have any novel uses or implications in the contexts we have discussed.

References

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

[2] P. Aczel. *Non-Well-Founded Sets*. Number 14 in CSLI Lecture Notes. Stanford University, 1988.

[3] V. Capretta. Coalgebras in functional programming and type theory. *Theoretical Computer Science*, 412(38):5006–5024, 2011. CMCS Tenth Anniversary Meeting.

[4] V. Capretta and J. 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.

[5] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, Sept. 2000.

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

[7] A. Courtney, H. Nilsson, and J. Peterson. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, page 7–18, New York, NY, USA, 2003. Association for Computing Machinery.

[8] G. HUET. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[9] B. Jacobs. *Introduction to Coalgebra : Towards Mathematics of States and Observation*. Cambridge University Press, Dec 2016.

[10] S. Katsumata, E. Rivas, and T. Uustalu. Interaction laws of monads and comonads. *CoRR*, abs/1912.13477, 2019.

[11] E. Kozliner. Algorithmic beauty: An introduction to cellular automata, 2019. https://towardsdatascience.com/algorithmic-beauty-an-introduction-to-cellular-automata-f53179b3cf8f.

[12] C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

[13] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[14] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Theoretical Computer Science*, volume 104 of *LNCS*, pages 167–183. Springer-Verlag, 1981. Proceedings of the 5th GI-Conference Karlsruhe.

[15] I. Perez, M. Bärenz, and H. Nilsson. Functional reactive programming, refactored. *SIGPLAN Not.*, 51(12):33–44, Sept. 2016.

[16] I. Perez, M. Bärenz, and H. Nilsson. On the mathematical properties of monadic stream functions. 2016. https://www.cs.nott.ac.uk/~psxip1/papers/msfmathprops.pdf (Unpublished).

[17] I. Perez and M. Bärenz. bearriver: Frp yampa replacement implemented with monadic stream functions. https://hackage.haskell.org/package/bearriver.

[18] I. Perez and M. Bärenz. dunai: Generalised reactive framework supporting classic, arrowized and monadic frp. https://hackage.haskell.org/package/dunai.

[19] D. Piponi. Evaluating cellular automata is comonadic, 2006. http://blog.sigfpe.com/2006/12/evaluating-cellular-automata-is.html.

[20] D. Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4), 2009.

[21] M. Snoyman. Evaluation order and state tokens, 2014. https://www.schoolofhaskell.com/user/snoyberg/general-haskell/advanced/evaluation-order-and-state-tokens.

[22] S. Staton. Relating coalgebraic notions of bisimulation. In A. Kurz, M. Lenisa, and A. Tarlecki, editors, *CALCO*, volume 5728 of *Lecture Notes in Computer Science*, pages 191–205. Springer, 2009.

[23] T. Uustalu and V. Vene. The essence of dataflow programming. pages 135–167, 11 2005.