1 Preliminary ideas

1.1 Definition of $joinPrelimMS$

The aim here is to find a way to manipulate the monadic actions in a monster matrix to find a join function such that the first and second monad laws are satisfied. The two cases below show how a monster matrix is produced in the case of each of these monad laws. This manipulation corresponds to the function $joinPrelimMS$ in MonStreams.hs.

1. fmap f (return a_0)

$$\begin{array}{cccccccc} & M_0(a_0, & & M_0(a_0, & & M_0(a_0, & \\ & M_1(a_1, & & M_1(a_1, & & M_1(a_1, & \\ R_M(& M_2(a_2, & R_M(& M_2(a_2, & R_M(& M_2(a_2, & ... \\ & M_3(a_3, & & M_3(a_3, & & M_3(a_3, & \\ & ... & , & ... & , & ... & , \end{array}$$

2. fmap return ma (where $ma :: MonStr\ m\ a$)

$$\begin{array}{cccccccc} & R_M(a_0, & & R_M(a_1, & & R_M(a_2, & \\ & R_M(a_0, & & R_M(a_1, & & R_M(a_2, & \\ M_0(& R_M(a_0, & M_1(& R_M(a_1, & M_2(& R_M(a_2, & ... \\ & R_M(a_0, & & R_M(a_1, & & R_M(a_2, & \\ & ... & , & ... & , & ... & , \end{array}$$

Now to start "joining" the inner and outer monsters. Taking the diagonal by joining monadic actions in the inner monster at each point in the outer monster gives:

1.

$$R_M(M_0(a_0), R_M(M_1^0(a_1), R_M(M_2^0(a_2), ...$$

2.

$$M_0(R_M(a_0), M_1(R_M(a_1), M_2(R_M(a_2), ...$$

where:

- $M_x(a_x)$ denotes a monadic action in monad M returning a value a, evaluated at position x in its corresponding monadic stream

- $M_y^x(a_y)$ denotes the sequence of monadic actions $M_x(M_{x+1}(...M_y(a_y)...))$, returning the element at index y in the corresponding monadic stream
- R_M denotes the "do nothing" monadic action for monad M - the monadic action produced by return of type $M(a)$

1.2 Going beyond the initial $joinPrelimMS$

This is an attempt to see whether you can manipulate the stream resutling from the previous function (in a different way to $joinInnerMS$ in section 2) to preserve the first two monad laws. The third law isn't currently considered (since the first two are enough trouble at the moment).
Assuming M satisfies the monad laws, you can join R_M with any action $M(a)$ and produce $M(a)$, and this commutes (Kleisli composition forms). Using the notation above, joining $M_y^x(a)$ with $M_{y+1}(a)$ gives $M_{y+1}^x(a)$.
With these, you can "distribute" each monad across its head and tail using $headMS$ and $tailMS$, and then join the nested monadic actions produced by $tailMS$ (this is the function of $tailMMS$, equivalent to ($absorbMS$. $tailMS$)). $fork\ (f, g)\ x$ produces the tuple (fx, gx):

1.

$$fork(headMS, tailMMS)\ \$\ R_M(M_0(a_0), R_M(M_1^0(a_1), R_M(...$$
$$=$$
$$\Big(R_M(M_0(a_0)), R_M(R_M(M_1^0(a_1), R_M(...\Big)$$
$$=$$
$$\Big(M_0(a_0), R_M(M_1^0(a_1), R_M(...\Big)$$

calling this on the second element of the tuple and recursing gives the infinitely nested tuple:

$$\Big(M_0(a_0), \Big(M_1^0(a_1), \Big(M_2^0(a_2), \Big(M_3^0(a_3), ...\Big)$$

2.

$$fork(headMS, tailMMS)\ \$\ M_0(R_M(a_0), M_1(R_M(a_1), M_2(...$$
$$=$$
$$\Big(M_0(R_M(a_0)), M_0(M_1(R_M(a_1), M_2(...\Big)$$
$$=$$
$$\Big(M_0(a_0), M_1^0(R_M(a_1), M_2(...\Big)$$

calling this on the second element of the tuple and recursing gives the infinitely nested tuple:

$$\Big(M_0(a_0), \Big(M_1^0(a_1), \Big(M_2^0(a_2), \Big(M_3^0(a_3), ...\Big)$$

Both of these apply the same sequence of monadic actions M_x^0 when you extract the element a_x.
If you take a monster of type $M(a)$, you can transform it in the same way to produce the same structure:

$$fork(headMS, tailMMS)\ \$\ M_0(a_0, M_1(a_1, M_2(...$$
$$=$$
$$\Big(M_0(a_0), M_0(M_1(a_1, M_2(...\Big)$$
$$=$$
$$\Big(M_0(a_0), M_1^0(a_1, M_2(...\Big)$$

recursively apply the same function on the second element of the tuple, and continue this:

$$\Big(M_0(a_0), \Big(M_1^0(a_1), \Big(M_2^0(a_2), \Big(M_3^0(a_3), ...\Big)$$

In this way, the ”joining” procedure above produces a structure that is the same as the one produced by the distribution procedure on a valid monadic stream. If there was some way of ”factoring” back out the monadic action, then this method of creating a join function would be possible, but this doesn't seem possible.
The ”distributing” step duplicates the monadic action, so recombining the new head and tail would produce a monadic stream with a different behaviour (a different monster, meaning the monad laws wouldn't be satisfied).
In a sense though, the pseudo-monsters produced by this join procedure operate similarly to monadic streams - to retrieve the element a_x in this nested tuple, you still

need to execute monadic actions M_0 to M_x (after applying $snd\ x$ number of times to reach the nested tuple at layer x).
It seems that to finally produce a monadic stream from a monster matrix in the $Monad$ ($MonStr\ m$) typeclass, the monad m used in the monadic stream needs to satify the property that joining two "equivalent" monadic actions should produce the same monadic action, for example:

- $Just\ (Just\ x\ )$ == $Just\ x$

However, counter-examples seem more common:

- If you join a $State$ computation which adds 1 to the state, with another one of the same, you produce a $State$ computation which adds 2 to the state
- Joining two nested IO computations which ask the user for an input, produces an IO computation which asks the user for 2 inputs

2 Cases for each law resulting from use of joinInnerMS

Consecutive monadic actions returning a single element are written without parentheses to reduce clutter.
$joinInnerMS$ is a function which, for a monster $MonStr\ m\ (m\ a)$, moves the tail at each level of the monster inside its monadic element. Basically transforms $M_0(M_a(a), M_1(...$ to $M_0(M_a(a, M_1(...$, by using fmap to insert the monadic stream tail inside the monad of the element (see the definition in MonStream.hs).

2.1 Left identity

$return\ a\ >>=\ f\ ==\ f\ a$
This case results in the follwing call to bind:

$(joinPrelimMS\ .\ joinInnerMS)\ \$\ fmap\ f\ (return\ a)$
$=$
$R_M M_0(a_0, R_M M_0 M_1(a_1, R_M M_0 M_1 M_2(a_2, ...$
$=$ [joining the monadic actions]
$M_0(a_0, M_1^0(a_1, M_2^0(a_2, ...$

In contrast, $f\ a$ where $f\ ::\ (a\ ->\ MonStr\ m\ a)$ returns: $M_0(a_0, M_1(a_1, M_2(a_2, ...,$ so the two sides of the equation are not equal.
To see why imaging taking the second element of both streams:

$\big(M_0(a_0, M_1^0(a_1, M_2^0(a_2, ...)$!!! 1 [!!! is the indexing operator]
$=$
$M_0(M_1^0(a_1)) == M_0 M_0 M_1(a_1)$

$\big(M_0(a_0, M_1(a_1, M_2(a_2, ...)$!!! 1
$=$
$M_0(M_1(a_1)) == M_0 M_1(a_1)$

As shown, the returned monadic action in the first instance has a duplication of M_0 - if this was a $State$ monadic action which modifies an existing state for example, this would change the behaviour of the monadic stream, potentially returning a different value for a_1 if the return value of the computation was dependant on the value of the state.

2.2 Right identity

$m >>= return == m$
This case results in the follwing call to bind:

$(joinPrelimMS \,.\, joinInnerMS)\ (fmap\ return\ m)$
$=$
$M_0 R_M(a_0, M_1 R_M R_M(a_1, M_2 R_M R_M R_M(a_2, ...$
$=$ [joining the monadic actions]
$M_0(a_0, M_1(a_1, M_2(a_2, ...$

This property seems to hold for any monad M - only "seems" since it *looks* like a valid monster is produced with the actions in the correct order, and experimenting with $State$ and [ ] monads seem to verify this assumption, but it hasn't been proven explcitly.

3 Required properties of the inner monad

This section looks specifically at the monster produced on the LHS of the left identity monad law, and compares it to the "clean" monster produced on the RHS. This is done by performing some operations and seeing if their behaviour is the same or different.
Looking at the left identity above, it seems like the monad used in a monster needs to satisfy the property that any "equivalent" monadic action performed twice will result in the same monadic action if they are joined.

In the case of taking the second element:

$$\begin{aligned}&\big(M_0(a_0, M_1^0(a_1, M_2^0(a_2, ...)\ !!!\ 1\\ &=\\ &M_0M_0M_1(a_1) = M_0M_1(a_1)\end{aligned}$$

In the case of taking the third element:

$$\begin{aligned}&M_0M_0M_1M_0M_1M_2(a_2)\\ &= M_0M_1M_0M_1M_2(a_2) && \text{[join the top two actions]}\\ &= (M_0M_1)(M_0M_1)M_2(a_2) && \text{[group equal "composite" actions]}\\ &= M_0M_1M_2(a_2) && \text{[join the composite actions]}\end{aligned}$$

It is valid to group actions, since the assumed property states that the same two actions performed sequentially results in the same overall action (using join).

3.1 Commutativity

What if an arbitrary monadic action was inserted at an arbitrary depth in the monster? Does this produce the same behaviour on both of the above monsters?
I'm going to use the function:

```
insertAct :: MonStr m a -> MonStr m a
insertAct = fmap (\(a,s) -> join (fmap (\(a',s') -> M_x (a',s')) s)
```

to insert a new action. It could be extended to introduce the action at any level by adding arbitrarily many $fmap$s, recursing down each tail s.

3.1.1 RHS "clean" monster

For "clean" monsters (ones not produced in the monadic stream join operation)

$$\begin{aligned}&insertAct\ \$\ M_0(a_0, M_1(a_1, M_2(a_2, ... && \text{[inserting a monadic action } M_x \text{]}\\ &= M_0(a_0, M_1M_x(a_1, M_2(a_2, ...\end{aligned}$$

Now take the nth element and you get the pile of monadic actions:

$$M_0M_1M_xM_2M_3M_4...(a_n)$$

If $n = 2$:

$$M_0M_1M_xM_2(a_2)$$

3.1.2 LHS monster produced by bind

Now to do the same for one produced by the monadic stream created in joining the left identity:

$$\begin{aligned} & insertAct\ \$\ M_0(a_0, M_1^0(a_1, M_2^0(a_2, ... && \text{[inserting a monadic action } M_x\text{]} \\ &= M_0(a_0, M_1^0 M_x(a_1, M_2^0(a_2, ... \end{aligned}$$

Now take the nth element, resulting in the sequence of actions (expanded for clarity):

$$M_0 M_0 M_1 M_x M_0 M_1 M_2 M_0 M_1 M_2 M_3 ... (a_n)$$

If $n = 2$:

$$\begin{aligned} & M_0 M_0 M_1 M_x M_0 M_1 M_2(a_2) \\ &= M_0 M_1 M_x M_0 M_1 M_2(a_2) \end{aligned}$$

Here we hit a problem - you can't combine the two groups of monadic actions $M_0 M_1$ to reduce them using join, since they aren't next to eachother. This removes that reduction as an option to show the equality of the two monsters.

3.1.3 Resolution

For this to work, the monadic actions also need to be **commutative**, which would allow the following reduction:

$$\begin{aligned} & M_0 M_1 M_x M_0 M_1 M_2(a_2) && \\ &= (M_0 M_1) M_x (M_0 M_1) M_2(a_2) && \text{[grouping actions]} \\ &= (M_0 M_1)(M_0 M_1) M_x M_2(a_2) && \text{[applying commutativity]} \\ &= M_0 M_1 M_x M_2(a_2) && \end{aligned}$$

This results in the same monster as the "clean" RHS.
So far, it seems that in order for a monadic stream of a monad M to itself be a monad, M needs to have at least these properties:

- Two monadic actions which perform the same action, should join to produce the same action - **indempotency**
- The order in which monadic actions are applied should be have no effect on the result - **commutativity**