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)

1. fmap f (return a_0)

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

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

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

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 $M(a)$

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).