

What is a monad?

Julian Porter

julian.porter@lincoln.oxon.org

1 Introduction

Monads and monad transformers have a wholly undeserved reputation for being somewhat fearful. Unfortunately, even the best books on Haskell tend to further this reputation by choosing to introduce monads by giving a couple of examples, then stating the monad laws and running away very fast, and the treatment of monad transformers, even in the otherwise excellent [9], is unclear at best.

As I said, this reputation is undeserved. In this paper I will attempt to give a general motivation for monads as machines which carry around information not all of which is visible at any time, and for which there is a defined recipe to, given two old machines, make a new one. I then move on to do the same for the closely related topics of monoids, additive monads (*MonadPlus* in Haskell) and finally monad transformers.

My approach for each of these is to start off with a metaphor, then to use the metaphor to infer the laws that control the behaviour of this type of object (instead of simply stating them) and then give some examples. I deliberately keep these simple, making particular use of the almost canonical example of the list monad. This does break down a little in the final section, where I give a general construction for making monad transformers out of monads, but I think that the computations involved are worth seeing, as I am not aware that they are written down in any easily accessible place, and they are interesting, if a little complex.

One notion I am keen to push is that of *naturally*: if I have two expressions that have the same type signature for all values of the variables in the system, then, in the absence of other information they must be the same. I use this to motivate the monad laws by showing that they are the only reasonable option, but in the calculations of the final section I show the power of this approach by deducing the general form of $\gg=$ for a whole class of monad transformers purely from consideration of type-signatures.¹

2 Monads

2.1 Motivation

Say I have been supplied with a collection of boxes. Each of them contains an amount of hidden machinery, about which I know nothing, and has on its exterior a display, which presents me with a value of some kind. So I can look at a box and read off its value, but I have no idea where that value came from. Now say I have a machine that, given a value, will produce a box with a new value. So, formally, a box is a type ma where a is the type of the value on its display.

So I have a way of making boxes from values, but I still don't know what's going on inside. Now it seems reasonable that if I have a box and the machine, I should be able to connect the box to the machine so the machine reads the box's value and uses it to produce a new box. So I can construct new boxes from old, thanks to my machine. Formally, this means that I have an operation

$$(\gg=) :: ma \rightarrow (a \rightarrow mb) \rightarrow mb$$

¹I want an operator of signature $mta \rightarrow (a \rightarrow mtb) \rightarrow mtb$. Given operators $a \rightarrow ma$, $ma \rightarrow (a \rightarrow mb) \rightarrow mb$, $a \rightarrow ta$ and $ta \rightarrow (a \rightarrow tb) \rightarrow tb$. It turns out that there is a way of building an operator of the desired signature from them, and I show that it does, in fact, give rise to the 'right' result in one easily-computed case.

where the argument of type $m a$ is the initial box, with value of type a , the argument of type $a \rightarrow m b$ is a machine that takes an a and uses it to produce a box with value of type b , and the returned $m b$ is the resulting new box.

We would also expect that there should be a very special kind of machine that, if I give it a value, presents me with what I will call a *trivial box* whose display shows precisely that value. Formally, this has signature:

$$\text{return} :: a \rightarrow m a$$

where the argument of type a is the value, and the returned $m a$ is the box that displays it.

These are, of course, the signatures of the monadic operations, so we see that we can motivate a monad simply as a class of thing that displays a value but has other hidden content, and that lets us combine things to make new things.

One might notice that the description looks very similar to functional composition, and that is precisely right. In fact the simplest monad of all, the **identity monad** is defined as follows:

```

data Id a = I { val :: a }
(>>=) :: Id a -> (a -> Id b) -> Id b
(>>=) x f = f (val x)
return :: a -> Id a
return = I

```

Clearly for any function $f :: a \rightarrow Id b$ we have $f \equiv \text{return} \circ \text{val} \circ f \equiv \text{return} \circ g$ where $g = \text{val} \circ f :: a \rightarrow b$. Now $I x \gg= \text{return} \circ g \equiv I (g x)$, so $\gg=$ is precisely functional composition. Therefore *a monad is a generalised function that can be composed*.

2.2 The monad laws

Let's start by thinking about trivial boxes. Say I program my machine so that when I feed in a value I get a trivial box whose display shows that value, and say I connect some box to the machine. Obviously the resulting box should display the same value, but what about its internal composition? This comes down to a critical point about *return*: the only information it has when constructing its result is the value on the display; no information is provided to specify the resulting trivial box's internal composition. Now the internal composition of a trivial box could depend in some way on its value, but the natural approach in such circumstances is to assume that if we have no information, then we do nothing. So a trivial box has no internal structure of its own, which means that when it is connected to another box, the machine just copies it. Formally, we have the **first monad law**:

$$x \gg= \text{return} \equiv x$$

From here to the second law is an easy step. Say I connect the boxes the other way round, so I connect a machine capable of producing any box to the output of a trivial box. As there is no information in the trivial box apart from its value, all that the new box can do is to take that value and push it into the machine to produce a new box from it. Formally:

$$\text{return } x \gg= f \equiv f x$$

which is the **second monad law**.

The third law is simultaneously simple to motivate and slightly tricky to express. Say I have two of the box-producing machines and one box. Connecting the first machine to the box produces a box,

which I can feed into the second machine to get a result box. But a each machine takes a value and produces a box, I ought to be able to connect them together to make a new machine, and then connect that to the original box, and common sense suggests that the result box should be the same as in the first case. Quasi-formally, this can be expressed as saying that

$$(x \ggg f) \ggg g \equiv x \ggg (f \ggg g)$$

The problem is on the right-hand side: I am not connecting a box to a machine, but a machine to a machine. Therefore, if I have a value, I feed it into the first machine, to produce a box, which I can then connect to the second machine. Formally, this means that I get the machine corresponding to the function

$$y \rightarrow (f y) \ggg g$$

which, for any value y , constructs the box $f y$ and then connects it to the machine g . Putting this together we get the **third monad law**:

$$(x \ggg f) \ggg g \equiv x \ggg (\lambda y \rightarrow f y \ggg g)$$

Therefore, we can state the formal definition :

Definition 2.2.1. A *monad* is a parameterised type m with two operations:

$$\begin{aligned} \text{return} &:: a \rightarrow m a \\ (\ggg) &:: m a \rightarrow (a \rightarrow m b) \rightarrow m b \end{aligned}$$

subject to the **monad laws** (see [1]):

1. $x \ggg \text{return} \equiv x$
2. $\text{return } x \ggg f \equiv f x$
3. $(x \ggg f) \ggg g \equiv x \ggg (\lambda y \rightarrow f y \ggg g)$

Note that for any monad m we can define a number of helper functions that combine *return* and \ggg in useful ways. Two that we will meet later on are:

$$\begin{aligned} \text{join} &:: (\text{Monad } m) \Rightarrow m (m a) \rightarrow m a \\ \text{join} \cdot x &= x \ggg \text{id} \\ \text{liftM} &:: (\text{Monad } m) \Rightarrow (a \rightarrow b) \rightarrow m a \rightarrow m b \\ \text{liftM} \cdot f &= \lambda x \rightarrow x \ggg \text{return} \circ f \end{aligned}$$

so *join* removes one layer of monadic wrapping, and *liftM* does what its name implies, and lifts any function into a monad. These and other functions are defined in the standard library module *Control.Monad* (see [1]).

2.3 Examples

We have already met the identity monad *Id*. A small generalisation is the **maybe monad**, which is just the type *Maybe* with the functions

$$\begin{aligned} \text{return } x &= \text{Just } x \\ \text{Nothing} \ggg f &= \text{Nothing} \\ \text{Just } x \ggg f &= f x \end{aligned}$$

This behaves just like the composition monad with the additional rule that *Nothing* passes straight through $\gg=$, so evaluating any function on *Nothing* gives *Nothing*.

The next most obvious example is the **list monad** $m = []$, so $m a = [a]$. The operations are just

```
return x = x : []
y >>= f = concat $ map f x
```

so *return* returns the list with one element, and $\gg=$ applies $f :: a \rightarrow [b]$ to get a list of lists of b and then concatenates the lists in the result to get a list of b .

Now a more complex example. Say s is some fixed type, then define the **mapping monad**:

```
data Hom s a = H {fn :: s -> a}
```

so things of type $Hom\ s\ a$ are just maps from s to a . We can make this a monad as follows:

```
return :: a -> Hom s a
return x = H (const x)
(>>=) :: Hom s a -> (a -> Hom s b) -> Hom s b
(>>=) x g = H (\y -> (fn o g) (fn x) y)
```

The definition of *return* is obvious: we just get the constant function. $\gg=$ is not so obvious. Say $g :: a \rightarrow Hom\ s\ b$, so if y is an a then $fn\ (g\ y)$ maps $s \rightarrow b$, so define

```
g' :: a -> s -> b
g' y z = (fn (g y)) z
```

Then if we define

```
f' :: s -> a
f' = fn x
```

we can form

```
h' :: s -> b
h' y = g' (f' y) y
```

which, when wrapped in the type constructor H , is just the definition of $\gg=$.

Related to this is the **state monad**. Say we have a system that has some state of type s , and we inhabit a state machine that when applied transforms the state and publishes some result; then the appropriate model is this:

```
State s a = data S {st :: s -> (s, a)}
```

so the output of the function is the new state and the published value. Then we have

```
return x = S (\y -> (y, x))
x >>= f = S (\y -> st (f (val y)) (sta y))
where
  val v = snd $ (st x) v
  sta v = fst $ (st x) v
```

As usual, *return* is simple. Let's look at $\gg=$. If $x :: State\ s\ a$ and $f :: a \rightarrow State\ s\ b$ then we want to construct a function $z :: s \rightarrow (s, b)$. Say y is an s . First we push y through $st\ x$ to get a tuple

$$(sta\ y, val\ y) = (st\ x)\ y$$

We apply $st \circ f$ to $val\ y$ to get a function $f' :: s \rightarrow (s, b)$. Then we use $sta\ y$ as the input to f' . The result is $z\ s$. In words we compose the transformation x with another transformation selected from f by the a part of x 's output.

There are clear similarities between the state monad and the mapping monad. It turns out (see [8]) that the state monad is the result of applying a monad transformer to the mapping monad, the monad transformer in question being a generalisation of MapReduce.

3 Monoids

3.1 Motivation

A monoid is a rather intimidating-sounding type, but in fact it's a very simple idea. The motivating idea I'll use is a bag with items in it. Now, there are any number of things I can put into a bag, but there's one very special bag which I can describe precisely: the *empty bag*: a bag with no items in it. Formally, if $m\ a$ is a bag of things of type a , this means that there is distinguished element

$$mzero :: m\ a$$

corresponding to the empty bag.

What else can I do with a bag? Well I can put an item in and I can take an item out, but these operations are not necessarily well-defined. If $m\ a$ is lists of things of type a then I can take the *head*, but if $m\ a$ is unordered sets of a then there is no such obvious choice. Similarly, I cannot always add a single item, for example m might be constrained to only allow an even number of items in the bag. So things seem hopeless. But say I have an allows to construct two bags; then I ought to be able to join them together to form a single bag with the joint contents of both. That is to say, if whatever constraints there are on creating a bag are satisfied by bags, then this constraints should be satisfied by the union of the two bags. Formally we have a function

$$madd :: m\ a \rightarrow m\ a \rightarrow m\ a$$

3.2 The monoid laws

Monoids too have laws, but these are extremely simple and boil down to the observations that combining a bag with an empty bag leaves the original bag unchanged, and if we combine three bags, it doesn't matter if we join the first to the second and third or the first and second to the third. Formalising this we can state:

Definition 3.2.1. A *monoid* is a parameterised type m with two operations:²

$$mzero :: m\ a$$

$$madd :: m\ a \rightarrow m\ a \rightarrow m\ a$$

subject to the **monoid laws** (see [4]):

1. $mzero\ 'madd'\ x \equiv x$
2. $x\ 'madd'\ mzero \equiv x$
3. $x\ 'madd'\ (y\ 'madd'\ z) \equiv (x\ 'madd'\ y)\ 'madd'\ z$

²Strictly, *Data.Monoid* calls its operations *mempty* and *mappend* but I prefer these names: first because they are more descriptive, second for consistency with the terminology for an additive monad.

3.3 Examples

Let's look at some examples. The obvious monoid, in fact almost the canonical example of a monoid, is the list type $[a]$ with

$$\begin{aligned} mzero &= [] \\ madd\ x\ y &= x ++ y \end{aligned}$$

Here, obviously, we have the constructor $:$ to add an item to the head of list, and *tail* to remove the tail. Consequently, stacks too are monoids. If $m\ a = Set\ a$ (see [5] for details of *Data.Set*) then

$$\begin{aligned} mzero &= empty \\ madd &= union \end{aligned}$$

Here *insert* adds an item to the set, but there is no well defined way of removing an item (the *delete* function is rather different, because it removes a specified item). Generalising this a little, database tables also form a monoid. More generally still, any type a in the class *Num a* is a monoid, with

$$\begin{aligned} mzero &= 0 \\ madd &= (+) \end{aligned}$$

In this case the 'add an item' and 'remove an item' functions cannot be defined in any generality.

4 Additive monads

4.1 Motivation

At its simplest, an additive monad (implemented as *Control.MonadPlus*) is a monad which is also a monoid, so it has all the properties of a monad and of a monoid, meaning it obeys the monad and monoid laws. However these laws treat the monadic and monoidal operations separately; we also need to consider how they interact.

4.2 The additive monad laws

So let's think about what happens when we do $\gg=$ with *mzero* on one side or the other. Combining the metaphors, we are looking at boxes which are bags with items in, so think of them as bags of sub-boxes. So, suppose we have a machine which, whatever the input value, produces an empty bag. Then no matter what box we connect to the machine, it can only produce an empty bag. Formally this is the **first additive monad law**:

$$x \gg= (\lambda y \rightarrow mzero) \equiv mzero$$

Now say we take any box-producing machine and connect it to an empty bag. Just as in the discussion of the second monad law, what this means is that the only information available to the machine is an empty bag: an absence of information. The only thing the machine can do is to produce some fixed box, and the only natural choice for a fixed box is the empty bag itself. So we have the **second additive monad law**:

$$mzero \gg= f \equiv mzero$$

Therefore we can conclude:

Definition 4.2.1. An *additive monad* is a parameterised type m which is a monad and a monoid, additionally subject to the *additive monad laws* (see [2]):

1. $x \gg\equiv (\lambda y \rightarrow mzero) \equiv mzero$
2. $mzero \gg\equiv f \equiv mzero$

4.3 Examples

The standard example of an additive monad is the list monad. We have seen that lists form a monad and a monoid, and it is simple to verify that they obey the additional additive monad laws:

$$\begin{aligned} x \gg\equiv (\lambda y \rightarrow []) \equiv concat \$ map (\lambda y \rightarrow []) x \equiv [] \\ [] \gg\equiv f = concat \$ f [] \equiv [] \end{aligned}$$

As another example, we showed in [7] that any persistent datastore is itself an additive monad if the data being stored is an additive monad.

5 Monad Transformers

5.1 Motivation

In spite of their apparent complexity, monad transformers are extremely simple: a monad transformer is just a thing that takes a monad and uses it to produce a new kind of monad: formally it is a parameterised type t such that if m is a monad then $t m$ is also a monad, and $t m a$ is the new monad showing a displayed value of type a . In terms of our metaphor, the transformer is a new kind of machine that given boxes and their monadic machines spits out new kinds of boxes and machines. So say t is a monad transformer and m is a monad. Then $t m$ is a monad, which means that we have monadic functions

$$\begin{aligned} \underline{return} :: a \rightarrow t m a \\ (\underline{\gg\equiv}) :: t m a \rightarrow (a \rightarrow t m b) \rightarrow t m b \end{aligned}$$

(I shall be using underlining to distinguish where the function happens, so \underline{return} and $\underline{\gg\equiv}$ are the monad functions in m , while \underline{return} and $\underline{\gg\equiv}$ are their counterparts in $t m$.)

Now let's have a think about these: why is \underline{return} of type $a \rightarrow t m a$ and not (say) $m a \rightarrow t m a$? The answer is that we should really write $t m a$ as $(t m) a$ to make it clear that $t m$ is a thing in its own right, which takes values in a . But this raises an interesting point. I said that a transformer is a thing that converts monads into other monads. So if I have a particular instance of a monad, say $myMonad :: m a$, then I ought to be able to wrap it and its value in the transformer to get an instance of the new monad $myTransformedMonad :: t m a$. This is precisely a function

$$\underline{lift} :: m a \rightarrow t m a$$

and so a transformer has two lifting operations: one which wraps a value in a transformed monad, and the other which wraps a monad in the transformer.

5.2 The monad transformer laws

Let's have a think about this new function \underline{lift} and how it interacts with the monadic functions \underline{return} and $\underline{\gg\equiv}$. We see that there are now two ways of getting from a to $t m a$, that is \underline{return} and $\underline{lift} \circ \underline{return}$. Now

these two functions exist for all possible monads, so by naturality the difference between them can only depend on t , not on m or a . So the only solution is to impose the **first monad transformer law**:

$$\underline{\text{return}} \equiv \text{lift} \circ \text{return}$$

Now let's do the same thing for functions $m a \rightarrow (a \rightarrow m b) \rightarrow t m b$. Again there are two approaches

$$\text{lifted1} :: m a \rightarrow (a \rightarrow m b) \rightarrow t m b$$

$$\text{lifted1 } x f = \text{lift } (x \gg\!\!= f)$$

$$\text{lifted2} :: m a \rightarrow (a \rightarrow m b) \rightarrow t m b$$

$$\text{lifted2 } x f = (\text{lift } x) \gg\!\!= (\text{lift } \circ f)$$

So in *lifted1* we apply $\gg\!\!=$ and then lift to t , whereas in *lifted2* we lift to t then apply $\gg\!\!=$. The naturality argument applies again, so we must have $\text{lifted1} \equiv \text{lifted2}$, and we get the **second monad transformer law**:

$$\text{lift } (x \gg\!\!= f) = (\text{lift } x) \gg\!\!= (\text{lift } \circ f)$$

Therefore we can state:

Definition 5.2.1. A *monad transformer* is a doubly parameterised type t , such that if m is a monad then there are functions:

$$\underline{\text{return}} :: a \rightarrow t m a$$

$$\text{lift} :: m a \rightarrow t m a$$

$$\gg\!\!= :: t m a \rightarrow (a \rightarrow t m b) \rightarrow t m b$$

such that $t m$ is also a monad with monadic functions $\underline{\text{return}}$ and $\gg\!\!=$, and subject in addition to the *monad transformer laws* (see [3]):

1. $\underline{\text{return}} \equiv \text{lift} \circ \text{return}$
2. $\text{lift } (x \gg\!\!= f) = (\text{lift } x) \gg\!\!= (\text{lift } \circ f)$

5.3 Examples

The simplest monad transformer is the identity transformer IdT which takes a monad to itself, so

$$IdT m a = m a$$

$$\text{lift } x = x$$

$$\underline{\text{return}} x = \text{return}_m x$$

$$x \gg\!\!= f = x \gg\!\!=_m f$$

This is rather trivial. A more interesting example is the *MaybeT* transformer, which takes a monad and wraps it around *Maybe*. It is defined by:

$$\mathbf{data} \text{ MaybeT } m a = M \{ ma :: m (\text{Maybe } a) \}$$

$$\text{lift } x = M \$ (\text{lift } M_m \text{ Just}) x$$

$$\underline{\text{return}} x = M \$ \text{return}_m (\text{Just } x)$$

$$x \gg\!\!= f = M \$ (ma \circ x) \gg\!\!=_m f'$$

where

$$\begin{aligned} f' \text{ Nothing} &= M \$ \text{return}_m \text{ Nothing} \\ f' \text{ Just } y &= M \$ (m a \circ f) y \end{aligned}$$

It behaves pretty much as we would expect: a *Nothing* value propagates as *Nothing* down any chain of applications of $\gg\equiv$, while on *Just x* values, $\gg\equiv$ behaves like $\gg\equiv$.

Now consider the more general case where t and m are monads and we define $t m a = m (t a)$. Take

$$\begin{aligned} \underline{\text{return}} &:: a \rightarrow m (t a) \\ \underline{\text{return}} &= \text{return}_m \circ \text{return}_t \\ \underline{\text{lift}} &:: m a \rightarrow m (t a) \\ \underline{\text{lift}} x &= x \gg\equiv_m \underline{\text{return}} \end{aligned}$$

Say we are given a function

$$\text{swap} :: t m c \rightarrow m t c$$

Note immediately that

$$\begin{aligned} \underline{\text{lift}} \circ \text{return}_m &\equiv \text{return}_m \gg\equiv_m \underline{\text{return}} \\ &\equiv \underline{\text{return}} \end{aligned}$$

by the second monad law, and so the first monad transformer law holds good.

Defining $\gg\equiv$ is more complex and requires extra conditions on m and t , specifically that there exists a function

$$\text{swap} :: t (m a) \rightarrow m (t a)$$

Given $x :: m (t a)$ and $f :: a \rightarrow m (t b)$ then

$$\begin{aligned} \underline{\text{lift}} M_t f &:: t a \rightarrow t (m (t b)) \\ \text{swap} \circ (\underline{\text{lift}} M_t f) &:: t a \rightarrow m (t (t b)) \end{aligned}$$

But we also have

$$\underline{\text{lift}} M_m \text{join}_t :: m (t (t b)) \rightarrow m (t b)$$

so if we define

$$\begin{aligned} \text{rhs} &:: (a \rightarrow m (t b)) \rightarrow t a \rightarrow m (t b) \\ \text{rhs } f x &= (\underline{\text{lift}} M_m \text{join}_t) \circ (\text{swap} \circ (\underline{\text{lift}} M_t f)) \end{aligned}$$

Then we have

$$x \gg\equiv f := x \gg\equiv_m (\text{rhs } f)$$

A not particularly enlightening computation proves that:

Proposition 5.3.1. *If m and t are monads and there is a function $\text{swap} :: t (m a) \rightarrow m (t a)$ then if we define $\text{Trans } t$ by*

$$\text{Trans } t m a := m (t a)$$

and define the operations $\underline{\text{return}}$, $\gg\equiv$ and $\underline{\text{lift}}$ on $\text{Trans } t m$ as above, then $\text{Trans } t$ is a monad transformer.

It is a good exercise to show that IdT is the monad transformer this recipe associates to the identity monad, with $swap = id$. More ambitiously, we can show that the two definitions for a *Maybe* transformer agree:

Proposition 5.3.2. *Trans Maybe \equiv MaybeT.*

Proof. Starting with return, we have

$$\begin{aligned} \underline{\text{return}} &\equiv \text{return}_m \circ \text{return}_t \\ &\equiv \text{return}_m \circ \text{Just} \end{aligned}$$

which is precisely how we defined it in *MaybeT*. Turning to *lift* we have

$$\begin{aligned} \text{lift } x &\equiv x \ggg_m \underline{\text{return}} \\ &\equiv x \ggg_m (\lambda y \rightarrow \text{return}_m \text{Just } y) \\ &\equiv \text{liftM}_m \text{Just} \end{aligned}$$

which is again, just as in *MaybeT*. Now consider \ggg . Define

$$\begin{aligned} \text{swap} &:: \text{Maybe } (m \ a) \rightarrow m \ (\text{Maybe } a) \\ \text{swap Nothing} &= \text{return}_m \text{Nothing} \\ \text{swap Just } x &= \text{lift } x \end{aligned}$$

So $\text{swap Just } x \equiv x \ggg_m (\lambda y \rightarrow \text{return}_m \text{Just } y)$. We can expand the definitions of join_t and liftM_t as follows:

$$\begin{aligned} \text{join}_t \text{Nothing} &\equiv \text{Nothing} \\ \text{join}_t \text{Just Nothing} &\equiv \text{Nothing} \\ \text{join}_t \text{Just Just } x &\equiv \text{Just } x \\ \text{liftM}_t f \$ \text{Nothing} &\equiv \text{Nothing} \\ \text{liftM}_t f \$ \text{Just } x &\equiv \text{Just } (f \ x) \end{aligned}$$

Given all this, we can expand the function *rhs* as follows:

$$\begin{aligned} 1 \text{ rhs } f \text{ Nothing} &\equiv (\text{liftM}_m \text{join}_t) \$ \text{return}_m \text{Nothing} \\ 2 &\equiv (\text{return}_m \text{Nothing}) \ggg_m \text{return}_m \circ \text{join}_t \\ 3 &\equiv \text{return}_m \text{Nothing} \\ 4 \text{ rhs } f \text{ (Just } x) &\equiv (\text{liftM}_m \text{join}_t) \$ \text{swap Just } f \ x \\ 5 &\equiv (\text{liftM}_m \text{join}_t) \$ \text{lift } (f \ x) \\ 6 &\equiv \text{lift } (f \ x) \ggg_m \text{return}_m \circ \text{join}_t \\ 7 &\equiv f \ x \ggg_m (\lambda y \rightarrow \text{return}_m \text{Just } y \ggg_m \text{return}_m \text{join}_t) \\ 8 &\equiv f \ x \ggg_m (\lambda y \rightarrow \text{return}_m \text{join}_t \text{Just } y) \\ 9 &\equiv f \ x \ggg_m (\lambda y \rightarrow \text{return}_m \ y) \\ 10 &\equiv f \ x \end{aligned}$$

(we used the first monad law in lines 3 and 8, the second monad law in line 10, and the third monad law implicitly in line 7). Now $x \ggg f \equiv x \ggg_m \text{rhs } f$, and therefore

$$\begin{aligned} x \ggg f &\equiv x \ggg_m f' \\ \text{where} & \\ f' \text{ Nothing} &= \text{return}_m \text{Nothing} \\ f' \text{ (Just } y) &= f \ y \end{aligned}$$

which is (up to some notation) the definition given above for *MaybeT*. □

There is also a monad transformer for lists

```
data ListT m a = L { li :: m [a] }
```

such that $ListT = Trans []$, but its definition is somewhat involved, as its version of *swap* is highly non-trivial. The interested reader is referred to [6]. Another instructive example of a reasonably complex monad transformer is the MapReduce transformer described in [8].

References

- [1] *Control.Monad documentation*. URL: <http://hackage.haskell.org/packages/archive/base/latest/doc/html/Control-Monad.html>.
- [2] *Control.MonadPlus documentation*. URL: <http://hackage.haskell.org/packages/archive/base/latest/doc/html/Control-Monad.html#t:MonadPlus>.
- [3] *Control.Monad.Trans.Class documentation*. URL: <http://hackage.haskell.org/packages/archive/transformers/latest/doc/html/Control-Monad-Trans-Class.html>.
- [4] *Data.Monoid documentation*. URL: <http://hackage.haskell.org/packages/archive/base/latest/doc/html/Data-Monoid.html>.
- [5] *Data.Set documentation*. URL: <http://hackage.haskell.org/packages/archive/containers/latest/doc/html/Data-Set.html>.
- [6] *ListT Done Right*. URL: http://haskell.org/haskellwiki/ListT_done_right.
- [7] Julian Porter. “Storage as a Monad Transformer”. In: (2011). URL: <http://jpembeddedsolutions.files.wordpress.com/2011/10/storagemonad.pdf>.
- [8] Julian Porter. “The MapReduce type of a Monad”. In: (2011). URL: <http://media.jpembeddedsolutions.com/pdf/mrmonad.pdf>.
- [9] John Goerzen; Bryan O’Sullivan & Donald Bruce Stewart. *Real World Haskell*. O’Reilly, 2008. ISBN: 0596514980.