Recursive Monadic Bindings: Technical Development and Details

Levent Erkok, John Launchbury

Oregon Graduate Institute
Department of Computer Science and Engineering
20000 NW Walker Rd
Beaverton, OR 97006-1999 USA

Technical Report No. CSE 00-011
June 20, 2000

Recursive Monadic Bindings: Technical Development and Details*

Levent Erkök John Launchbury
Oregon Graduate Institute of Science and Technology

June 20, 2000

Abstract

Monads have become a popular tool for dealing with computational effects in Haskell for two significant reasons: equational reasoning is retained even in the presence of effects; and program modularity is enhanced by hiding "plumbing" issues inside the monadic infrastructure. Unfortunately, not all the facilities provided by the underlying language are readily available for monadic computations. In particular, while recursive monadic computations can be defined directly using Haskell's built-in recursion capabilities, there is no natural way to express recursion over the *values* of monadic actions. Using examples, we illustrate why this is a problem, and we propose an extension to Haskell's do-notation to remedy the situation. It turns out that the structure of monadic value-recursion depends on the structure of the underlying monad. We propose an axiomatization of the recursion operation and provide a catalogue of definitions that satisfy our criteria. The proofs of the claims we make throughout the report, along with other technical development, is presented in the appendices.

Computing Review Subject Categories: Formal definitions and theory (D.3.1), Language constructs and features (D.3.3).

Keywords: Haskell, monads, recursion, mfix, fixed-point operators.

1 Introduction

We begin with a puzzle. Consider the following piece of almost-Haskell code:

```
isEven :: Int -> Maybe Int
isEven n = if even n then Just n else Nothing

puzzle :: [Int]
puzzle = do (x, z) <- [(y, 1), (y^2, 2), (y^3, 3)]
            Just y <- map isEven [z+1 .. 2*z]
            return (x + y)
```

*A version of this paper, without the appendices, is going to appear in the *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*, 2000.

```
newtype Out a = Out (a, String)

instance Monad Out where
  return x          = Out (x, "")
  Out ~(x, s) >>= f = let Out (y, s') = f x
                      in Out (y, s ++ s')

instance Show a => Show (Out a) where
  show (Out (v, s))  = "Value: " ++ show v
                       ++ "\nTrace:" ++ s

comp :: Int -> (Int, Int) -> Out (Int, Int)
comp i (a, b) = Out ((max a b, min a b), msg)
      where c1  = ": swap: " ++ show (a, b)
            c2  = ": pass: " ++ show (a, b)
            msg = "\nUnit " ++ show i ++
                  (if a < b then c1 else c2)

type QuadInts = (Int, Int, Int, Int)
sort4 :: QuadInts -> Out QuadInts
sort4 (a, b, c, d) =
        do (e, f) <- comp 1 (a, b)  -- unit 1
           (g, h) <- comp 2 (c, d)  -- unit 2
           (n, i) <- comp 3 (e, g)  -- unit 3
           (j, k) <- comp 4 (f, h)  -- unit 4
           (m, l) <- comp 5 (i, j)  -- unit 5
           return (k, l, m, n)
```

Figure 2: Haskell code implementing network of Figure 1

```
Main> sort4 (23, 12, -1, 2)
Value: (-1,2,12,23)
Trace:
Unit 1: pass: (23,12)
Unit 2: swap: (-1,2)
Unit 3: pass: (23,2)
Unit 4: pass: (12,-1)
Unit 5: swap: (2,12)
```

A quick look at the trace reveals that it is consistent with the operation of the network for this input.

In the definition of **sort4**, we carefully selected the execution order of the units such that all values were available before they were used. What if it was inconvenient to arrange for this? In our example, for instance, what if we want to observe the action of unit 3 after unit 5 in the sorting network problem? Notice that unit 5 uses the value i, which is produced by unit 3. Ideally, we would like to be able to change the function **sort4** to:

```
sort4 (a, b, c, d) =
        do (e, f) <- comp 1 (a, b)  -- unit 1
           (g, h) <- comp 2 (c, d)  -- unit 2
           (j, k) <- comp 4 (f, h)  -- unit 4
           (m, l) <- comp 5 (i, j)  -- unit 5
           (n, i) <- comp 3 (e, g)  -- unit 3
           return (k, l, m, n)
```

3 Recursive bindings for the do-notation

Currently, a do-expression in Haskell behaves like the `let*` of Scheme: the bound variables are available only in the textually following expressions. We need the do-notation to behave more like the `let` of Haskell, which allow recursive bindings. Of course, it is not necessarily the case that all monads will allow for such recursive bindings. We call a monad *recursive*, if there is a "sensible" way to allow for this kind of recursion. We codify what "sensible" should mean in Section 4. In this section, we look at a syntactic extension to Haskell that allows recursive bindings in the do-notation. This extension is a variant of the do-notation, called the μdo-notation. Just like the do-notation is available for any monad, the μdo-notation will be automatically available for any recursive-monad.

3.1 μdo: The details

Recall that a do-expression is translated into a series of applications of $\gg=$ [9]. Similarly, we need μdo to translate into more primitive components. We use a fixed-point operator, called mfix, whose type is $\forall a.\ (a \to m\ a) \to m\ a$, where m is the underlying monad. The translation is:

$$
\begin{array}{l}
\mu\texttt{do } p_1 \leftarrow e_1 \\
\quad \ldots \\
\quad p_n \leftarrow e_n \\
\quad e
\end{array}
\quad \Longrightarrow \quad
\begin{array}{l}
\texttt{mfix } (\lambda\tilde{}BV.\ \texttt{do } p_1 \leftarrow e_1 \\
\qquad \ldots \\
\qquad p_n \leftarrow e_n \\
\qquad v \leftarrow e \\
\qquad \texttt{return } BV) \\
\gg= \ \lambda\ BV.\ \texttt{return } v
\end{array}
$$

where BV stands for the k-tuple consisting of all the variables occurring in all the binding patterns plus the brand new variable v. Notice that each one of $p_1 \ldots p_n$, the binding patterns, can be any valid Haskell pattern, not just simple variables. The variables that are bound by these patterns may appear anywhere in $e_1 \ldots e_n$ and e. A variable may not be multiply bound: neither in the same pattern, nor in different patterns.

As an example, consider the following μdo expression, which implements a sorting network for three numbers:

```
mdo (d, e) <- comp 1 (a, b)
    (i, h) <- comp 3 (d, f)
    (f, g) <- comp 2 (e, c)
    return (g, h, i)
```

After the translation, it becomes:

```
mfix (\~(d, e, i, h, f, g, v) ->
     do (d, e) <- comp 1 (a, b)
        (i, h) <- comp 3 (d, f)
        (f, g) <- comp 2 (e, c)
        v      <- return (g, h, i)
        return (d, e, i, h, f, g, v))
   >>= \(d, e, i, h, f, g, v) -> return v
```

The translation fails to type check for obvious reasons: The function f is no longer polymorphic.

The solution we adopt is to require let bindings to be monomorphic in a μdo. That is, `let` becomes just a syntactic sugar within μdo, translated as:[2]

$$\begin{array}{lcl} \texttt{let } p_1 \texttt{ = } e_1 & & p_1 \leftarrow \texttt{return } e_1 \\ \quad \ldots & \Longrightarrow & \ldots \\ \quad p_n \texttt{ = } e_n & & p_n \leftarrow \texttt{return } e_n \end{array}$$

This gives us a uniform design. If a polymorphic value definition is required, one should use the standard `let` expressions of Haskell, rather than the `let` generator, which will create its own scope with poymorphic names. The translation and the related issues are detailed in [4].

3.3 Implementation

We have a straightforward implementation available obtained by modifying the source code for the Hugs system.[3] This implementation acts as a preprocessor, i.e. it performs the translation at the source level, and hence the amount of changes required in the Hugs source code is fairly small. We expect the same to hold when the translation is done inside the compiler. The required changes will be localized to type checking and desugaring routines.

The related class declaration for recursive monads is:

```
class Monad m => MonadRec m where
  mfix :: (a -> m a) -> m a
```

In this simple implementation, occurrences of `let` expressions are translated blindly, without requiring them to be monomorphic.

4 Recursive monads

The previous section addressed syntax. Now we turn to the meat of the issue and study mfix directly. We start by looking for a generic mfix.

4.1 The generic mfix

The fixed point operator, fix, which has type $\forall a.\ (a \to a) \to a$, has a generic definition that works for all cases.[4] For a lazy language like Haskell, the definition is just:

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

[2] This extends to functions as well, basically `let f x y = z` will become `f <- return (\x y -> z)`.

[3] More information and downloading instructions are available online at URL: `http://www.cse.ogi.edu/PacSoft/projects/muHugs`.

[4] Technically, the underlying type needs to be a pointed CPO, but this requirement is vacuously satisfied in Haskell as all types are pointed, i.e. non-termination can happen at any type.

Figure 5: Interpreting axiom 1

this axiom. The dashed box represents where the mfix computation takes place. In this figure, the loop on the right hand side represents fix, while the one on the left corresponds to mfix. The thin line represents the value being processed through the computation. The thick line in the lower part of the diagram represents the computational effect (side effects, other changes in the monadic data, etc.) The fixed-point is computed only over the value part.

Axiom 2 shows how to pull a term that doesn't contribute to the fixed-point computation from the left-hand-side of a $\gg\!=$, provided x does not appear free in a:

$$\text{mfix}\ (\lambda x.a \gg\!= f\ x) = a \gg\!= \lambda y.\text{mfix}\ (\lambda x.f\ x\ y)$$

Notice that the value of a is constant throughout the computation. Hence, we should be able to compute it only once (if need be) and put it into the fixed-point loop. Figure 6 is a pictorial representation of this axiom. Notice that both hand sides of the diagram are essentially the same.

Figure 6: Interpreting axiom 2

Axiom 3, depicted in Figure 7, states a useful fact about fixed-point computations involving more than one variable:

$$\text{mfix}\ (\lambda\tilde{}(x,\_).\text{mfix}\ (\lambda\tilde{}(\_,y).f\ (x,y))) = \text{mfix}\ f$$

The function f has type: $\forall a, b.\ (a,b) \to m\ (a,b)$. On the right hand side, we compute the fixed point simultaneously over both variables. On the left hand side, we perform a two step computation, where the fixed-point is computed using only one variable at a time.

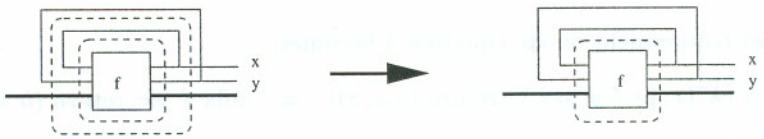


Figure 7: Interpreting axiom 3

This axiom corresponds to Bekić's theorem for the usual fixed-point computations [21]. Notice that, again, both hand sides of Figure 7 are essentially the same. It can be shown that the symmetric law:

$$\text{mfix}\ (\lambda\tilde{}(\_,y).\text{mfix}\ (\lambda\tilde{}(x,\_).f\ (x,y))) = \text{mfix}\ f$$

[5] This is the so-called *extension* of a function from values to computations to a function from computations to computations, see [16].

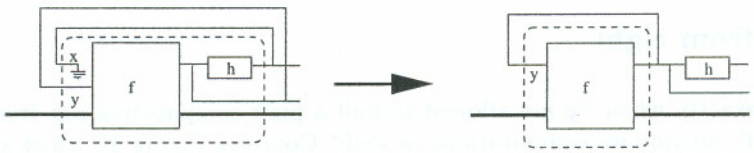


Figure 8: Interpreting equation 4

Corollary 4.6 Provided h is strict, the following equation holds for any recursive-monad:

$$\text{mfix}\ (\lambda x.f\ x \gg\!= \text{return} \cdot h) \quad = \quad \text{mfix}\ (\lambda x.\text{return}\ (h\ x) \gg\!= f) \gg\!= \text{return} \cdot h \tag{6}$$

where $f :: a \to m\ b$ and $h : b \to a$. Equivalently:

$$\text{mfix}\ (\text{map}\ h \cdot f) = \text{map}\ h\ (\text{mfix}\ (f \cdot h))$$

Figure 9 depicts the situation. The purity requirement on h is essential: we cannot reorder any effects, as order does matter in performing them. The strictness requirement on h is quite important as well.

Figure 9: Interpreting equation 6

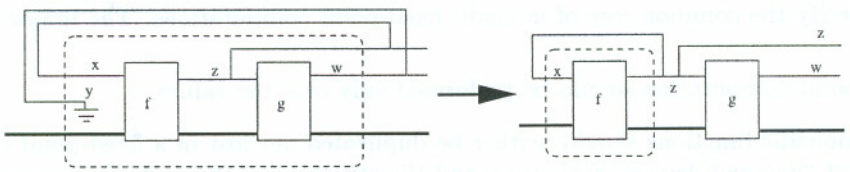


Figure 10: Interpreting equation 7

Intuitively, the fixed-point computation on the lhs will start of by feeding $\perp$ to f, while the computation on the rhs will start of by feeding $h\ \perp$. Unless $h\ \perp = \perp$, this will provide more information to f on the rhs. Hence, we might get a $\perp$ on the lhs, while a non-$\perp$ value on the right. (We will see an example in Section 6.2.) However, there are monads for which the equality holds even when h is non-strict. The state monad is such an example (Section 6.4).

The inspiration for Corollary 4.6 comes from a a very well known law for the ordinary fixed-point computations. We have:

$$\text{fix}\ (f \cdot g) = f\ (\text{fix}\ (g \cdot f))$$

One can see the correspondence more clearly by using Kleisli composition, defined as: $f \Diamond g = \lambda x.f\ x \gg\!= g$, where x does not occur free in f or g. Now, equation 6 becomes ($\Diamond$ binds less tightly than $\cdot$):

$$\text{mfix}\ (f \Diamond \text{return} \cdot h) = \text{mfix}\ (\text{return} \cdot h \Diamond f) \gg\!= \text{return} \cdot h$$

Definition 5.1 *Monad homomorphisms and embeddings.* Let $(m, \text{return}_m, \gg\!=_m)$ and $(n, \text{return}_n, \gg\!=_n)$ be two monads. A monad homomorphism, $\epsilon : m \to n$, is a family of functions (one for each type a, $\epsilon_a : m\ a \to n\ a$) such that:

$$\epsilon \cdot \text{return}_m = \text{return}_n \tag{8}$$

$$\epsilon_b\ (p \gg\!=_m h) = \epsilon_a\ p \gg\!=_n \epsilon_b \cdot h \tag{9}$$

where $p : m\ a$ and $h : a \to m\ b$. An embedding is a monic (i.e. injective) monad-homomorphism.

We extend the definition to cover the recursive case:

Definition 5.2 *Recursive-monad homomorphisms and embeddings.* Let m and n be two recursive-monads and let $\epsilon : m \to n$ be a monad homomorphism. We call ϵ a recursive-monad homomorphism if it also satisfies:

$$\epsilon\ (\text{mfix}_m h) = \text{mfix}_n\ (\epsilon \cdot h) \tag{10}$$

Similarly, a recursive-monad embedding is a monic recursive-monad homomorphism.

We will see concrete examples of recursive-monad embeddings in the next section.

Theorem 5.3 Let $\epsilon : m \to n$ be an embedding of a monad m into a recursive-monad n. To conclude that m is recursive, it's sufficient to show that there exists a function mfix_m such that ϵ is a recursive-monad embedding.

The proof is by simple equational reasoning. We also note that equations 4, 6 and 7 are preserved through monad-embeddings as well. Furthermore, composition of two embeddings is still an embedding.

This theorem not only provides a method for obtaining proofs for mfix axioms automatically for certain monads, but it also provides additional assurance that the axioms represent characteristic properties of monadic fixed-points.

6 A catalogue of recursive-monads

In this section we examine a number of monads that are frequently used in programming.

6.1 Identity

The identity monad is the monad of pure values. The Haskell declaration is:

```
newtype Id a = Id { unId :: a }

instance Monad Id where
  return x   = Id x
  Id x >>= f = f x

instance MonadRec Id where
  mfix f = fix (f . unId)
```

Notice that we use a `newtype` declaration rather than a `data`. This choice is not arbitrary. Since all Haskell data types are lifted (i.e. $\perp$ and `Id` $\perp$ are different), we would introduce an unwanted element if we had used `data`. It is a simple matter to check that mfix axioms are satisfied. One particular way of doing so is by embedding the `Id` monad into another recursive-monad, for instance the `State` monad (Section 6.4). In addition, equation 5 is satisfied, equation 6 holds even if h is non-strict, and equation 7 holds as an equality.

For this example, lhs of equation 7 yields $\bot$, while the rhs yields `Nothing`. Looking closely, we see that the right hand side first produces the fixed point of f, which is the infinite list $[1\ldots]$. Then, outside the mfix loop, g ignores this value and returns `Nothing`. Within the mfix loop, the fixed-point is constructed as the limit of the chain: $\{\bot,\ 1:\bot,\ 1:1:\bot,\ \ldots\}$. When we look at the left hand side, we see a different situation. The function g acts on each value in this chain, and it yields $\bot$ for the second element. (Matching $1:\bot$ against $[x]$ leads to nontermination.) Now, the fixed point is computed over and over starting from $\bot$, yielding $\bot$ as the result. In general, the `Maybe` monad will satisfy property 7 as an inequality. If we look more closely, we see that the problem lies within the fact that $\gg=$ for the `Maybe` monad is strict in its first argument, resulting in the failure. Unfortunately, there is no way to alleviate this problem. We conclude that this equation can not be satisfied as long as the $\gg=$ of the monad is strict in its first argument. This requirement practically rules out any datatype that has more than one constructor from satisfying property 7 as an equality.

6.3 List

Apart from `List`'s normal use as a convenient data structure, it is also used as a monad for capturing backtracking computations. The `MonadRec` declaration is:

```
instance MonadRec [] where
  mfix f = case fix (f . head) of
             []    -> []
             (x:_) -> x : mfix (tail . f)
```

The intuition behind this definition of mfix is the following: For a function of type $a \to [a]$, the fixed point is of type $[a]$, i.e. it's a list. Each element of this fixed-point should be the fixed point of the function restricted to that particular position. That is, the ith entry of the fixed point of a function with type $a \to [a]$, say f, should be the fixed point of the function: $\text{head} \cdot \text{tail}^i \cdot f$. In other words,

$$\text{mfix}\ (\lambda x.[h_1\ x, \ldots, h_n\ x]) = [\text{fix}\ h_1, \ldots, \text{fix}\ h_n]$$

or, more generally:

$$\text{mfix}\ f = \text{fix}\ (\text{head} \cdot f) : \text{mfix}\ (\text{tail} \cdot f)$$

This definition would work well if the fixed-point were an infinite list. However, it fails to capture the finite case. Notice that we are computing the fixed points of the functions of the form $\text{head} \cdot f$. If f ever returns `[]`, we want to stop the computation, rather than taking the head (which will yield $\bot$). Hence, recalling that

$$\text{fix}\ (\text{head} \cdot f) = \text{head}\ (\text{fix}\ (f \cdot \text{head}))$$

we can compute the fixed points of the functions of the form $f \cdot \text{head}$ (whose results will be a lists), and stop when we get an empty list. Putting these ideas together, we arrive at the definition we have given above.

Analogous to Lemma 6.1, we have:

Lemma 6.2 The `List` instance of mfix satisfies:

$$\begin{aligned}
\text{mfix}\ f = \bot &\longleftrightarrow f\ \bot = \bot \\
\text{mfix}\ f = [\,] &\longleftrightarrow f\ \bot = [\,] \\
\text{mfix}\ f = [\bot] &\longleftrightarrow f\ \bot = [\bot] \\
\text{head}\ (\text{mfix}\ f) &= \text{fix}\ (\text{head} \cdot f) \\
\text{tail}\ (\text{mfix}\ f) &= \text{mfix}\ (\text{tail} \cdot f) \\
\text{mfix}\ (\lambda x.f\ x : g\ x) &= \text{fix}\ f : \text{mfix}\ g \\
\text{mfix}\ (\lambda x.f\ x +\!\!+ g\ x) &= \text{mfix}\ f +\!\!+ \text{mfix}\ g
\end{aligned}$$

Without tags, the definition of mfix is simply:

$$\text{mfix } f = \lambda s.\text{let } (a, s') = f\ a\ s \text{ in } (a, s')$$

The `State` monad satisfies all mfix axioms, hence it is recursive. The definition of mfix clearly shows that the fixed-point computation is performed only on values, not on the other parts of the monad. Furthermore, equation 5 holds, equation 6 does not require a strict h and equation 7 is satisfied as an equality.

6.5 State with exceptions

Often the computations that have side effects fail to yield a value. This concept is generally modeled with a combination of the state and exception monads. In this section we look at two examples.

The first version considers the case when neither a value nor an updated state is available after a computation. The declarations are (again, we drop explicit tags):

```
newtype STE s a = s -> Maybe (a, s)

instance Monad (STE s) where
  return x = \s -> Just (x, s)
  f >>= g  = \s -> case f s of
                     Nothing      -> Nothing
                     Just (a, s') -> g a s'

instance MonadRec (STE s) where
  mfix f = \s -> let a = f b s
                     b = fst (unJust a)
                 in a
```

Now we consider when the computation might fail but an updated state is still available. The declarations are:

```
newtype STE2 s a = s -> (Maybe a, s)

instance Monad (STE2 s) where
  return x = \s -> (Just x, s)
  f >>= g  = \s -> case f s of
                     (Nothing, s') -> (Nothing, s')
                     (Just a, s')  -> g a s'

instance MonadRec (STE2 s) where
  mfix f = \s -> let a = f b s
                     b = unJust (fst a)
                 in a
```

In both cases, the computation of the fixed-point is similar to those of `State` and `Maybe` monads. We equate the value part of the result with the input to the function. Notice the symmetry between the definitions and the `newtype` declarations.

It turns out both of these monads are recursive. However, they require strict h for satisfying equation 6 and they don't satisfy equation 7 as an equality. This is hardly surprising since the `Maybe` monad behaves like this as well. As with all other cases, both monads satisfy equation 5.

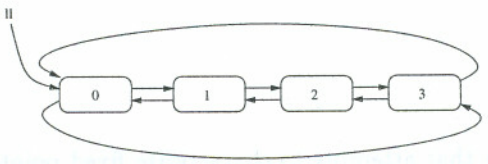
By this declaration, the μdo-notation becomes available for the `IO` monad. Each node in our list will have a mutable boolean value indicating whether it has been visited, left and right nodes and a single integer value for the data:

```
> data N = N (IORef Bool, N, Int, N)
```

To create a new node with value `i` in between the nodes `b` and `f`, we use the function `newNode`:

```
> newNode :: N -> Int -> N -> IO N
> newNode b i f = do v  <- newIORef False
>                    return (N (v, b, i, f))
```

Notice that the visited flag is set to `False`. We will use this function to create the following structure:



Here's the code for it:

```
> ll = mdo n0 <- newNode n3 0 n1
>          n1 <- newNode n0 1 n2
>          n2 <- newNode n1 2 n3
>          n3 <- newNode n2 3 n0
>          return n0
```

The use of μdo is essential: the cyclic nature of the construction is not expressible using an ordinary do-expression. We can test our implementation with a traversal function:

```
> data Dir = F | B deriving Eq
>
> traverse :: Dir -> N -> IO [Int]
> traverse d (N (v, b, i, f)) =
>     do visited <- readIORef v
>        if visited
>           then return []
>           else do writeIORef v True
>                   let next = if d == F then f else b
>                   is <- traverse d next
>                   return (i:is)
```

Here's a sample run:

```
Main> ll >>= traverse F >>= print
[0,1,2,3]
Main> ll >>= traverse B >>= print
[0,3,2,1]
```

9 Conclusions

Monads play an important role in functional programming by providing a clean methodology for expressing computational effects. Monadic computations use a certain sublanguage shaped by the functions that act on monadic objects. Haskell makes this approach quite convenient by providing the do-notation. A shortcoming, however, is that recursion over the results of monadic actions can not be conveniently expressed. Furthermore, it is not clear how to perform recursion on values in the presence of effects. In order to alleviate this problem, we have axiomatized monadic fix and implemented an extension to the do-notation, which can be used in expressing such recursive computations in a natural way. We expect that many applications can benefit from this work, as monads become more pervasive in functional programming.

Even though we have proposed a separate μdo construct, we believe that the usual do-expression of Haskell should be extended to capture this new style of programming. That is, there should not be a separate μdo keyword, but rather the compiler should analyze do-expressions to see if recursive bindings are employed, performing the translations as appropriate. An ambitious compiler may also perform simplifications based on the mfix axioms.

10 Acknowledgements

We are thankful to Ross Paterson, Amr Sabry, and to John Matthews and other members of the OGI PacSoft Research Group for valuable discussions.

The research reported in this paper is supported by Air Force Materiel Command (F19628-96-C-0161) and the National Science Foundation (CCR-9970980).

References

[1] Bjesse, P., Claessen, K., Sheeran, M., and Singh, S. Lava: Hardware design in Haskell. In *International Conference on Functional Programming* (Baltimore, July 1998).

[2] Cormen, T., Leiserson, C., and Rivest, R. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1989.

[3] Crole, R. L., and Pitts, A. M. New foundations for fixpoint computations. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science* (June 1990), pp. 489–497.

[4] Erkök, L., and Launchbury, J. A recursive `do` for Haskell: Design and Implementation. Available at `http://www.cse.ogi.edu/PacSoft/projects/muHugs/`.

[5] Friedman, D., and Sabry, A. Recursion in monads, or when is recursion a computational effect? Unpublished. Available at `http://www.cs.uoregon.edu/~sabry/papers/`.

[6] Hasegawa, M. Recursion from cyclic sharing: traced monoidal categories and models of cyclic lambda calculi. *Lecture Notes in Computer Science 1210* (1997).

[7] Hughes, J. Generalising monads to arrows. *Science of Computer Programming 37*, 1-3 (May 2000), 67–111.

[8] Jones, M. P., and Duponcheel, L. Composing monads. Tech. Rep. YALEU/DCS/RR-1004, Department of Computer Science, Yale University, Dec. 1993.

Appendices

In this appendix, we provide the proofs of the claims we have made, along with other technical details.

A Corollary 4.2

Corollary Provided x does not appear free in a, $\text{mfix}\ (\lambda x.a) = a$.

Proof

$$
\begin{array}{lll}
 & \text{mfix}\ (\lambda x.a) & \\
= & \text{mfix}\ (\lambda x.a \gg\!= \lambda y.\text{return}\ y) & \{f \gg\!= \text{return} = f\} \\
= & a \gg\!= \lambda y.\text{mfix}\ (\lambda x.\text{return}\ y) & \{\text{axiom 2}\} \\
= & a \gg\!= \lambda y.\text{mfix}\ (\lambda x.(\text{return} \cdot \text{const}\ y)\ x) & \{\text{const}\ y\ x = y\} \\
= & a \gg\!= \lambda y.\text{mfix}\ (\text{return} \cdot \text{const}\ y) & \{\text{eta-conversion}\} \\
= & a \gg\!= \lambda y.\text{return}\ (\text{fix}\ (\text{const}\ y)) & \{\text{axiom 1}\} \\
= & a \gg\!= \lambda y.\text{return}\ y & \{\text{fix} \cdot \text{const} = \text{id}\} \\
= & a & \{f \gg\!= \text{return} = f\}\ \square
\end{array}
$$

B Corollary 4.3

Corollary $f\ \bot \sqsubseteq \text{mfix}\ f$

Proof Notice that

$$(\lambda x.f\ \bot) \sqsubseteq f$$

For any argument x, lhs yields $f\ \bot$ while the rhs yields $f\ x$, satisfying the inequality trivially by the monotonicity of f. Since mfix is monotonic, we have:

$$\text{mfix}\ (\lambda x.f\ \bot) \sqsubseteq \text{mfix}\ f$$

By the previous corollary lhs is exactly $f\ \bot$, concluding the proof. $\square$

C Theorem 4.4

Theorem $\forall s : A \to B$, $f : A \to m\ A$, $g : B \to m\ B$, if $g \cdot s = \text{map}\ s \cdot f$ then $\text{map}\ s\ (\text{mfix}_A\ f) = \text{mfix}_B\ g$, provided s is strict.

Proof Recall the type of mfix: $\forall X.(X \to mX) \to mX$, where m is a recursive-monad. We derive the free theorem as follows: By parametricity: $(\text{mfix}, \text{mfix}) \in \forall \mathcal{X}.(\mathcal{X} \to m\mathcal{X}) \to m\mathcal{X}$. This implies that, for all relations $s : A \Leftrightarrow B$, $(\text{mfix}_A, \text{mfix}_B) \in (s \to m\ s) \to m\ s$. As usual, we will restrict to a function instance, i.e. we'll consider the case where s is a function of type $A \to B$. Now, for all $(f, g) \in s \to m\ s$, we have $(\text{mfix}_A\ f, \text{mfix}_B\ g) \in m\ s$. Notice that $f : A \to m\ A$ and $g : B \to m\ B$. The condition $(f, g) \in s \to m\ s$ implies that for all $(x, y) \in s$ we should have $(f\ x, g\ y) \in m\ s$. Since s is a function, this is the same as saying: $y = s\ x$ implies $g\ y = \text{map}\ s\ (f\ x)$, or equivalently: $g \cdot s = \text{map}\ s \cdot f$. Now we look at the result: $(\text{mfix}_A\ f, \text{mfix}_B\ g) \in m\ s$, which is equivalent to: $\text{map}\ s\ (\text{mfix}_A\ f) = \text{mfix}_B\ g$. The strictness requirement on s arises from the statetement of the parametricity theorem; Since every type in Haskell contains $\bot$, no general remarks can be made for non-strict s. $\square$

we have: map ss (mfix f) = mfix (map ss $\cdot$ f $\cdot$ ss). (We'll refer to this equation as the mfix-swap rule below.) Consider:

$$
\begin{array}{lll}
 & \text{mfix } (\lambda\tilde{}(\_, y).\text{mfix } (\lambda\tilde{}(x, \_).f\ (x, y))) & \\
= & \text{mfix } (\lambda t.\text{mfix } (\lambda v.f\ (\pi_1\ v, \pi_2\ t))) & \{\text{rewrite}\} \\
= & \text{map } (\text{ss} \cdot \text{ss}) \ (\text{mfix } (\lambda t.\text{mfix } (\lambda v.f\ (\pi_1\ v, \pi_2\ t)))) & \{\text{ss} \cdot \text{ss} = \text{id}, \text{map id} = \text{id}\} \\
= & \text{map ss } (\text{map ss } (\text{mfix } (\lambda t.\text{mfix } (\lambda v.f\ (\pi_1\ v, \pi_2\ t))))) & \{\text{map is a functor}\} \\
= & \text{map ss } (\text{mfix } (\text{map ss} \cdot (\lambda t.\text{mfix } (\lambda v.f\ (\pi_1\ v, \pi_2\ (\text{ss } t)))))) & \{\text{mfix-swap and rewrite}\} \\
= & \text{map ss } (\text{mfix } (\text{map ss} \cdot (\lambda t.\text{mfix } (\lambda v.f\ (\pi_1\ v, \pi_1\ t))))) & \{\pi_2 \cdot \text{ss} = \pi_1\} \\
= & \text{map ss } (\text{mfix } (\lambda t.(\text{map ss } (\text{mfix } (\lambda v.f\ (\pi_1\ v, \pi_1\ t)))))) & \{\text{rewrite}\} \\
= & \text{map ss } (\text{mfix } (\lambda t.\text{mfix } (\text{map ss} \cdot (\lambda v.f\ (\pi_1\ v, \pi_1\ t)) \cdot \text{ss}))) & \{\text{mfix-swap}\} \\
= & \text{map ss } (\text{mfix } (\lambda t.\text{mfix } (\text{map ss} \cdot (\lambda v.f\ (\pi_1\ (\text{ss } v), \pi_1\ t))))) & \{\text{rewrite}\} \\
= & \text{map ss } (\text{mfix } (\lambda t.\text{mfix } (\text{map ss} \cdot (\lambda v.f\ (\pi_2\ v, \pi_1\ t))))) & \{\pi_1 \cdot \text{ss} = \pi_2\} \\
= & \text{map ss } (\text{mfix } (\lambda t.\text{mfix } (\lambda v.\text{map ss } (f\ (\pi_2\ v, \pi_1\ t))))) & \{\text{rewrite}\} \\
= & \text{map ss } (\text{mfix } (\lambda t.\text{mfix } (\lambda v.(\text{map ss} \cdot f)\ (\pi_2\ v, \pi_1\ t)))) & \{\text{rewrite}\} \\
= & \text{map ss } (\text{mfix } (\lambda t.\text{mfix } (\lambda v.(\text{map ss} \cdot f \cdot \text{ss})\ (\pi_1\ t, \pi_2\ v)))) & \{\text{rewrite}\} \\
= & \text{map ss } (\text{mfix } (\text{map ss} \cdot f \cdot \text{ss})) & \{\text{axiom 3}\} \\
= & \text{mfix } (\text{map ss} \cdot \text{map ss} \cdot f \cdot \text{ss} \cdot \text{ss}) & \{\text{mfix-swap}\} \\
= & \text{mfix } (\text{map } (\text{ss} \cdot \text{ss}) \cdot f) & \{\text{map is a functor, ss} \cdot \text{ss} = \text{id}\} \\
= & \text{mfix } f & \{\text{map id} = \text{id}\}
\end{array}
$$

G Theorem 5.3

Theorem Let $\epsilon : m \to n$ be an embedding of a monad m into a recursive-monad n. To conclude that m is recursive, it's sufficient to show that there exists a function mfix_m such that ϵ is a recursive-monad embedding.

Proof The proof proceeds by considering each axiom in turn. We first look at axiom 1. Consider the expression $\epsilon\ (\text{mfix}_m\ (\eta_m \cdot h))$:

$$
\begin{array}{lll}
 & \epsilon\ (\text{mfix}_m\ (\eta_m \cdot h)) & \\
= & \text{mfix}_n\ (\epsilon \cdot \eta_m \cdot h) & \{\text{eqn 10}\} \\
= & \text{mfix}_n\ (\eta_n \cdot h) & \{\text{eqn 8}\} \\
= & \eta_n\ (\text{fix } h) & \{\text{axiom 1 for } n\} \\
= & (\epsilon \cdot \eta_m)\ (\text{fix } h) & \{\text{eqn 8}\} \\
= & \epsilon\ (\eta_m\ (\text{fix } h)) & \{\text{definition of } \cdot\}
\end{array}
$$

The result follows by the assumption that ϵ is an embedding, i.e. it's monic. Notice that we only relied on the first axiom for the recursive-monad n.

Similarly, for axiom 2, we consider: $\epsilon\ (\text{mfix}_m\ (\lambda x.a \gg\!=_m f\ x))$ where x does not appear free in a:

$$
\begin{array}{lll}
 & \epsilon\ (\text{mfix}_m\ (\lambda x.a \gg\!=_m f\ x)) & \\
= & \text{mfix}_n\ (\epsilon \cdot (\lambda x.a \gg\!=_m f\ x)) & \{\text{eqn 10}\} \\
= & \text{mfix}_n\ (\lambda x.\epsilon\ (a \gg\!=_m f\ x)) & \{\text{rewrite}\} \\
= & \text{mfix}_n\ (\lambda x.\epsilon\ a \gg\!=_n \epsilon \cdot f\ x) & \{\text{eqn 9}\} \\
= & \epsilon\ a \gg\!=_n \lambda y.\text{mfix}_n\ (\lambda x.(\epsilon \cdot f\ x)\ y) & \{\text{axiom 2 for } n\} \\
= & \epsilon\ a \gg\!=_n \lambda y.\text{mfix}_n\ (\lambda x.\epsilon\ (f\ x\ y)) & \{\text{rewrite}\} \\
= & \epsilon\ a \gg\!=_n \lambda y.\text{mfix}_n\ (\epsilon \cdot \lambda x.f\ x\ y) & \{\text{rewrite}\} \\
= & \epsilon\ a \gg\!=_n \lambda y.\epsilon\ (\text{mfix}_m\ (\lambda x.f\ x\ y)) & \{\text{eqn 10}\} \\
= & \epsilon\ a \gg\!=_n \epsilon \cdot \lambda y.\text{mfix}_m\ (\lambda x.f\ x\ y) & \{\text{rewrite}\} \\
= & \epsilon\ (a \gg\!=_m \lambda y.\text{mfix}_m\ (\lambda x.f\ x\ y)) & \{\text{eqn 9}\}
\end{array}
$$

$$
\begin{array}{lll}
 & \epsilon\ (\mathrm{mfix}_m\ (\lambda x.f\ x \gg\!=_m \mathrm{return}_m \cdot h)) & \\
= & \mathrm{mfix}_n\ (\epsilon \cdot (\lambda x.f\ x \gg\!=_m \mathrm{return}_m \cdot h)) & \{\text{eqn 10}\} \\
= & \mathrm{mfix}_n\ (\lambda x.\epsilon\ (f\ x) \gg\!=_n \epsilon \cdot \mathrm{return}_m \cdot h) & \{\text{eqn 9}\} \\
= & \mathrm{mfix}_n\ (\lambda x.(\epsilon \cdot f)\ x \gg\!=_n \mathrm{return}_n \cdot h) & \{\text{eqn 8}\} \\
= & \mathrm{mfix}_n\ (\lambda x.\mathrm{return}_n\ (h\ x) \gg\!=_n \epsilon \cdot f) \gg\!=_n \mathrm{return}_n \cdot h & \{\text{corollary 4.6 for } n\} \\
= & \mathrm{mfix}_n\ (\lambda x.(\epsilon \cdot \mathrm{return}_m)\ (h\ x) \gg\!=_n \epsilon \cdot f) \gg\!=_n \epsilon \cdot \mathrm{return}_m \cdot h & \{\text{eqn 8}\} \\
= & \mathrm{mfix}_n\ (\lambda x.\epsilon\ (\mathrm{return}_m\ (h\ x)) \gg\!=_n \epsilon \cdot f) \gg\!=_n \epsilon \cdot \mathrm{return}_m \cdot h & \{\text{rewrite}\} \\
= & \mathrm{mfix}_n\ (\lambda x.\epsilon\ (\mathrm{return}_m\ (h\ x) \gg\!=_m f)) \gg\!=_n \epsilon \cdot \mathrm{return}_m \cdot h & \{\text{eqn 9}\} \\
= & \mathrm{mfix}_n\ (\epsilon \cdot (\lambda x.\mathrm{return}_m\ (h\ x) \gg\!=_m f)) \gg\!=_n \epsilon \cdot \mathrm{return}_m \cdot h & \{\text{rewrite}\} \\
= & \epsilon\ (\mathrm{mfix}_m\ (\lambda x.\mathrm{return}_m\ (h\ x) \gg\!=_m f)) \gg\!=_n \epsilon \cdot \mathrm{return}_m \cdot h & \{\text{eqn 10}\} \\
= & \epsilon\ (\mathrm{mfix}_m\ (\lambda x.\mathrm{return}_m\ (h\ x) \gg\!=_m f) \gg\!=_m \mathrm{return}_m \cdot h) & \{\text{eqn 9}\}
\end{array}
$$

which implies the required result since ϵ is monic. Notice that we have never mentioned whether h was strict or not, proof carries on for both cases. □

J Equation 7 revisited

Recall (in-)equality 7:

$$\mathrm{mfix}\ (\lambda\tilde{}(x,y).f\ x \gg\!= \lambda z.g\ z \gg\!= \lambda w.\mathrm{return}\ (z,w)) \sqsubseteq \mathrm{mfix}\ f \gg\!= \lambda z.g\ z \gg\!= \lambda w.\mathrm{return}\ (z,w)$$

We would like to prove that, if a recursive-monad n satisfies this property as an equality (inequality) then any recursive-monad m that embeds into n will satisfy it as an equality (inequality). We will see that this, in general, requires the embedding to be split, i.e. there should be a left-inverse for ϵ:

Proof We start with the expression:

$$\epsilon\ (\mathrm{mfix}_m\ (\lambda\tilde{}(x,y).f\ x \gg\!=_m \lambda z.g\ z \gg\!=_m \lambda w.\mathrm{return}_m\ (z,w)))$$

and proceed as in the previous embedding proofs:

$$
\begin{array}{lll}
 & \epsilon\ (\mathrm{mfix}_m\ (\lambda\tilde{}(x,y).f\ x \gg\!=_m \lambda z.g\ z \gg\!=_m \lambda w.\mathrm{return}_m\ (z,w))) & \\
= & \mathrm{mfix}_n\ (\epsilon \cdot (\lambda\tilde{}(x,y).f\ x \gg\!=_m \lambda z.g\ z \gg\!=_m \lambda w.\mathrm{return}_m\ (z,w))) & \{\text{eqn 10}\} \\
= & \mathrm{mfix}_n\ (\lambda\tilde{}(x,y).\epsilon\ (f\ x \gg\!=_m \lambda z.g\ z \gg\!=_m \lambda w.\mathrm{return}_m\ (z,w))) & \{\text{rewrite}\} \\
= & \mathrm{mfix}_n\ (\lambda\tilde{}(x,y).\epsilon\ (f\ x) \gg\!=_n \epsilon \cdot (\lambda z.g\ z \gg\!=_m \lambda w.\mathrm{return}_m\ (z,w))) & \{\text{eqn 9}\} \\
= & \mathrm{mfix}_n\ (\lambda\tilde{}(x,y).(\epsilon \cdot f)\ x \gg\!=_n (\lambda z.\epsilon\ (g\ z \gg\!=_m \lambda w.\mathrm{return}_m\ (z,w)))) & \{\text{rewrite}\} \\
= & \mathrm{mfix}_n\ (\lambda\tilde{}(x,y).(\epsilon \cdot f)\ x \gg\!=_n (\lambda z.(\epsilon\ (g\ z) \gg\!=_n \epsilon \cdot (\lambda w.\mathrm{return}_m\ (z,w))))) & \{\text{eqn 9}\} \\
= & \mathrm{mfix}_n\ (\lambda\tilde{}(x,y).(\epsilon \cdot f)\ x \gg\!=_n (\lambda z.(\epsilon \cdot g)\ z \gg\!=_n \lambda w.(\epsilon \cdot \mathrm{return}_m)\ (z,w))) & \{\text{rewrite}\} \\
= & \mathrm{mfix}_n\ (\lambda\tilde{}(x,y).(\epsilon \cdot f)\ x \gg\!=_n (\lambda z.(\epsilon \cdot g)\ z \gg\!=_n \lambda w.\mathrm{return}_n\ (z,w))) & \{\text{eqn 8}\}
\end{array}
$$

In the next step, we apply the corresponding equation for the recursive-monad n. We use the symbol $\approx$ to mean either one $=$ or $\sqsubseteq$. If monad n satisfies the property as a strict equality then it means $=$, otherwise it means $\sqsubseteq$:

$$
\begin{array}{lll}
\approx & \mathrm{mfix}_n\ (\epsilon \cdot f) \gg\!=_n \lambda z.(\epsilon \cdot g)\ z \gg\!=_n \lambda w.\mathrm{return}_n\ (z,w) & \{\text{eqn 7 for } n\} \\
= & \mathrm{mfix}_n\ (\epsilon \cdot f) \gg\!=_n \lambda z.(\epsilon \cdot g)\ z \gg\!=_n \lambda w.(\epsilon \cdot \mathrm{return}_m)\ (z,w) & \{\text{rewrite}\} \\
= & \mathrm{mfix}_n\ (\epsilon \cdot f) \gg\!=_n \lambda z.(\epsilon \cdot g)\ z \gg\!=_n \lambda w.\epsilon\ (\mathrm{return}_m\ (z,w)) & \{\text{rewrite}\} \\
= & \mathrm{mfix}_n\ (\epsilon \cdot f) \gg\!=_n \lambda z.\epsilon\ (g\ z) \gg\!=_n \epsilon \cdot \lambda w.\mathrm{return}_m\ (z,w) & \{\text{rewrite}\} \\
= & \mathrm{mfix}_n\ (\epsilon \cdot f) \gg\!=_n \lambda z.\epsilon\ (g\ z \gg\!=_m \lambda w.\mathrm{return}_m\ (z,w)) & \{\text{eqn 9}\} \\
= & \mathrm{mfix}_n\ (\epsilon \cdot f) \gg\!=_n \epsilon \cdot (\lambda z.g\ z \gg\!=_m \lambda w.\mathrm{return}_m\ (z,w)) & \{\text{rewrite}\} \\
= & \epsilon\ (\mathrm{mfix}_m\ f) \gg\!=_n \epsilon \cdot (\lambda z.g\ z \gg\!=_m \lambda w.\mathrm{return}_m\ (z,w)) & \{\text{eqn 10}\} \\
= & \epsilon\ (\mathrm{mfix}_m\ f \gg\!=_m \lambda z.g\ z \gg\!=_m \lambda w.\mathrm{return}_m\ (z,w)) & \{\text{eqn 9}\}
\end{array}
$$

Now, since f is not strict (otherwise mfix f would be $\bot$ by above equality), $f\ \bot$ is either $\mathtt{N}$ or $\mathtt{J}\ x$ for some x. The case $\mathtt{J}\ x$ would have resulted in the limit of the chain to be a `Just` term (by monotonicity) which is not the case by the assumption. Hence, $f\ \bot = \mathtt{N}$. To prove the second implication, assume $f\ \bot = \mathtt{N}$. Now, mfix f = fix $(f \cdot \text{unJust}) = \bigsqcup_k (f \cdot \text{unJust})^k \bot = \bigsqcup\{\bot, \mathtt{N}, \mathtt{N}, \ldots\} = \mathtt{N}$.

Third equivalence: Similarly, we prove: mfix $f = \mathtt{J}\ \bot \rightarrow f\ \bot = \mathtt{J}\ \bot$, and $f\ \bot = \mathtt{J}\ \bot \rightarrow$ mfix $f = \mathtt{J}\ \bot$. For the first implication, we assume mfix $f = \mathtt{J}\ \bot$ and reason exactly as above to conclude that mfix $f = \bigsqcup\{\bot, f\ \bot, \ldots\} = \mathtt{J}\ \bot$ and hence, $f\ \bot = \mathtt{J}\ \bot$ by monotonicity. For the second implication, we assume: $f\ \bot = \mathtt{J}\ \bot$. Now, mfix f = fix $(f \cdot \text{unJust}) = \bigsqcup_k (f \cdot \text{unJust})^k \bot = \bigsqcup\{\bot, \mathtt{J}\ \bot, \mathtt{J}\ \bot, \ldots\} = \mathtt{J}\ \bot$.

Fourth equality: Recall that fix $(f \cdot g) = f$ (fix $(g \cdot f)$). We have: unJust (mfix f) = unJust (fix $(f \cdot \text{unJust})$) = fix (unJust $\cdot f$). □

L.2 Proving that the maybe monad is recursive

The proof that the maybe monad is recursive is done by embedding it into the `List` monad. The embedding is:

$$\epsilon\ x = \begin{cases} \bot & x = \bot, \\ [\,] & x = \text{Nothing}, \\ [\text{y}] & x = \text{Just}\ y \end{cases}$$

Here are the proofs for the embedding equations:

Monic requirement: ϵ is in fact a split-monic, with the obvious left inverse:

$$\epsilon^l\ x = \begin{cases} \bot & x = \bot, \\ \text{Nothing} & x = [\,], \\ \text{Just}\ y & x = y : ys \end{cases}$$

Before proving the equations, recall that: $\text{return}_l = \lambda x.[x]$ and $\text{return}_m = \mathtt{Just}$. (We use the subscript m for the maybe monad and l for the list.)

Equation 8: We need $\epsilon \cdot \mathtt{Just} = \lambda x.[x]$. By applying both hand sides to an arbitrary p, we get $[p]$, proving the equivalence.

Equation 9: We need $\epsilon\ (p \gg\!=_m h) = \epsilon\ p \gg\!=_l \epsilon \cdot h$. Case analysis on p:

- $p = \bot$: Both hand sides reduce to $\bot$.
- $p = \mathtt{N}$: Both hand sides reduce to $[\,]$.
- $p = \mathtt{J}\ x$: Both hand sides reduce to $\epsilon\ (h\ x)$.

Equation 10: We need $\epsilon\ (\text{mfix}_m\ h) = \text{mfix}_l\ (\epsilon \cdot h)$. Case analysis on $\text{mfix}_m\ h$:

- $\text{mfix}_m\ h = \bot$: By lemma 6.1, h is strict. Since ϵ is strict, so is $\epsilon \cdot h$. By lemma 6.2 (which is yet to be proven), $\text{mfix}_l\ (\epsilon \cdot h) = \bot$.

Similarly, consider the rhs:

$$\begin{array}{lll} & \text{mfix } f \gg\!= \lambda z.g\ z \gg\!= \lambda w.\text{return } (z,w) & \\ = & \text{fix } (f \cdot \text{unJust}) \gg\!= \lambda z.g\ z \gg\!= \lambda w.\text{return } (z,w) & \{\text{defn of mfix}\} \\ = & (\bigsqcup_k (f \cdot \text{unJust})^k\ \bot) \gg\!= \lambda z.g\ z \gg\!= \lambda w.\text{return } (z,w) & \{\text{defn of fix}\} \\ = & \bigsqcup_k ((f \cdot \text{unJust})^k\ \bot \gg\!= \lambda z.g\ z \gg\!= \lambda w.\text{return } (z,w)) & \{\text{continuity}\} \end{array}$$

Now, we will do a case analysis on the value of $f\ \bot$:

- $f\ \bot = \bot$: Both hand sides become: $\bigsqcup \{\bot, \bot, \ldots\} = \bot$.
- $f\ \bot = \mathtt{N}$: Both hand sides become: $\bigsqcup \{\bot, \mathtt{N}, \mathtt{N} \ldots\} = \mathtt{N}$.
- $f\ \bot = \mathtt{J}\ \bot$. In this case, mfix $f = \mathtt{J}\ \bot$. The rhs simply becomes: $g\ \bot \gg\!= \lambda w.\text{return } (\bot, w)$. The lhs chain looks like:

  $$\{\bot, g\ \bot \gg\!= \lambda w.\text{return } (\bot, w), \ldots\}$$

  We perform a case analysis on $g\ \bot$:
  - $g\ \bot = \bot$. Both lhs and rhs reduce to $\bot$.
  - $g\ \bot = \mathtt{N}$. Both lhs and rhs reduce to $\mathtt{N}$.
  - $g\ \bot = \mathtt{J}\ a$. Both lhs and rhs reduce to $\mathtt{J}\ (\bot, a)$.
- $f\ \bot = \mathtt{J}\ v_0,\ v_0 \neq \bot$. We analyse this final case in detail below.

Notice that, we haven't mentioned any side conditions yet, i.e. the proof so far applies for all f and g. The final case, however, requires the side conditions. Before going into the details, we make some observations:

1. For $k \geq 0$, unJust $((f \cdot \text{unJust})^k\ \bot) = (\text{unJust} \cdot f)^k\ \bot$. The proof is by simple induction on k and is skipped.
2. Consider the chain $(f \cdot \text{unJust})^k\ \bot,\ k \geq 0$. We have:

   $$f^k\ \bot = \{\bot, \mathtt{J}\ v_0, \mathtt{J}\ v_1, \mathtt{J}\ v_2, \ldots\}$$

   by the monotonicity of f. Here, $v_k = \text{unJust } ((f \cdot \text{unJust})^k \bot)$, $k > 0$. Furthermore, $\{v_0, v_1, \ldots\}$ is a chain too.
3. By the first observation, we have: $v_k = (\text{unJust} \cdot f)^k\ \bot,\ k > 0$.

Now, we perform a case analysis on the value of $g\ v_0$.

- $g\ v_0 = \bot$: In this case, lhs $= \bigsqcup \{\bot, \bot, \ldots\} = \bot$. And rhs is:

  $$\bigsqcup \{\bot, \bot, g\ v_0 \gg\!= \lambda w.\text{return } (v_0, w), g\ v_1 \gg\!= \lambda w.\text{return } (v_1, w), \ldots\}$$

  We claim $g\ v_i = \bot$ for $i \geq 0$. Notice that $f\ \bot = \mathtt{J}\ v_0$ and $v_0 \neq \bot$ and $g\ v_0 = \bot$ (case assumptions). Then $(g \cdot \text{unJust} \cdot f)\ \bot = \bot$, i.e. $(g \cdot \text{unJust} \cdot f)$ is strict. Now, all the conditions in the theorem hold, hence we can use the fact that $g \cdot (\text{unJust} \cdot f)^k$ is strict for $k > 0$. That is, $g\ ((\text{unJust} \cdot f)^k \bot) = \bot = g\ v_k$, $k > 0$, as required. (The case $i = 0$ is covered by the case assumption.) Hence the rhs becomes $\bot$ as well.

  Notice that this is the only place in the proof that we resort to the side conditions. If the side conditions are not satisfied we can only state that lhs (which is $\bot$) will be $\sqsubseteq$ rhs.

Recall the counter-example, stating the need for inequality for the maybe monad. We repeat the example here for convenience:

```
f :: [Int] -> Maybe [Int]
f xs = Just (1:xs)

g :: [Int] -> Maybe Int
g [x] = Nothing
g _   = Nothing
```

We see that:

- $f : [\text{Int}] \to [\text{Int}]$ (i.e. [Int] is non-flat).
- $f \perp = \texttt{Just}\ (1 : \perp)$ (i.e. $(1 : \perp) \neq \perp$).
- $(g \cdot \text{unJust} \cdot f)\ \perp = \perp$, i.e. The function $g \cdot \text{unJust} \cdot f$ is strict.

But,

- The function $g \cdot (\text{unJust} \cdot f)^2$ is *not* strict. Notice that $(g \cdot (\text{unJust} \cdot f)^2)\ \perp = \texttt{Nothing}$.

This clearly violates the requirement of the theorem.

M The list monad

M.1 Lemma 6.2

Lemma The `List` instance of mfix satisfies:

$$\begin{aligned}
\text{mfix}\ f = \perp &\longleftrightarrow f\ \perp = \perp \\
\text{mfix}\ f = [\,] &\longleftrightarrow f\ \perp = [\,] \\
\text{mfix}\ f = [\perp] &\longleftrightarrow f\ \perp = [\perp] \\
\text{head}\ (\text{mfix}\ f) &= \text{fix}\ (\text{head} \cdot f) \\
\text{tail}\ (\text{mfix}\ f) &= \text{mfix}\ (\text{tail} \cdot f) \\
\text{mfix}\ (\lambda x.f\ x : g\ x) &= \text{fix}\ f : \text{mfix}\ g \\
\text{mfix}\ (\lambda x.f\ x +\!\!+ g\ x) &= \text{mfix}\ f +\!\!+ \text{mfix}\ g
\end{aligned}$$

Proof We look at each case in turn:

First equivalence:

$$\begin{aligned}
\text{mfix}\ f = \perp &\longleftrightarrow \text{fix}\ (f \cdot \text{head}) = \perp \\
&\longleftrightarrow \bigsqcup \{\perp, (f \cdot \text{head})\ \perp, (f \cdot \text{head})^2 \perp, \ldots\} = \perp \\
&\longleftrightarrow (f \cdot \text{head})\ \perp = \perp \\
&\longleftrightarrow f\ \perp = \perp
\end{aligned}$$

We have:

$$\text{fix}\ (\text{tail} \cdot f \cdot \text{head}) = \bigsqcup\{\bot, \bot, \ldots\} = \bot$$

hence, the rhs is $\bot$. But so is lhs, trivially.

Case 2: mfix $f = x : xs$:

```
tail (mfix f) = tail (case fix (f . head) of
                        []    -> []
                        (x:_) -> x : mfix (tail . f))
```

Since mfix $f = x : xs$, the `case` should be taking its second branch, finally yielding mfix (tail $\cdot f$), as required.

Sixth equality:

```
mfix (\x. (f x : g x)) = case fix ((\x. (f x : g x)) . head) of ...
                       = case (\x. (f x : g x)) (fix (\x. f x)) of...
                       = case f (fix f) : g (fix f) of
                           []    -> []
                           (q:_) -> q : mfix (tail . \x.(f x : g x))
                       = fix f : mfix (\x. g x)
                       = fix f : mfix g
```

Seventh equality: To prove: mfix $(\lambda x.f\ x +\!\!+ g\ x) =$ mfix $f +\!\!+$ mfix g, we do a case analysis on $f\ \bot$:

Case 1: $f\ \bot = \bot$: Since $(\lambda x.f\ x +\!\!+ g\ x)\ \bot = \bot$, both hand sides reduce to $\bot$.

Case 2: $f\ \bot = [\,]$: This implies that $f =$ const $[\,]$ (by the monotonicity of f). The rhs becomes mfix g. Similarly, lhs becomes: mfix $(\lambda x.\text{const}\ [\,]\ x +\!\!+ g\ x) =$ mfix $(\lambda x.[\,] +\!\!+ g\ x) =$ mfix g.

Case 2: $f\ \bot = x : xs$: First, two observations: mfix f is also a cons-cell, and, $f\ a$ is a cons-cell for any a. Consider the lhs:

$$\begin{aligned}
\text{mfix}\ (\lambda x.f\ x +\!\!+ g\ x) &= \text{mfix}\ (\lambda x.(\text{head}\ (f\ x) : \text{tail}\ (f\ x)) +\!\!+ g\ x) \\
&= \text{mfix}\ (\lambda x.(\text{head} \cdot f)\ x : (\text{tail}\ (f\ x) +\!\!+ g\ x)) \\
&= \text{fix}\ (\text{head} \cdot f) : \text{mfix}\ (\lambda x.(\text{tail} \cdot f)\ x +\!\!+ g\ x) \\
&= \text{head}\ (\text{mfix}\ f) : \text{mfix}\ (\lambda x.(\text{tail} \cdot f)\ x +\!\!+ g\ x)
\end{aligned}$$

The rhs becomes:

$$\begin{aligned}
\text{mfix}\ f +\!\!+ \text{mfix}\ g &= \text{head}\ (\text{mfix}\ f) : (\text{tail}\ (\text{mfix}\ f) + +\text{mfix}\ g) \\
&= \text{head}\ (\text{mfix}\ f) : (\text{mfix}\ (\text{tail} \cdot f) +\!\!+ \text{mfix}\ g)
\end{aligned}$$

After these preliminary steps, we use the approx lemma, i.e. we prove:

$$\forall n.\ \text{approx}\ n\ (\text{mfix}\ (\lambda x.f\ x +\!\!+ g\ x)) = \text{approx}\ n\ (\text{mfix}\ f +\!\!+ \text{mfix}\ g)$$

Recall the definition of approx:

Axiom 3 We do a case analysis on the value of mfix f:

Cases 1 and 2: mfix $f = \bot/[\,]$. Then $f\ \bot = \bot/[\,]$. We have:

$$\begin{array}{ll} & \text{mfix}\ (\lambda u.\text{mfix}\ (\lambda v.f\ (\pi_1\ u, \pi_2\ v))) = \bot/[\,] \\ \longleftrightarrow & \text{mfix}\ (\lambda v.f\ (\bot, \pi_2\ v)) = \bot/[\,] \\ \longleftrightarrow & f\ (\bot, \bot) = \bot/[\,] \\ \longleftrightarrow & \text{mfix}\ f = \bot/[\,] \end{array}$$

Cases 3: Now, we know that both hand sides are "cons-cells". We use the approx lemma to prove:

$$\forall n.\ \text{approx}\ n\ (\text{mfix}\ (\lambda u.\text{mfix}\ (\lambda v.f\ (\pi_1\ u, \pi_2\ v)))) = \text{approx}\ n\ (\text{mfix}\ f)$$

We perform an induction on n. The base case, $n = 0$, is trivial, as both sides reduce to $\bot$. For the inductive step, we assume:

$$\text{approx}\ k\ (\text{mfix}\ (\lambda u.\text{mfix}\ (\lambda v.f\ (\pi_1\ u, \pi_2\ v)))) = \text{approx}\ k\ (\text{mfix}\ f)$$

Notice that this holds $\forall f, g$. Here is the inductive step:

$$\begin{array}{ll} & \text{approx}\ (k+1)\ (\text{mfix}\ (\lambda u.\text{mfix}\ (\lambda v.f\ (\pi_1\ u, \pi_2\ v)))) \\ = & \text{head}\ (\text{mfix}\ (\lambda u.\text{mfix}\ (\lambda v.f\ (\pi_1\ u, \pi_2\ v)))) \\ & \quad : \text{approx}\ k\ (\text{tail}\ (\text{mfix}\ (\lambda u.\text{mfix}\ (\lambda v.f\ (\pi_1\ u, \pi_2\ v))))) \\ = & \text{fix}\ (\lambda u.\text{fix}\ (\lambda v.(\text{head} \cdot f)(\pi_1\ u, \pi_2\ v))) \\ & \quad : \text{approx}\ k\ (\text{mfix}\ (\lambda u.\text{mfix}\ (\lambda v.(\text{tail} \cdot f)(\pi_1\ u, \pi_2\ v)))) \\ = & \text{fix}\ (\text{head} \cdot f) : \text{approx}\ k\ (\text{mfix}\ (\text{tail} \cdot f)) \\ = & \text{approx}\ (k+1)\ (\text{head}\ (\text{mfix}\ f) : \text{tail}\ (\text{mfix}\ f)) \\ = & \text{approx}\ (k+1)\ (\text{mfix}\ f) \end{array}$$

M.3 Equation 5

Equation 5 holds as an equality for the list monad. Here's the proof:

Proof For notational convenience, define:

$$\langle f,\ g\rangle\ x = (f\ x,\ g\ x)$$

Notice that we don't impose a strictness requirement. Now, the lhs of equation 5 can be written as:

$$\begin{array}{lll} & \text{mfix}\ (\lambda\tilde{}(x, y).f\ y \gg\!= \text{return}\ (h\ z,\ z)) & \\ = & \text{mfix}\ (\lambda p.f\ (\pi_2\ p) \gg\!= \text{return} \cdot \langle h,\ \text{id}\rangle) & \{\text{rewrite}\} \\ = & \text{mfix}\ (\text{map}\ \langle h,\ \text{id}\rangle \cdot f \cdot \pi_2) & \{a \gg\!= \text{return} \cdot f = \text{map}\ f\ a\} \end{array}$$

Similarly, rhs becomes: map $\langle h,\ \text{id}\rangle$ (mfix f). To prove that

$$\text{mfix}\ (\text{map}\ \langle h,\ \text{id}\rangle \cdot f \cdot \pi_2) = \text{map}\ \langle h,\ \text{id}\rangle\ (\text{mfix}\ f)$$

Hence we have proved that equation 5 will hold for the list monad and any monad that embeds into it.

M.4 Equations 6 and 7

The list monad satisfies equation 6 for only strict h. The example given for the maybe monad applies here as well. Equation 7 does not hold as an equality. This is not surprising at all, since the `Maybe` monad does not satisfy it as an equality either. Although we have not constructed an explicit proof that property 7 will hold as an inequality, we have strong evidence that it does. Hence, we conjecture that it will apply as an inequality.

N The state monad

For simplicity, we drop the tags from the declarations. (A `newtype` declaration achieves essentially the same thing in Haskell.) We repeat the definitions for convenience:

```
type State s a = s -> (a, s)

instance Monad (State s) where
  return x = \s -> (x, s)
  f >>= g  = \s -> let (a, s') = f s in g a s'

instance MonadRec (State s) where
  mfix f = \s -> let (a, s') = f a s
                 in (a, s')
```

Axiom 1

```
mfix (return . h) = \s. let (a, s') = (return . h) a s in (a, s')
                  = \s. let (a, s') = (\s. (h a, s)) s in (a, s')
                  = \s. let (a, s') = (h a, s) in (a, s')
                  = \s. let a  = h a
                            s' = s
                        in (a, s')
                  = \s. let a = fix h in (a, s)
                  = \s. (fix h, s)
                  = return (fix h)
```

Axiom 2

First transform the lhs:

```
  mfix (\x. a >>= f x)
= \s. let (b, s') = (a >>= f b) s in (b, s')
= \s. let (b, s') = (\s''. let (c, s''') = a s'' in f b c s''') s
      in (b, s')
= \s. let (b, s') = let (c, s''') = a s in f b c s'''
      in (b, s')
= \s. let (b, s')   = f b c s'''
```

```
  mfix (\~(x, y). f y >>= \z. return (h z, z))
= \s. let (a, s') = (\~(x, y). f y >>= \z. return (h z, z)) a s
      in (a, s')
= \s. let (a, s') = (f (snd a) >>= \z. return (h z, z)) s
      in (a, s')
= \s. let (a, s') = (\s. let (b, s'')  = f (snd a) s
                             (c, s''') = return (h b, b)
                         in (c, s''')) s
      in (a, s')
= \s. let (a, s') = let (b, s'')  = f (snd a) s
                        (c, s''') = ((h b, b), s'')
                    in (c, s''')
      in (a, s')
= \s. let (a, s')   = (c, s''')
          (c, s''') = ((h b, b), s'')
          (b, s'')  = f (snd a) s
      in (a, s')
= \s. let (b, s'') = f b s
      in ((h b, b), s'')
= \s. let (a, s') = f a s
      in ((h a, a), s')
```

Now transform the rhs:

```
   mfix f >>= \z. return (h z, z)
= (\s. let (a, s') = f a s
       in (a, s'))
   >>= \z. return (h z, z)
= \s. let (b, s'') = let (a, s') = f a s
                     in (a, s')
      in ((h b, b), s'')
= \s. let (a, s') = f a s
      in ((h a, a), s')
```

Equation 6

First work on lhs:

```
  mfix (\x. f x >>= return . h)
= \s. let (a, s') = (\x. f x >>= return . h) a s
      in (a, s')
= \s. let (a, s') = (f a >>= return . h) s
      in (a, s')
= \s. let (a, s') = (\s'. let (b, s'') = f a s' in (return . h) b s'') s
      in (a, s')
= \s. let (a, s') = let (b, s'') = f a s in (h b, s'')
      in (a, s')
= \s. let (a, s')  = (h b, s'')
          (b, s'') = f a s
      in (a, s')
= \s. let (b, s'') = f (h b) s in (h b, s'')
= \s. let (a, s') = f (h a) s in (h a, s')
```

Now transform rhs:

```
      in ((a, b), s'')
= \u. let (a, s')  = (q, s')
          (q, s')  = f q u
          (b, s'') = g a s'
      in ((a, b), s'')
= \u. let (a, s') = f a u
          (b, s'') = g a s'
      in ((a, b), s'')
```

O State with exceptions

O.1 When the whole computation might fail

Recall the definitions (no tags):

```
newtype STE s a = s -> Maybe (a, s)

instance Monad (STE s) where
  return x = \s -> Just (x, s)
  f >>= g  = \s -> case f s of
                     Nothing      -> Nothing
                     Just (a, s') -> g a s'

instance MonadRec (STE s) where
  mfix f = \s -> let a = f b s
                     b = fst (unJust a)
                 in a
```

We first verify the monad laws:

Monad Axiom: return is the right unit:

```
f >>= return = \s. case f s of
                     Nothing      -> Nothing
                     Just (a, s') -> return a s'
             = \s. case f s of
                     Nothing      -> Nothing
                     Just (a, s') -> Just (a, s')
             = \s. f s
             = f
```

Monad Axiom: return is the left unit:

```
return x >>= f = \s. case return x s of
                       Nothing      -> Nothing
                       Just (a, s') -> f a s'
               = \s. case Just (x, s) of
                       Nothing      -> Nothing
                       Just (a, s') -> f a s'
               = \s. f x s
               = f x
```

```
  mfix (\x. a >>= f x)
= \s. let b = (a >>= f c) s
          c = fst (unJust b)
      in b
= \s. let b = case a s of
                 Nothing      -> Nothing
                 Just (d, s') -> f c d s'
          c = fst (unJust b)
      in b
```

Now look at rhs:

```
  a >>= \y. mfix (\x. f x y)
= \s. case a s of
        Nothing      -> Nothing
        Just (d, s') -> mfix (\x. f x d) s'
```

Apply both handsides to an arbitrary s, we get:

```
lhs = let b = case a s of
                 Nothing      -> Nothing
                 Just (d, s') -> f c d s'
          c = fst (unJust b)
      in b

rhs = case a s of
        Nothing      -> Nothing
        Just (d, s') -> mfix (\x. f x d) s'
```

Now, do a case analysis on $a\ s$. The cases $\bot$ and `Nothing` are immediate. When $a\ s =$ `Just` (d, s'), we have:

```
lhs = let b = f c d s'
          c = fst (unJust b)
      in b

rhs = mfix (\x. f x d) s'
    = (\s. let b = (\x. f x d) c s
               c = fst (unJust b)
           in b) s'
    = let b = f c d s'
          c = fst (unJust b)
      in b
```

which are identical.

Axiom 3

At this point, define an auxiliary function aux as follows:

```
aux q = case q of
          Nothing      -> Nothing
          Just (c, s') -> (g c >>= \w. return (c, w)) s'
```

Now continue the derivation:

```
= {use aux}
  \s. let a = aux ((f . fst) b s)
          b = fst (unJust a)
      in a
= \s. let a = aux ((f . fst) (fst (unJust a)) s) in a
= \s. let a = aux (((f . fst . fst . unJust) a) s) in a
= \s. let a = (aux . flip (f . fst . fst . unJust) s) a in a
= \s. fix (aux . flip (f . fst . fst . unJust) s)
```

Similarly, manipulate rhs:

```
  mfix f >>= \z. g z >>= \w. return (z, w)
= \s. case mfix f s of
        Nothing      -> Nothing
        Just (c, s') -> (g c >>= \w. return (c, w)) s'
= {use aux}
  \s. aux (mfix f s)
= {expand mfix}
  \s. aux (let a = f b s
               b = fst (unJust a)
           in a)
= \s. aux (let a = f (fst (unJust a)) s in a)
= \s. aux (fix (flip (f . fst . unJust) s))
```

Since both lhs and rhs are functions, to prove that lhs $\sqsubseteq$ rhs, we need to prove that when applied to an arbitrary s, the inequality is preserved. Furthermore, recalling fix $(f \cdot g) = f$ (fix $(g \cdot f)$) and that fix and aux are monotonic, we need:

$$\text{flip}\ (f \cdot \pi_1^2 \cdot \text{unJust})\ s \cdot \text{aux} \sqsubseteq \text{flip}\ (f \cdot \pi_1 \cdot \text{unJust})\ s$$

Again, since both hand sides are functions, we apply to an arbitrary A (of type `Maybe (a, s)`):
Case 1: $A = \bot$: $f \bot s \sqsubseteq f \bot s$.
Case 2: $A =$ `N`: $f \bot s \sqsubseteq f \bot s$.
Case 3: $A =$ `J` $\bot$: $f \bot s \sqsubseteq f \bot s$.
Case 4: $A =$ `J` (c, s'): Look at lhs:

```
  flip (f . fst . fst . unJust) s ((g c >>= \w. return (c, w)) s')
= flip (f . fst . fst . unJust) s
                      (case g c s' of
                         Nothing       -> Nothing
                         Just (d, s'') -> Just ((c, d), s''))
= case g c s' of
    undefined  -> f undefined s
    Nothing    -> f undefined s
    J (d, s'') -> f c s
```

Again, we first verify the monad laws:

Monad Axiom: return is the right unit:

```
f >>= return = \s. case f s of
                      (Nothing, s') -> (Nothing, s')
                      (Just a, s')  -> return a s'
             = \s. case f s of
                      (Nothing, s') -> (Nothing, s')
                      Just (a, s') -> (Just a, s')
             = \s. f s
             = f
```

Monad Axiom: return is the left unit:

```
return x >>= f = \s. case return x s of
                       (Nothing, s') -> (Nothing, s')
                       (Just a, s')  -> f a s'
               = \s. case (Just x, s) of
                       (Nothing, s') -> (Nothing, s')
                       (Just a, s')  -> f a s'
               = \s. f x s
               = f x
```

Monad Axiom: $\gg=$ is associative:
Look at lhs:

```
  f >>= \x. (g x >>= h)
= \s. case f s of
        (Nothing, s') -> (Nothing, s')
        (Just a, s')  -> (g a >>= h) s'
= \s. case f s of
        (Nothing, s') -> (Nothing, s')
        (Just a, s')  -> case g a s' of
                           (Nothing, s'') -> (Nothing, s'')
                           (Just b, s'')  -> h b s''
```

Now transform rhs:

```
  (f >>= g) >>= h
= \s. case (f >>= g) s of
        (Nothing, s') -> (Nothing, s')
        (Just a, s')  -> h a s'
= \s. case (case f s of
              (Nothing, s'') -> (Nothing, s'')
              (Just b, s'')  -> g b s'') of
        (Nothing, s') -> (Nothing, s')
        (Just a, s')  -> h a s'
= \s. case f s of
```

Now, do a case analysis on $a\ s$. The cases $\bot$ and $(\texttt{Nothing}, s')$ are immediate. When $a\ s = (\texttt{Just}\ d, s')$, we have:

```
lhs = let b = f c d s'
          c = unJust (fst b)
      in b

rhs = mfix (\x. f x d) s'
    = (\s. let b = (\x. f x d) c s
               c = unJust (fst b)
           in b) s'
    = let b = f c d s'
          c = unJust (fst b)
      in b
```

which are identical.

Axiom 3

```
  mfix (\~(x, _). mfix (\~(_, y). f x y))
= mfix (\u. mfix (\v. f (fst u, snd v)))
= \s. let a = (\u. mfix (\v. f (fst u, snd v))) b s
          b = unJust (fst a)
      in a
= \s. let a = mfix (\v. f (fst b, snd v)) s
          b = unJust (fst a)
      in a
= \s. let a = let c = f (fst b, snd d) s
                  d = unJust (fst c)
              in c
          b = unJust (fst a)
      in a
= \s. let a = c
          c = f (fst b, snd d) s
          d = unJust (fst c)
          b = unJust (fst a)
      in a
= \s. let a = f (fst b, snd d) s
          d = unJust (fst a)
          b = unJust (fst a)
      in a
= \s. let a = f (fst b, snd b) s
          b = unJust (fst a)
      in a
= \s. let a = f b s
          b = unJust (fst a)
      in a
= mfix f
```

O.2.1 Equations 5, 6, and 7

This version of the state-with-exceptions monad behaves exactly as the first version discussed above. Again, we first prove 7 holds as an inequality. The proof is very similar to the previous version. We start by looking at the lhs:

```
  flip (f . fst . unJust . fst) s ((g c >>= \w. return (c, w)) s')
= flip (f . fst . fst . unJust) s
                       (case g c s' of
                          (Nothing, s') -> (Nothing, s')
                          (Just d, s'') -> (Just (c, d), s''))
= case g c s' of
    undefined      -> f undefined s
    (Nothing, s'') -> f undefined s
    (Just d, s'')  -> f c s
```

The right hand side is simply $f\ c\ s$. Now, a simple case analysis on the value of $g\ c\ s'$, shows that lhs $\sqsubseteq$ rhs holds in all cases.

As in the previous case, the proof for equation 5 follows exactly the same pattern. The new definition of aux is:

```
aux q = case q of
          (Nothing, s')  -> (Nothing, s')
          (Just c, s')   -> (Just (c, h c), s')
```

with the proof obligation:

$$\text{flip}\ (f \cdot \pi_1 \cdot \text{unJust} \cdot \pi_1)\ s \cdot \text{aux} = \text{flip}\ (f \cdot \text{unJust} \cdot \pi_1)\ s$$

And the final case when $A = (\texttt{J}\ c, s')$, both hand sides yield: $f\ c\ s$, completing the proof for equation 5.

P The reader monad

We give details for the reader monad, which is just mentioned in the actual paper. The declarations are (again, no tags):

```
type Reader e a = e -> a

instance Monad (Reader e) where
 return x = \e -> x
 m >>= k  = \e -> k (m e) e

instance MonadRec (Reader e) where
  mfix f = \e -> let a = f a e in a
```

The definitions are very similar to that of the state monad, as expected. Typically, one fixes a certain type e (such as: `[(String, String)]`), to behave as the environment from which values are read, while some non-standard morphisms (such as `fetch` and `extend` with obvious definitions) are used to manipulate environments. We prove that the reader monad is recursive by showing that it (obviously) embeds into the state monad. The embedding is:

$$\epsilon\ r = \lambda s.(r\ s, s)$$

Notice that $\eta_r = \text{const}$, and $\eta_s = \lambda x.\lambda s.(x, s)$.

Q The output monad

We give details for the output monad. The declarations are:

```
newtype Out a = Out (a, String)

instance Monad Out where
  return x          = Out (x, "")
  Out ~(x, s) >>= f = let Out (y, s') = f x
                      in Out (y, s ++ s')

instance MonadRec Out where
  mfix f = fix (f . unOut)
         where unOut (Out (a, _)) = a
```

We prove that the output monad is recursive by showing that it embeds into the state monad. The embedding is:

$$\epsilon\ (x, s) = \lambda s'.(x, s' +\!\!+ s)$$

Equation 8: We need: $\epsilon \cdot \text{return}_o = \text{return}_s$.

$$\begin{aligned} \epsilon \cdot \text{return}_o &= \lambda s'.(x, s' +\!\!+ \text{""}) \\ &= \lambda s'.(x, s') \\ &= \eta_s \end{aligned}$$

Equation 9: We need to establish that $\epsilon\ (p \gg\!=_o h) = \epsilon\ p \gg\!=_s \epsilon \cdot h$.

```
eps ((a, s) >>= f) = eps (let (b, s') = f a in (b, s ++ s'))
                   = let (b, s') = f a in eps (b, s ++ s')
                   = let (b, s') = f a in \s''. (b, s'' ++ s ++ s')
```

and,

```
eps (a, s) >>= eps . f = (\s'. (a, s' ++ s)) >>= eps . f
                       = \s''. let (b, s''') = (a, s'' ++ s)
                               in eps (f b) s'''
                       = \s''. let (b, s') = (a, s''++ s)
                               in eps (f b) s'
                       = \s''. eps (f a) (s'' ++ s)
                       = \s''. (let (b, s') = f a
                                in \s'''. (b, s''' ++ s')) (s'' ++ s)
                       = \s''. let (b, s') = f a
                               in (b, s'' ++ s ++ s')
                       = let (b, s') = f a in \s''. (b, s'' ++ s ++ s')
```

Equation 10: We need: $\epsilon\ (\text{mfix}_o\ h) = \text{mfix}_s\ (\epsilon \cdot h)$. We use the following equivalent definition of mfix to simplify the proof:

Finally, we need ϵ to be monic. The obvious left inverse: $\epsilon^l\ f = f\ ""$ guarantees that it's split:

$$\begin{aligned}\epsilon^l\ (\epsilon\ (x, s)) &= \epsilon^l\ (\lambda s'.(x, s' +\!\!+ s)) \\ &= (x, "" +\!\!+ s) \\ &= (x, s)\end{aligned}$$

R The tree monad

In this section we look at the tree monad, which is just mentioned in the paper. The declarations are:

```
data T a = L a | F (T a) (T a)

unL (L a)   = a
lc  (F l _) = l
rc  (F _ r) = r

instance Monad T where
 return x      = L x
 (L a)   >>= f = f a
 (F l r) >>= f = F (l >>= f) (r >>= f)

instance MonadRec T where
 mfix f = case fix (f . unL) of
            L x   -> L x
            F _ _ -> F (mfix (lc . f)) (mfix (rc . f))
```

We start by proving the monad laws.

Monad Axiom: return is the right unit: $t \gg\!= \text{return} = t$. Induction on the structure of t:

Base Case 1: $t = \bot$. $\bot = \bot$.
Base Case 2: $t = \mathtt{L}\ x$. $\mathtt{L}\ x = \mathtt{L}\ x$.
Inductive Step: $t = \mathtt{F}\ l\ r$.

```
F l r >>= return = F (l >>= return) (r >>= return)
                 = F l r                              {I.H}
```

Monad Axiom: return is the left unit:

$$\text{return}\ x \gg\!= f = \mathtt{L}\ x \gg\!= f = f\ x$$

Monad Axiom: $\gg\!=$ *is associative:* $t \gg\!= \lambda x.(fx \gg\!= g) = (t \gg\!= f) \gg\!= g$: Induction on the structure of t:

Base Case 1: $t = \bot$. $\bot = \bot$.
Base Case 2: $t = \mathtt{L}\ x$. $f\ x \gg\!= g = f\ x \gg\!= g$.
Inductive Step: $t = \mathtt{F}\ l\ r$.

```
  F l r >>= \x. (f x >>= g)
= F (l >>= \x. (f x >>= g)) (r >>= \x. (f x >>= g))
= F ((l >>= f) >>= g) ((r >>= f) >>= g)              {I.H}
= (F (l >>= f) (r >>= f)) >>= g
= (F l r >>= f) >>= g
```

Now, lc, f and unL are all strict functions, and hence their composition is strict as well, resulting in $\perp$ for rhs.

Case 2: mfix $f = \mathtt{L}\ x$. Again, lhs is $\perp$ and we know that $f\ \perp$ is L of something (by monotonicity). For rhs, we have the same expansion above, and the case expression becomes:

$$\bigsqcup\{\perp, (\mathrm{lc} \cdot f \cdot \mathrm{unL})\ \perp = \perp, \ldots\} = \perp$$

Hence, both hand sides are, again, $\perp$.

Case 3: mfix $f = \mathtt{F}\ l\ r$.

```
lc (mfix f) = lc (case fix (f . unL) of
                    L x   -> L x
                    F _ _ -> F (mfix (lc . f)) (mfix (rc . f)))
```

Since mfix $f = \mathtt{F}\ l\ r$, the case expression should take its 2nd branch:

```
            = lc (F (mfix (lc . f)) (mfix (rc . f)))
            = mfix (lc . f)
```

Fifth Equality: Completely symmetric to the previous equality.

This completes the proof of the lemma. □

R.2 Proving that the tree monad is recursive

Axiom 1

```
mfix (return . h) = case fix (return . h . unL) of ...
                  = case return (fix (h . unL . return)) of ...
                  = case L (fix (h . id)) of
                      L x   -> L x
                      F _ _ -> ...
                  = L (fix h)
                  = return (fix h)
```

Axiom 2 By induction on the structure of a. The cases $\perp$ and L x are trivial. (When $a = \perp$, both lhs and rhs become $\perp$. When $a = \mathtt{L}\ u$, both become mfix $(\lambda x.f\ x\ u)$.) In the inductive case, we assume: $a = \mathtt{F}\ u\ v$, and proceed as follows:

```
  mfix (\lx. F u v >>= f x)
= mfix (\x. F (u >>= f x) (v >>= f x))
= case fix (\x. (F (u >>= f x) (v >>= f x)) . unL) of ...
= case fix (\x. F (u >>= f (unL x)) (v >>= f (unL x))) of ...
= {By monotonicity, the fix expression necessarily yields a fork.
   Notice that when fed bottom, it yields a fork}
  F (mfix (lc . (\x. F (u >>= f (unL x)) (v >>= f (unL x)))))
    (mfix (rc . (\x. F (u >>= f (unL x)) (v >>= f (unL x)))))
= F (mfix (u >>= f (unL x))) (mfix (v >>= f (unL x)))
= F (u >>= \y. mfix (\x. f x y)) (v >>= \y. mfix (\x. f x y))     {I.H}
= (F u v) >>= \y. mfix (\x. f x y)
```

R.3 The approxT lemma

Similar to approx lemma for lists, we have used the approxT lemma for trees. The function `approxT` is defined as:

```
approxT :: Integer -> T a -> T a
approxT (n+1) (L x)   = L x
approxT (n+1) (F l r) = F (approx n l) (approx n r)
```

The lemma we used in our proof is:

Lemma $\lim_{n\to\infty}$ approxT $n\ t = t$.

Proof By induction on the structure of the tree t.
Base Case 1: $t = \bot$: $\bot = \bot$.
Base Case 2: $t = \mathtt{L}\ x$:

$$\lim_{n\to\infty} \text{approxT}\ n\ (\mathtt{L}\ x) = \lim\{\bot, \mathtt{L}\ x, \mathtt{L}\ x, \ldots\} = \mathtt{L}\ x$$

Inductive step: $t = \mathtt{F}\ l\ r$: The induction hypotheses are: $\lim_{n\to\infty}$ approxT $n\ l = l$ and $\lim_{n\to\infty}$ approxT $n\ r = r$. The inductive step is:

$$\begin{aligned}
& \lim_{n\to\infty} \text{approxT}\ n\ (\mathtt{F}\ l\ r) \\
= \quad & \lim\{\bot, \mathtt{F}\ \bot\ \bot,\ \mathtt{F}\ (\text{approxT}\ 1\ l)\ (\text{approxT}\ 1\ r), \\
& \qquad\qquad \mathtt{F}\ (\text{approxT}\ 2\ l)\ (\text{approxT}\ 2\ r), \ldots\} \\
= \quad & \mathtt{F}\ (\lim_{n\to\infty} \text{approxT}\ n\ l)\ (\lim_{n\to\infty} \text{approxT}\ n\ r) \\
= \quad & \mathtt{F}\ l\ r
\end{aligned}$$

Which completes the proof of the approxT lemma. □

R.4 Equation 5

The tree monad satisfies equation 5, as all others do. Here's the proof:

Proof We're going to prove:

$$\forall k.\ \text{approxT}\ k\ (\text{mfix}\ (\text{map}\ \langle h,\ \text{id}\rangle \cdot f \cdot \pi_2)) = \text{approxT}\ k\ (\text{map}\ \langle h,\ \text{id}\rangle\ (\text{mfix}\ f))$$

The equivalence of this form and equation 5 was discussed in the list monad case. The proof proceeds by induction, the base case when $k = 0$ is trivial, both hand sides are $\bot$. The induction hypothesis states:

$$\forall f.\ \text{approxT}\ k\ (\text{mfix}\ (\text{map}\ \langle h,\ \text{id}\rangle \cdot f \cdot \pi_2)) = \text{approxT}\ k\ (\text{map}\ \langle h,\ \text{id}\rangle\ (\text{mfix}\ f))$$

and we try to prove:

$$\text{approxT}\ (k+1)\ (\text{mfix}\ (\text{map}\ \langle h,\ \text{id}\rangle \cdot f \cdot \pi_2)) = \text{approxT}\ (k+1)\ (\text{map}\ \langle h,\ \text{id}\rangle\ (\text{mfix}\ f))$$

Before proceeding, recall the definition of map in this case:

Now consider rhs. Since $f\ \bot$ is a fork, so is mfix f, yielding:

```
   approxT (k+1) (map <h, id> (mfix f))
=  approxT (k+1) (map <h, id> (F (lc (mfix f)) (rc (mfix f))))
=  approxT (k+1) (F (map <h, id> (lc (mfix f)))
                    (map <h, id> (rc (mfix f))))
=  F (approxT k ((map <h, id> . lc) (mfix f)))
     (approxT k ((map <h, id> . rc) (mfix f)))
=  F (approxT k (lc (map <h, id> (mfix f))))
     (approxT k (rc (map <h, id> (mfix f))))
```

Which completes the proof. □

R.5 Equations 6 and 7

The tree monad satisfies equation 6 for only strict h. It is possible to construct a counter-example, like in the list and `Maybe` cases. As in the case of the list monad, although we do not have an explicit proof, we conjecture that equation 7 will apply as an inequality.