Recursive Monadic Bindings

Levent Erkök John Launchbury

Oregon Graduate Institute of Science and Technology

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.

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)
```

Notice that variable y appears free in the first line of the do-expression: for the sake of this puzzle, assume that the do-notation binds variables recursively, much like `let` of Haskell or `letrec` of Scheme. Under this assumption, what should the value of `puzzle` be?

Our goal in this paper is to provide a general answer to this type of question. We develop a framework for recursion over the values resulting from monadic action. The discussion is set in the context of Haskell, but the ideas have wider applicability.

We first motivate the need for recursion in monadic computations, then we propose an extension to the do-notation supporting recursive bindings. We proceed to argue that the structure of the underlying monad specifies how the recursion should be performed and axiomatize the required behavior. The remainder of the paper contains a catalogue of monads that have recursion operators that satisfy our criteria. On the way, of course, we provide an answer to the puzzle (in Section 6.3).

2 Motivating problems

In this section, we present two examples to motivate the value of having recursive bindings in the do-notation. The first example is about sorting-networks with traces and is done in some detail. The second is from modeling circuits. We cover this much more briefly.

2.1 Sorting networks

A *sorting network* is a collection of comparators, connected in such a way that the output of the network is always the sorted permutation of its input [2]. Figure 1 shows an example that can sort four numbers. For each comparator, the wire to its right carries the maximum of its inputs, while the lower one carries the minimum.

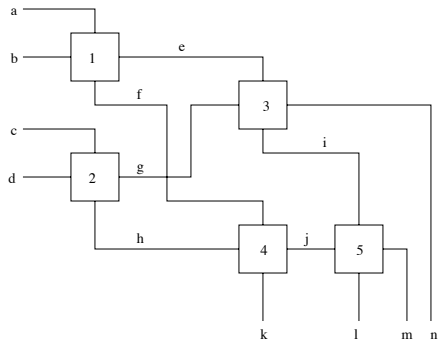

Figure 1: A sorting network of capacity 4

In this particular example, a, b, c, and d are the inputs, while k, l, m, and n are the outputs. A moment of thought will confirm that this network correctly sorts its input.

How can we implement a sorting network so that we not only get the values sorted, but also a transcript of the oper-

ations performed during sorting? We want each comparator unit to report on the operation it performed while sorting took place. The *output* monad springs to mind. We translate the sorting network in Figure 1 almost literally into Haskell code of Figure 2.

```
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

Here is a sample run:

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

A quick look at the trace reveals that it is consistent with the operation of the network for this particular 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 5 before unit 3 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)
```

That is, we replace the lines corresponding to units 3 and 5. Although this is the most intuitive thing to do, it is no longer valid in Haskell. The problem is that the variable i is not in scope when it's used.

How should we fix this? Obviously, in this simple case, the easiest solution would be to postprocess the output of the original circuit to obtain the required ordering. But this is not a very satisfactory solution in general. In particular, the failed attempt of reordering lines in the do-expression is very appealing. After all, the value that is computed by `sort4` (i.e. the quadruple representing the sorted permutation) doesn't depend on which order we observe the output. The attempt would have been successful, if only we had a way to bind variables recursively.

2.2 Resettable counter

The previous example didn't absolutely require recursive bindings because the values bound by the do-notation could be sequentially defined. This is not always the case. Our second example comes from the hardware-modeling domain. Microarchitectural design languages have been the target of programming language research in recent years because of the complexity of such designs. *Lava* [1] and *Hawk* [8, 12] are two recent systems designed to address this need. Lava uses monads intensively in modeling various circuit elements, and originally, Hawk used a similar monad-based approach as well. This approach is very flexible in translating specifications to VHDL or Verilog descriptions that could be used in producing real circuits. By just "plugging-in" the appropriate monad, the very same description can be used in simulation or in obtaining descriptions of the circuit in other languages. But this comes at a certain cost, as we explore here.

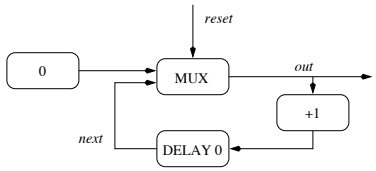

Figure 3: Resettable counter circuit

Hawk uses lazy lists to model signals flowing through circuits. The early monad-based implementation of Hawk used a `Circuit` monad, which is basically a combination of the state and output monads. Without going into details, we consider the circuit of Figure 3 as modeled in Hawk. Using the `Circuit` monad, we would like to model this circuit by:

```
counter :: Signal Bool -> Circuit (Signal Int)
counter reset = do next <- delay 0 inc
                   inc  <- lift1 (+1) out
                   out  <- mux reset zero next
                   zero <- lift0 0
                   return out
```

Notice that the description follows the circuit almost literally, but again, the program presented is not valid Haskell. The variables *inc, out* and *zero* are used before they are defined. Furthermore, because the definitions are cyclic, there is no way to serialize this program. The feedback present in the circuit causes the problem. Again, we need to be able to bind variables recursively in the do-notation. This problem is the main reason why the current implementation of Hawk does not use explicit monadic style.

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 bind [7]. 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:

```
                                mfix (\~BV -> do p1 <- e1
mdo p1 <- e1                                    ...
    ...          ===>                           pN <- eN
    pN <- eN                                    v  <- e
    e                                           return BV)
                                 >>= \BV -> return v
```

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 instance of `mfix` used in the translation is automatically deduced by the Haskell type system to be the instance at the output monad.

Syntactically, the last generator in a do-expression should be an expression. This condition can be relaxed in the μdo-notation: We might want to name the value of the last generator as well, so that we can use it in an earlier expression. In this variation we define the final result to be the value of the last generator. The translation in this case is:

```
                                mfix (\~BV -> do p1 <- e1
mdo p1 <- e1                                    ...
    ...          ===>                           v@pN <- eN
    pN <- eN                                    return BV)
                                 >>= \BV -> return v
```

Although the translation of μdo using *do* is similar to the translation of `letrec` using `let`, there is a difference: languages such as Haskell provide a generic definition of `fix` that works for all types. But there seems to be no appropriate generic definition of `mfix` that will work for all monads. Instead, we have to find an appropriate definition of `mfix` for each monad. To achieve some level of of uniformity, we stipulate some axioms that mfix must satisfy, and attempt to discover satisfactory definitions for each of the monads. We say that a monad is *recursive* when there is a definition of mfix satisfying our axioms. A μdo-expression is well-typed if the underlying monad is recursive and the translation is well-typed.

3.2 Repeated pattern variables and `let` bindings

In the translation of the μdo-notation, we explicitly prohibited a variable from being repeated in different patterns (repetition within the same pattern is disallowed following the usual Haskell convention). In the do-notation, a repeated variable has nothing to do with its previous binding: a new binding using the same name shadows the earlier one. If we allow repetitions in the μdo-notation, however, the translation would not treat them as independent. Furthermore, a repeated variable might change its type in the do-notation, but this will fail to type check for the μdo-notation. More importantly, one might expect that repeated variables will provide a way of constraining the values that they might take in the μdo-notation, which is not what the translation implies. Hence, even if the translation goes through, this might lead to misunderstandings.[1]

Allowing `let` bindings in the μdo-notation is another issue. In the do-notation, `let` bindings allow giving names to non-monadic computations in a convenient manner. Can we allow them in the μdo-notation as well? An obvious extension is to treat them as recursive bindings that are valid throughout the whole body, suggesting the following translation:

```
                                mfix(\~BV -> do ...1...
  mdo ...1...                                  let p1 = e1
      let p1 = e1   ===>                           ...
          ...                                      pN = eN
          pN = eN                               ...2...
      ...2...                                   v <- return e
      e                                         return BV)
                                 >>= \BV -> return v
```

The translation is similar to what we had before, except now the variables bound in $p_1 \ldots p_n$ appear in BV as well.

However, this poses some problems. In Haskell, `let` bound variables are polymorphic, while λ bound ones are monomorphic. This implies that the variables bound in $p_1 \ldots p_n$ are monomorphic in the code block marked by `...1...` but polymorphic in $e_1 \ldots e_N$, `...2...` and in e. This is not a desirable situation. As a concrete example consider the following translation:

[1] In a similar vein, it can be argued that the usual do-notation should not allow repetitions either. List comprehensions become especially horrible: `f x = [x | x <- [x..4], x <- [x..8]]` is a confusing (yet legal) Haskell function.

```
                          mfix (\(~(z, y, f, v)) ->
mdo                           do z <- return (f 2)
 z <- return (f 2)                y <- return (f 'a')
 y <- return (f 'a')              let f x = x
 let f x = x          ===>        v <- return ()
 return ()                        return (z, y, f, v))
                          >>= \(z, y, f, v) -> return v
```

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

3.3 Implementation

We have a straightforward implementation available obtained by modifying the source code for the Hugs system.[2] The 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 as described in the previous section, ignoring the possible typing problem.

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.[3] For a lazy language like Haskell, the definition is just:

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

Is there a generic definition for `mfix` as well? Inspired by the generic definition of `fix`, we consider the following (equivalent) definitions for `mfix`:

```
mfix :: Monad m => (a -> m a) -> m a
mfix f = mfix f >>= f
mfix f = do { x <- mfix f; f x }
mfix f = fix (join . map f)
```

Unfortunately, this definition is simply not appropriate. To see why not, we should specify what sort of properties we want `mfix` to have. First of all, we would expect a constant function, one that ignores its argument and always returns the same result, should have that result as its fixed point. This certainly holds for `fix`. We illustrate that this property does not hold for the `Maybe` monad with this definition. Here is the simplest test:

```
Main> mfix (const (Just 3))
ERROR: Control stack overflow
```

It is not hard to see why this definition fails to satisfy the required property: Consider the third version of the attempted definition. Since both `join` and `map f` are strict, so is their composition: `join . map f`. Since the least fixed-point of any strict function is $\bot$, the result is $\bot$ as well.

Looking closely at the default definition, we see the following: To compute the `mfix` of a function of type $a \to m\ a$, we first construct a function $m\ a \to m\ a$[4], and then compute the usual fixed-point of it. In other words, the fixed-point is computed not only for the values that the monad manipulates, but also for the effects of the execution that the monad generates.

Now, recalling the original intuition behind μdo-notation, we see that this is not what we wanted. We want the fixed-point computation to take place *only* on the values manipulated by the monad, while the effects and other computations remain untouched.

4.2 Axiomatizing `mfix`

So far, we have been using phrases like "a suitable definition of `mfix`" somewhat loosely. The time has come to make "suitable" precise. We give three axioms that `mfix` must satisfy in Figure 4. For notational aesthetics, we use $\rhd$ instead of `>>=`, and η instead of `return`.[5]

Axiom 1 reflects the intuition behind almost-pure computations: If the actual computation takes place in the pure world and the result is lifted using η, the fixed-point should be the fixed-point in the pure world lifted into the monad. Figure 5 is a pictorial representation of 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 $\rhd$. Since the variable x does not appear free in a, 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 diagrams are essentially the same, just the mfix box is replaced.

Axiom 3 states a useful fact about fixed-point computations involving more than one variable. 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. This axiom corresponds to Bekić's theorem for the usual fixed-point computations [17]. Figure 7 is a pictorial representation of this axiom. Notice that, again, both hand sides are essentially the same.

Now, we can precisely define what it means for a monad to be recursive:

Definition 4.1 *Recursive Monads.* A monad m is recursive if there is a function mfix :: $\forall a.(a \to m\ a) \to m\ a$ satisfying the mfix axioms.

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

[3] 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.

[4] This is the so-called *extension* of a function from values to computations to a function from computations to computations, see [13].

[5] Furthermore, in order to simplify the reading, we explicitly indicate which variables are used when. For instance, in axiom 2, the variable x does not appear free in the term a.

$$\begin{aligned}
\text{mfix}\ (\eta \cdot h) &= \eta\ (\text{fix}\ h) && (1)\\
\text{mfix}\ (\lambda x.a \rhd f\ x) &= a \rhd \lambda y.\text{mfix}\ (\lambda x.f\ x\ y) && (2)\\
\text{mfix}\ (\lambda\tilde{}(x,_).\text{mfix}\ (\lambda\tilde{}(_,y).f\ (x,y))) &= \text{mfix}\ f && (3)
\end{aligned}$$

Figure 4: Axioms for mfix

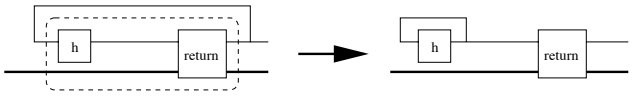

Figure 5: Interpreting axiom 1

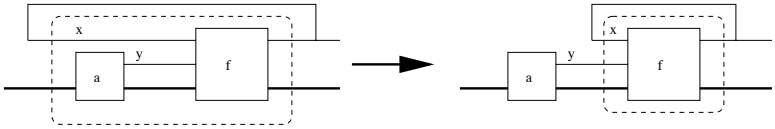

Figure 6: Interpreting axiom 2

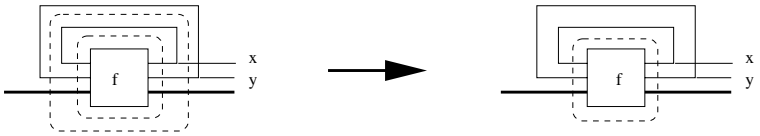

Figure 7: Interpreting axiom 3

Figure 8 presents mfix axioms using the μdo-notation. These equalities provide templates for compilers in simplifying μdo-expressions. In the second law, A represents any (type correct) term that might refer to the variable y. However, a should be a term mentioning neither y nor any other variables bound in A. This law can be used repeatedly to convert a μdo-expression to a pure do-expression, in case there are no recursive bindings.

4.3 Derived equivalences

A direct corollary to first two mfix axioms guarantees an expected property of constant functions:

Corollary 4.2 mfix $(\lambda x.a) = a$, provided x does not appear free in a.

Furthermore, the polymorphic nature of `mfix` provides further properties. By the parametricity theorem [15], we have:

Theorem 4.3 $\forall s : A \to B,\ f : A \to m\ A,\ g : B \to m\ B$, if $g \cdot s = \text{map}\ s \cdot f$ then map s (mfix$_A$ f) = mfix$_B$ g. (If mfix is defined recursively, s needs to be strict.)

As specific instances of this theorem, we obtain the following two corollaries:

Corollary 4.4 The following equation holds for any recursive-monad:

$$\begin{aligned}
&\text{mfix}\ (\lambda\tilde{}(x,y).f\ y \rhd \lambda z.\eta\ (h\ z,z))\\
&\quad = \quad \text{mfix}\ f \rhd \lambda z.\eta\ (h\ z,z) \qquad (4)
\end{aligned}$$

Considering equation 4, we see that the function f only refers to y and it is fed back exactly its own result. However, the fixed-point value also gets acted upon by a *pure* function h, whose result is ignored by f. (There is also a completely symmetric equation where h acts on the second component of the pair while f uses only x.) This equation is important in the sense that it tells us exactly when we can pull certain "pure" computations out of an mfix loop. Figure 9 depicts the situation.

Finally, we can move pure computations around:

Corollary 4.5 Provided h is strict, the following equation holds for any recursive-monad:

$$\begin{aligned}
&\text{mfix}\ (\lambda x.f\ x \rhd \eta \cdot h)\\
&\quad = \quad \text{mfix}\ (\lambda x.\eta\ (h\ x) \rhd f) \rhd \eta \cdot h \qquad (5)
\end{aligned}$$

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 10 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. 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. This law will only hold as an

$$
\begin{array}{rcl}
\mu\text{do}\ \{x \leftarrow \eta\,(h\ x)\} & = & \eta\,(\text{fix}\ h) \\
\mu\text{do}\ \{y \leftarrow a;\ A\} & = & \text{do}\ \{y \leftarrow a;\ \mu\text{do}\ A\} \\
\mu\text{do}\ \{\tilde{}(x,\_) \leftarrow \mu\text{do}\ \{\tilde{}(\_,y) \leftarrow f\ (x,y)\};\ A\} & = & \mu\text{do}\ \{\tilde{}(x,y) \leftarrow f\ (x,y);\ A\}
\end{array}
$$

Figure 8: Laws for the μdo-notation.

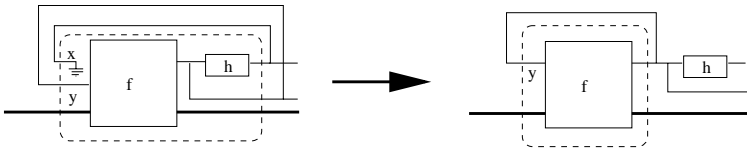

Figure 9: Interpreting equation 4

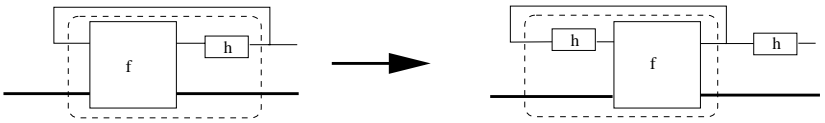

Figure 10: Interpreting equation 5

inequality (i.e. lhs $\sqsubseteq$ rhs), when h is non-strict. (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 4.5 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: $f \diamondsuit g = \lambda x.f\ x \rhd g$, where x does not occur free in f or g. Now, equation 5 becomes ($\diamondsuit$ binds less tightly than $\cdot$):

$$\text{mfix}\ (f \diamondsuit \eta \cdot h) = \text{mfix}\ (\eta \cdot h \diamondsuit f) \rhd \eta \cdot h$$

4.4 Shrinking from right

Corollary 4.4 states exactly when we are allowed to pull a pure computation out from the right-hand-side of a $\rhd$. Can we pull out impure computations as well? Consider Figure 11 which depicts the case when g is allowed to make computational effects. Since the value produced by g is ignored in the fixed-point computation, one might expect pulling g out of the loop wouldn't change the value of the computation. Indeed, our early axiomatization stipulated this property. However, it turns out that the equality is too strong for many monads. A problem arises because the monadic part of the computation in g might interfere with the fixed-point computation, possibly changing the termination behavior. Therefore, the best we could hope for is an inequality in the general case, that is:

$$
\begin{array}{rl}
& \text{mfix}\ (\lambda\tilde{}(x,y).f\ x \rhd \lambda z.g\ z \rhd \lambda w.\eta\ (z,w)) \\
\sqsubseteq & \text{mfix}\ f \rhd \lambda z.g\ z \rhd \lambda w.\eta\ (z,w) \qquad (6)
\end{array}
$$

We will note the examples in which the equality holds or fails in Section 6.

4.5 A reflection on mfix axioms

We have tried to axiomatize how recursion over the values of monadic actions should behave. All three axioms emerge from our intuitions for recursive monadic computations. At this point, however, our approach is more definitive than explanatory in its nature.

In this regard, the extent to which our axiomatization is successful will be determined by practice. Our axioms could be deemed appropriate if they rule out useless definitions of `mfix` and admit only those that are meaningful in practice. Notice that we do not require a unique definition of `mfix` (if any) for a given monad: different applications using the same monad might benefit from different definitions of `mfix`. Our concern is in trying to specify the common core of monadic fixed-point computations. The major points are:

- The fixed-point computation should be performed only over the values.
- Effects of monadic functions should neither be duplicated nor lost in a fixed-point computation. The usual laws of "demand driven" evaluation and the structure of the underlying monad will determine when, if ever, these effects will be performed.
- In the case when recursive bindings are not present, a μdo-expression should behave exactly like a do-expression.

Our axioms try to capture these points formally. Some monads might, of course, satisfy more laws (such as shrinking from right), and users might exploit these facts in programs. On the other hand, we believe that our axiomatization captures the minimal common core that should be satisfied by any monad in order to perform recursive computations over the results of monadic actions.

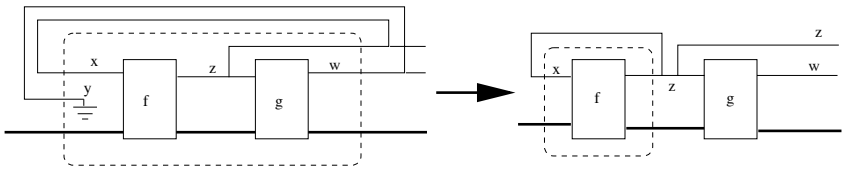

Figure 11: Interpreting equation 6

5 Embedding monads

Whenever we want to establish that a monad is recursive, we need to prove that the axioms are satisfied by the proposed definition of `mfix`. In practice, we have found ourselves repeating essentially the same proof for many different monads. Monad *embeddings* lets us eliminate much of the duplicated work. We first recall the definition of monad-homomorphisms [16]:

Definition 5.1 *Monad homomorphism.* Let $(m, \eta_m, \rhd_m)$ and $(n, \eta_n, \rhd_n)$ be two monads and let $\epsilon : m \to n$ be a family of functions (one for each type a, $\epsilon_a : m\ a \to n\ a$) such that:

$$\epsilon \cdot \eta_m = \eta_n \quad (7)$$

$$\epsilon_b\ (p \rhd_m h) = \epsilon_a\ p \rhd_n \epsilon_b \cdot h \quad (8)$$

where, in the second equation: $p : m\ a$ and $h : a \to m\ b$. Then, ϵ is called a monad homomorphism.

We extend the definition to cover the recursive case:

Definition 5.2 *Recursive-monad homomorphism.* Let $(m, \eta_m, \rhd_m)$ and $(n, \eta_n, \rhd_n)$ 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:

$$\epsilon\ (\mathrm{mfix}_m h) = \mathrm{mfix}_n\ (\epsilon \cdot h) \quad (9)$$

An *embedding* of one monad into another is simply a monic monad-homomorphism (i.e. it should be injective). We will see concrete examples of embeddings in the next section.

Theorem 5.3 Let $(m, \eta_m, \rhd_m)$ and $(n, \eta_n, \rhd_n)$ be two recursive-monads and let $\epsilon : m \to n$ be an embedding. If mfix_n satisfies any mfix axiom then mfix_m will satisfy that axiom as well.

The proof is by simple equational reasoning. We also note that equations 4, 5 and 6 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 holds even if h is non-strict, and equation 6 is satisfied as an equality.

6.2 Maybe

The `Maybe` monad, the monad of exceptions, has the following `MonadRec` declaration:

```
instance MonadRec Maybe where
  mfix f = fix (f . unJust)
       where unJust (Just x) = x
```

The proof that the `Maybe` monad is recursive follows from the fact that it can be embedded into the `List` monad, as described in Section 5. Before studying the `List` monad, we state a lemma classifying mfix of functions for the `Maybe` monad.

Lemma 6.1 The `Maybe` instance of mfix satisfies (`J` abbreviates `Just`, `N` abbreviates `Nothing`):

$$\mathrm{mfix}\ f = \perp \longleftrightarrow f\ \perp = \perp$$

$$\mathrm{mfix}\ f = \mathtt{N} \longleftrightarrow f\ \perp = \mathtt{N}$$

$$\mathrm{mfix}\ f = \mathtt{J}\ \perp \longleftrightarrow f\ \perp = \mathtt{J}\ \perp$$

$$\mathrm{unJust}\ (\mathrm{mfix}\ f) = \mathrm{fix}\ (\mathrm{unJust} \cdot f)$$

The first three equivalences exactly determine when the fixed-point is $\perp$, `Nothing` or `Just` $\perp$. The last equality is a consequence of the definition of mfix. An implication of these equations is that mfix of a function f of type $a \to \mathrm{Maybe}\ a$ is $f\ \perp$, whenever a is a flat domain.

The `Maybe` monad demonstrates the need for a strict h for Equation 5 to hold in general. Consider the following example:

```
f :: [Int] -> Maybe [Int]
f (x:_) = Just [x]

h :: [Int] -> [Int]
h xs = 1:xs
```

On the lhs, we get $\bot$, while rhs yields `Just` $[1,1]$. This is due to the fact that f performs a case analysis to see if its argument is a non-empty list. When the fixed-point computation starts, f first receives $\bot$ as the argument and produces $\bot$. Since $\rhd$ for the `Maybe` monad is strict in its first argument, the whole computation fails. On the rhs, however, f first receives $1 : \bot$, and produces `Just` $[1]$, and the computation proceeds.

Similarly, we revisit equation 6 of Section 4.4 in the context of the `Maybe` monad. Consider the following example:

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

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

For this example, lhs of Equation 6 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. If we look more closely, we see that the problem lies within the fact that $\rhd$ 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 $\rhd$ 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 6 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 nondeterministic 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. (The call to `head` will fail.) The definition given in the instance declaration handles this problem. Analogous to Lemma 6.1, we have:

Lemma 6.2 The `List` instance of mfix satisfies:

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

The first two equivalences imply that when $f\ \bot$ is a cons-cell (i.e. of the form $(x : xs)$), then mfix f is also a cons-cell. Using this lemma, proving that mfix axioms hold is a tedious but straightforward exercise.

The embedding of the `Maybe` monad into the `List` monad simply takes `Nothing` to `[]` and `Just x` to [x]. By theorem 5.3, we do not expect the `List` monad to satisfy equation 5 when h is non-strict and equation 6 as an equality since we know that the `Maybe` monad does not have these properties. In deed, it is possible to construct counterexamples for the `List` monad as well.

We can finally solve the puzzle posed in the introduction. First, using our intuition for the `List` monad and recursive bindings, we try to derive the solution. It is well known that the do-notation and the usual list comprehensions of Haskell coincide for the `List` monad. Hence, we can think of the puzzle as the following list comprehension:

```
[x+y | (x, z) <- [(y, 1), (y^2, 2), (y^3, 3)],
       Just y <- map isEven [z+1 .. 2*z]]
```

Notice that this list comprehension is still not valid Haskell: The variable y is used before its value is generated. Nevertheless, we apply the usual rules for decomposing list comprehensions [16]. We obtain:

```
concat [   [x+y | Just y <- map isEven [z+1 .. 2*z]]
         | (x, z) <- [(y, 1), (y^2, 2), (y^3, 3)] ]
```

Notice that we have a nested comprehension now. At this point, we can expand the outer comprehension for each assignment to (x, z). This step is where we use our intuition for the recursive bindings for interpreting the free variable y: We substitute it for x symbolically. This yields:

```
concat [ [y  +y | Just y <- map isEven [2 .. 2]],
         [y^2+y | Just y <- map isEven [3 .. 4]],
         [y^3+y | Just y <- map isEven [4 .. 6]] ]
```

Now, routine calculations yield: `[4, 20, 68, 222]`.

When we run the puzzle using the μdo modifed version of Hugs (after replacing the keyword `do` with `mdo`), we get exactly the same answer.

6.4 State

The `State` monad is used to capture computations that involve mutable variables [9, 10]. Here are the definitions:

```
newtype State s a = ST { unST :: (s -> (a, s)) }

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

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

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 is satisfied even when h is not strict and equation 6 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 5 and they don't satisfy equation 6 as an equality. This is hardly surprising since the `Maybe` monad behaves like this as well.

6.6 Other monads

We take a brief look at a couple of other monads without going into much detail. The `Reader` (or *environment*) monad is a version of the `State` monad where we only read the state without ever changing it [6]. The obvious embedding into the `State` monad suffices to prove that the `Reader` monad is recursive. In fact, the work on implicit parameters [11] provides an *implicit* recursive `Reader` monad in Haskell where the usual `let` construct expresses recursive computations, (implicit parameters provide the mechanism for accessing values within the monad).

The output monad, as described in Section 2.1, is recursive also. It also embeds into the `State` monad. The definition of mfix is:

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

The `Tree` monad [6] is recursive. The definition of mfix closely mimics that of the `List` monad. Unlike lists, however, these trees are never empty and hence the `List` monad cannot be embedded into them. It is, however, not clear what sort of applications can benefit from recursive bindings for the `Tree` monad.

Two other recursive-monads that are very well known in the Haskell community are the internal input/output (`IO`) and state (`ST`) monads. The library functions `fixIO` and `fixST` correspond to our mfix for the `IO` and `ST` monads, respectively. These monads are "internal" in the sense that, unlike others, their implementations use destructive updates and hence need to be defined as primitives. This prevents us from constructing explicit proofs, but the Haskell folklore suggests that they indeed satisfy our axioms.

The continuation monad continues to cause us grief. We have been unable to produce a viable definition for mfix in this case. Furthermore, in the Scheme case—when state and continuations coexist—the standard definition of `letrec` (Scheme's equivalent of mfix) seems to lack the appropriate uniformity properties implied by axiom 2 and property 6.

7 Related work

O'Haskell is a concurrent, object oriented extension of Haskell designed for addressing issues in reactive functional programming [14]. One application of O'Haskell is in programming layered network protocols. Each layer interacts with its predecessor and successor by receiving and passing information in both directions. In order to connect two protocols that have mutual dependencies, one needs a recursive knot-tying operation. Since O'Haskell objects are monadic, recursive monads are employed in establishing connections between objects. O'Haskell adds a keyword `fix` to the do-notation whose translation is a simplified version of ours. The O'Haskell work, however, does not try to axiomatize or generalize the idea any further.

Although we limited our attention to monadic computations, recursion makes sense in the more general setting of *arrows* as well [5]. Recently, Ross Paterson axiomatized `arrowFix`, the arrow version of `mfix`, which turns out to be quite similar to our formulation. He draws links with Hasegawa's work in traced monoidal categories that provide a general framework for recursion and cyclic sharing [4].

Recent work by Friedman and Sabry tries to address the problem from a different angle [3]. Rather than an axioma-

tization, their work suggests combining monads with a state monad and performing a generic recursion computation in this combined world. The semantics of the recursion is then defined by this implementation. Since the recursion is performed in the combined monad, programs and values have to be translated to and from this new world.

8 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.

9 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] Friedman, D., and Sabry, A. Recursion in monads, or when is recursion a computational effect? Submitted to ICFP 2000.

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

[5] Hughes, J. Generalising monads to arrows. *Science of Computer Programming* (To appear.).

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

[7] Launchbury, J. Lazy imperative programming. In *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, DK, SIPL '92* (1993), pp. 46–56.

[8] Launchbury, J., Lewis, J., and Cook, B. On embedding a microarchitectural design language within Haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99)* (1999), pp. 60–69.

[9] Launchbury, J., and Peyton Jones, S. L. Lazy functional state threads. *ACM SIGPLAN Notices 29*, 6 (June 1994), 24–35.

[10] Launchbury, J., and Peyton Jones, S. L. State in Haskell. *Lisp and Symbolic Computation 8*, 4 (Dec. 1995), 293–341.

[11] Lewis, J., Shields, M., Meijer, E., and Launchbury, J. Implicit parameters: Dynamic scoping with static types. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)* (2000).

[12] Matthews, J., Cook, B., and Launchbury, J. Microprocessor specification in Hawk. In *Proceedings of the 1998 International Conference on Computer Languages* (1998), IEEE Computer Society Press, pp. 90–101.

[13] Moggi, E. Notions of computation and monads. *Information and Computation 93*, 1 (1991).

[14] Nordlander, J. *Reactive Objects and Functional Programming*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 1999.

[15] Wadler, P. Theorems for free! In *FPCA'89, London, England*. ACM Press, Sept. 1989, pp. 347–359.

[16] Wadler, P. Comprehending Monads. In *LISP'90, Nice, France*. ACM Press, 1990, pp. 61–78.

[17] Winskel, G. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing series. MIT Press, Feb. 1993.