Monads are Trees with Grafting

Dan Piponi
A Neighborhood of Infinity

January 1, 2010

1 Goals and Prerequisites

This article is intended to give an elementary introduction to an aspect of monads not covered in most introductions. The reader is expected to know some basics of Haskell, for example what a type class is and what a lambda term is. They are also expected to be familiar with the usual notion of a tree from computer science. No category theory mentioned anywhere but in this sentence.

The original source code to this document is written in literate Haskell and can be executed with ghc.

2 Trees

Haskell type classes are interfaces shared by different types, and Haskell's *Monad* type class is no different. It describes an interface common to many types of tree structure, all of which share the notion of a *leaf node* and *grafting.*

Let's start by looking at the type class definition:

```
class Monad m where
  return :: a → m a
  (>>=) :: m a → (a → m b) → m b
```

Here m is a type constructor. Given a type a, $m\ a$ is a new type with these two functions. We can make an instance of this class by defining a binary tree type:

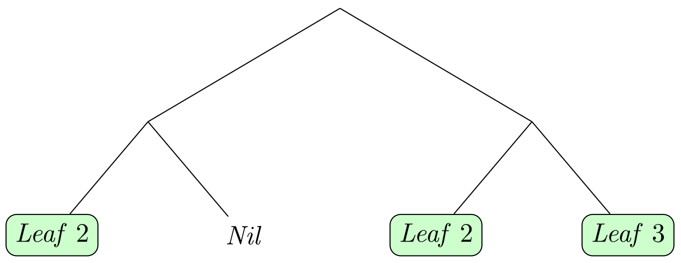
```
data Tree a = Fork (Tree a) (Tree a) | Leaf a | Nil deriving Show
```

Tree takes a type a and makes a new type from it *Tree* a. Elements of type *Tree* a are either forks with two subtrees, leaves containing a single element of type a, or empty trees called *Nil*.

Here's a typical expression representing a tree:

```
tree1 = Fork
  (Fork (Leaf 2) Nil)
  (Fork (Leaf 2) (Leaf 3))
```

We can draw this in the standard way:



Although there are two different kinds of terminal node, *Leaf*s and *Nil*s, our interest will be focussed on the *Leaf*s.

To show how the monad interface works we can start defining a *Monad* instance for *Tree*:

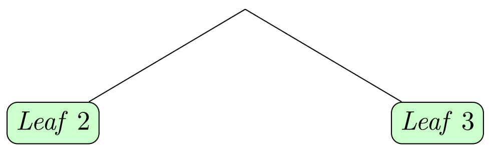
```
instance Monad Tree where
  return a = Leaf a
```

The function *return*, despite the name, is nothing more than a function for creating *Leaf* nodes. The next function is a grafting operation:

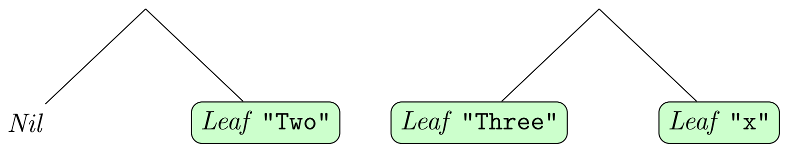
```
Nil      >>= f = Nil
Leaf a   >>= f = f a
Fork u v >>= f = Fork (u >>= f) (v >>= f)
```

The idea is that given a tree we'll replace every leaf node with a new subtree. We need a scheme to be able to specify what trees we are grafting in to replace which leaves. One way to do this is like this: we'll use the value stored in the leaf to specify what tree to graft in its place, and we'll make the specification by giving a function mapping leaf values to trees. I'll illustrate it pictorially first, with a simple tree. Consider:

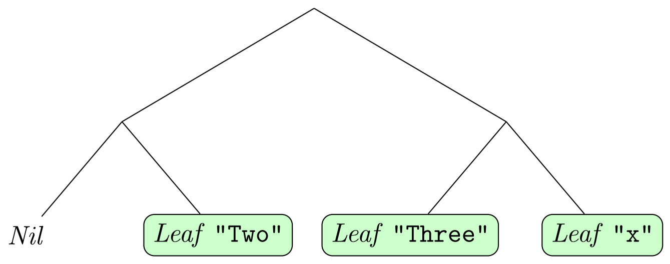
```
tree2 = Fork (Leaf 2) (Leaf 3)
```



Now I want to graft these two trees into *tree2* so that the left one replaces *Leaf* 2 and the right one replaces *Leaf* 3:



The result should be:



We carry this out by writing a function that maps 2 to the left tree and 3 to the right tree. Here's such a function:

```
f 2 = Fork Nil (Leaf "Two")
f 3 = Fork (Leaf "Three") (Leaf "String")
```

We can now graft our tree using:

```
tree3 = tree2 >>= f
```

I hope you can see that the implementation of $\gg\!=$ does nothing more than recursively walk the tree looking for leaf nodes to graft.

Instances of *Monad* can be viewed as trees similar to this.

3 Computations

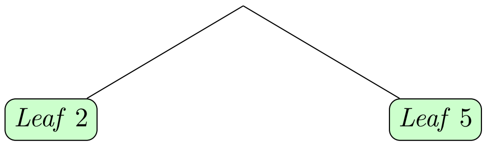
"Monads turn control flow into data flow, where it can be constrained by the type system." Oleg Kiselyov

If this interface were merely for building tree structures it wouldn't be all that interesting. Where it starts to get useful is when we use trees to represent different ways to organise a computation. For example, consider the kind of combinatorial search involved in finding the best move in a game. Or consider decision-tree flowcharts or probability trees. Even simple ordered linear sequences of operations form a kind of degenerate tree without branching. The monad interface can be used with all of these structures giving a uniform way of working with them.

4 Combinatorial Search

Let's start by putting the *Tree* example above to work. Combinatorial search trees can quickly grow too large to fit on a page so I'm deliberately going to pick a particularly simple problem to solve so that every part of it can be laid bare.

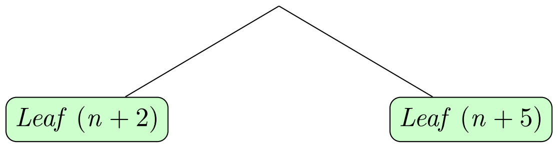
Let S be the set $\{2, 5\}$ and suppose we wish to find all of the possible ways we can form the sum of three numbers chosen from this set (with possible repeats). For example $2 + 5 + 2$ or $5 + 5 + 5$. We can break this problem down into three stages, picking a number at each stage. We can draw a diagram representing the possibilities as follows:



The Haskell code is:

$\quad tree4 = Fork\ (return\ 2)\ (return\ 5)$

Now we wish to construct the tree for the next stage. We want to replace the left node with a subtree that represents the two possibilities we might get given that we picked 2 at the first stage. Similarly we want to replace the right leaf with the possibilities that start with 5. In other words, in both cases we want to replace *Leaf n* with the tree



In other words, we want to graft with the function

$\quad choose\ n = Fork\ (Leaf\ (n+2))\ (Leaf\ (n+5))$

Our two stage tree is given by

$\quad stage2 = tree4 \ggg\!= choose$

Now we want to replace each of these leaf nodes with a new subtree. We can reuse our rule to get

$\quad stage3_1 = stage2 \ggg\!= choose$

If you run the code you'll see that $stage3_1$ has all of the possibilities stored in the tree and we have solved our problem.

We can reorganise this code a little. A helper $fork_1$ function saves on typing:

$\quad fork_1\ a\ b = Fork\ (Leaf\ a)\ (Leaf\ b)$

I've implemented *choose* as a separate function but we could write out everything longhand as follows:

$$\begin{aligned}
&stage3_2 = (fork_1\ 2\ 5 \qquad \ggg\!= \lambda a \rightarrow \\
&\quad fork_1\ (a+2)\ (a+5)) \ggg\!= \lambda b \rightarrow \\
&\quad fork_1\ (b+2)\ (b+5)
\end{aligned}$$

I've simply substituted lambda terms for the function *choose*. We can now see the three stages clearly as one line after another. We can read this code from top to bottom as a sequence of three operations interpreting the $fork_1$ function as something like the Unix $fork_1$ function. After a fork, the remainder of the lines of code are executed *twice*, each time using a different value. Writing it out fully makes it clear that we can easily change the choice at each line, for example to use sets other than $\{2,5\}$.

An important thing to notice about our three stage tree is that there were two ways of building it. I first built a two stage tree and then grafted a one stage tree into each of its leaves. But I could have built a one stage tree and substituted a two stage tree into each of its leaves.

Our alternative code looks like this:

$$\begin{aligned}
&stage3_3 = fork_1\ 2\ 5 \quad\quad \gg\!= \lambda a \rightarrow\\
&\quad fork_1\ (a+2)\ (a+5) \gg\!= \lambda b \rightarrow\\
&\quad fork_1\ (b+2)\ (b+5)
\end{aligned}$$

It's almost the same, I just removed a pair of parentheses.

We can try to step back a bit and think about what that code means. Each time we see ... $\gg\!= \lambda a \rightarrow$ ... we can think of a as a handle onto the leaves of the tree on the left and the tree on the right is what those leaves get replaced with. If we're going to do lots of grafting with lambda terms like this then it'd be nice to have special syntax. This is exactly what Haskell **do**-notation provides. After a **do**, this fragment of code can be written as

$$\begin{aligned}
&a \leftarrow ...\\
&...
\end{aligned}$$

So we can rewrite our code yet again as

$$\begin{aligned}
&stage3_4 = \mathbf{do}\\
&\quad a \leftarrow fork_1\ 2\ 5\\
&\quad b \leftarrow fork_1\ (a+2)\ (a+5)\\
&\quad fork_1\ (b+2)\ (b+5)
\end{aligned}$$

Now it really is looking like imperative code with a Unix-like fork function. We have a very straightforward interpretation of **do** notation. The line

$$a \leftarrow ...$$

means exactly this: using a to represent the value in the leaf, replace all of the leaves on the right hand side with the tree defined by the rest of this **do** block. The sequence of lines in a **do** block are a sequence of operations to graft subtrees one below the other.

5 The Monad Laws

There are some properties that we can expect to hold for all trees. The first of these is this: if we graft with the rule $\lambda a \rightarrow return\ a$ then we're just replacing a leaf with itself. This rule is the first *monad law* and a monad isn't a monad unless it holds. We can write it using **do** notation as:

$$\begin{aligned}
&x \equiv \mathbf{do}\\
&\quad a \leftarrow x\\
&\quad return\ a
\end{aligned}$$

That was a rule for grafting $\lambda a \rightarrow return\ a$ into a tree. We can also consider grafting a subtree into a single leaf. This should simply replace the leaf with the tree with no trace of the leaf left behind. So we get the second monad law:

$$\begin{aligned}
&\mathbf{do}\\
&\quad a \leftarrow return\ a\\
&\quad f\ a\\
&\ \equiv\\
&f\ a
\end{aligned}$$

Consider again the two ways of grafting the three stage combinatorial search tree we built above. That again can be expressed purely in the language of monads making no specific reference to the *Tree* type. It is:

```
    do
      y ← do
        x ← m
        f x
      g y
    ≡
    do
      x ← m
      y ← f x
      g y
```

The first expression builds a two stage tree y and then grafts into that using the function g. The second expression grafts a two stage tree directly into the tree m. We'd expect this to hold for any kind of tree and it is known as the third monad law.

The monad laws just express the property that $\gg\!=$ is intended to act like tree grafting.

6 Reduced Trees

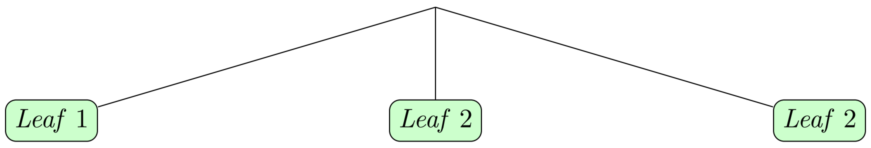
Suppose we used the *Tree* monad to perform a combinatorial search and it resulted in the tree

```
    Fork (Leaf 1) (Fork (Leaf 2) (Leaf 3))
```

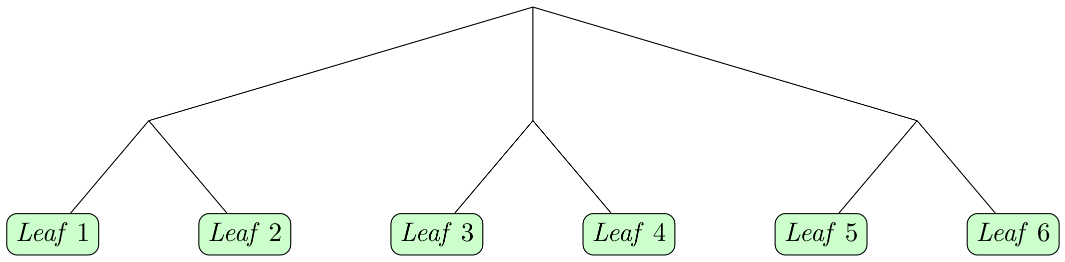
Chances are, this contains more information than we needed. If we only need the leaf values, 1, 2 and 3 then have no need for the tree structure. We could write a function to run through our tree extracting all of the values from it:

```
    runTree :: Tree a → [a]
    runTree Nil = []
    runTree (Leaf a) = [a]
    runTree (Fork a b) = runTree a ++ runTree b
```

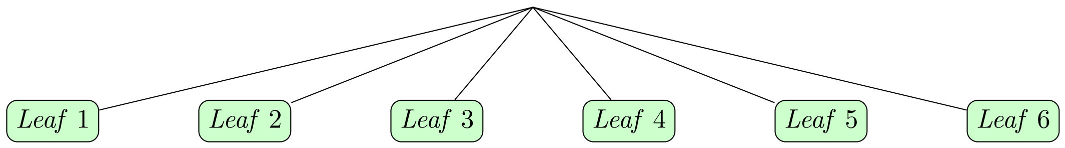
It seems a little inefficient to build a tree and then discard it at the end. It would be more efficient to build the list directly as we go along and never make an intermediate tree. Another way to look at it is to consider that a list is itself a type of tree. You can think of the list elements as being the children of a root, but that the children have no children of their own. Here's a picture of the list $[1, 2, 3]$:



The problem now is that grafting seems like it ought to make the tree deeper, but then it'd no longer be a simple list. The solution is to define grafting for lists in such a way that the resulting tree is flattened back out again. So a tree like



should be immediately flattened out to



We can implement this as follows:

```
instance Monad [] where
  return a = [a]
  a ≫= f   = concat (map f a)
```

Our leaves are simply singleton lists, and the grafting operation temporarily makes a deeper list of lists that is flattened out to a single list using *concat*.

Now we can reimplement $fork_1$ as

```
fork₂ a b = [a, b]
```

and reimplement our search:

```
stage3₅ = do
  a ← fork₂ 2 5
  b ← fork₂ (a + 2) (a + 5)
  fork₂ (b + 2) (b + 5)
```

We don't need the $fork_1$ function. we could have simply written:

```
stage3₆ = do
  a ← [2, 5]
  b ← [a + 2, a + 5]
  [b + 2, b + 5]
```

Of course we can now put more than two elements into those lists for more complex searches. But there's one more transformation I want to perform on this code:

```
stage3₇ = do
  a ← [2, 5]
  b ← [2, 5]
  c ← [2, 5]
  return (a + b + c)
```

I hope you can see how this works. The last two lines build a list parameterised by the values a and b. That list is grafted into the list made at the $b \leftarrow$ ... line. And that list is grafted into the list made at the $a \leftarrow$ ... line.

There's another way of looking at this code. It's a lot like imperative code. For example the Python

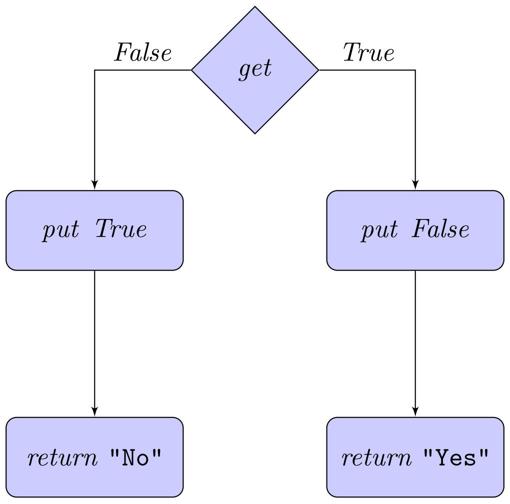
```
results = []
for a in [2, 5]:
  for b in [2, 5]:
      for c in [2, 5]:
          results.append(a+b+c)
```

This is a common characteristic of many types of monad: we build a tree structure that represents a computation (for example, an imperative one with loops) and then interpret it using something like *runTree*. In many cases we can do the interpretation as we go along and so we don't need a separate interpretation step at the end, as we just did with lists.

7 Flowcharts and State

To illustrate how flexible monads are we'll now look at a completely different type of tree structure that also represents a type of computation and that also shares the *Monad* interface.

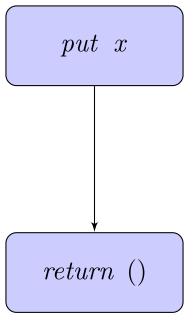
Here's a familiar kind of flowchart:



For now I want to concentrate on how we build these trees and then later talk about actually getting them to perform an action.

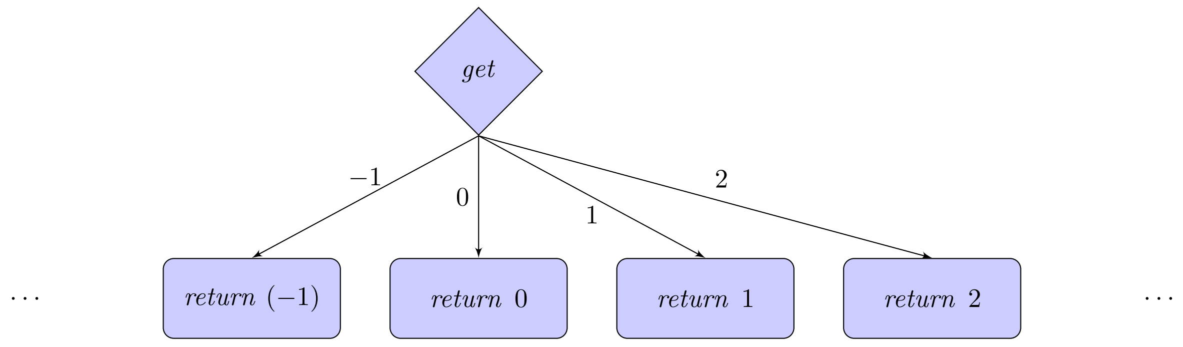
The idea now is that we have a mutable state variable of some type and nodes to set and get this state. The *put* function represents putting its argument into the state. The *get* function is used to represent getting data from the current state, selecting a branch below according to its value. We also have leaf nodes representing the final value of our computation. With the list monad, a branching tree meant executing all branches. With the *State* monad we'll just execute one branch at any branch point, the one corresponding to the value in the state at that point.

More precisely, we have a type constructor *State* that builds a flowchart tree type, *State* s a, from two types, s and a. s is the type of the state, and a is the type of the leaf nodes. We also have two functions, not part of the monad interface, that we can use to construct flowcharts. $put :: s \to State\ s\ ()$ builds a tree that looks like this:

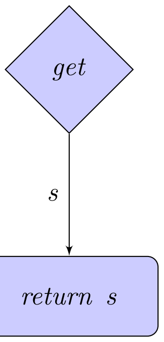


This tree represents storing the value x in our state. () is an element of the type with one element, also called (). You can ignore this value, it's just there so that we have a leaf node suitable for grafting.

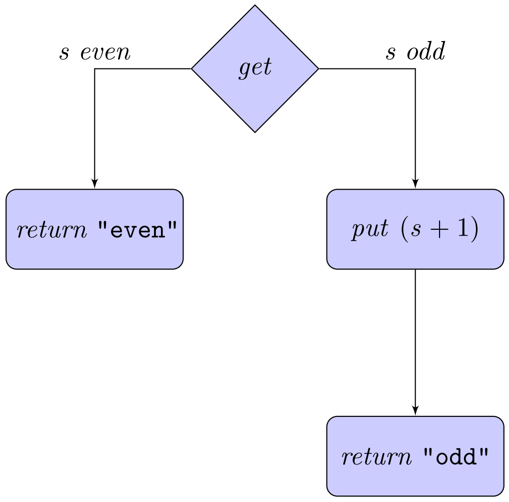
We also have *get* nodes. In the first flowchart example of this section I illustrated state of type *Bool* so we only needed a two way branch. More generally we have state of type s and to cover them all we need infinitely many branches, and for each branch we have a leaf whose value corresponds to the branch selected by the current state. So, for example, $get :: State\ s\ s$ can be thought of as looking like



The labels emerging from the *get* specify which branch is taken as a function of the state. The *get* function builds this entire thing for us, including the infinity of leaves. (Though I've also used *get* to label just the branch diamond in its own.) This could get a little unwieldy so it's easier to draw all of the branches emerging from the *get* by a scheme like:



Let's consider an example. We'll construct a block of code which has *Integer* valued state. If the state is odd then it'll add one to it and return the string `"odd"`, otherwise it just returns the string `"even"`. We can draw this as a flowchart:



Note that though we have drawn two branches we are still representing an infinite number of branches. All of the branches corresponding to even values returned by *get* are going to the left and the rest are going to the right. We'll need a selective grafting rule that grafts one thing for even values and another for odd values. We can start with

```
do
  get
```

which simply gives us a tree with leaves corresponding to the value of the state. Now we want to graft a simple leaf containing a string on the left.

```
do
  s ← get
  if even s
    then do
```

```
        return "even"
      else ...
```
On the right we need to graft twice. So we get
```
  do
    s ← get
    if even s
      then do
        return "even"
      else do
        _ ← put (s + 1)
        return "odd"
```
We don't care about the leaf value that *put* gives us so I stored the result in _. Haskell **do**-notation allows us to simplify that further by simply omitting the _ ←. So here is the final function to build our tree:
```
  ex1 = do
    s ← get
    if even s
      then do
        return "even"
      else do
        put (s + 1)
        return "odd"
```
Despite this code using an abstract interface to build trees, it looks remarkably like straightforward imperative code. In a sense it is - we're building an abstract syntax tree for an imperative mini-DSL inside Haskell. But this just builds a data structure. It doesn't yet do anything. The last step is to actually interpret the tree. The Haskell libraries provide a function $runState :: State\ s\ a \rightarrow s \rightarrow (a, s)$ that converts one of these trees to a function that takes an initial state value as argument, and returns both the final leaf value and the final state. We can test it out:
```
  go1 = runState ex1 26
  go2 = runState ex1 27
```
You might think this is all inefficient, first building a tree, and then implementing it. But the same trick as with the list monad is used. The *State* type is simply a wrapper around the final type we want, $s \rightarrow (a, s)$ and a suitable value is constructed at each graft rather than waiting for the end. Nonetheless, it's still convenient to think in terms of building trees.

8 Summary

Instances of type class *Monad* can be thought of as trees describing 'computations'. The *Monad* interface provides a way to graft subtrees into trees. There are as many types of 'computation' as there are interpreters for tree structures. In practice the interpretation is interleaved with the graft operation so that we don't have separate tree-building and interpretation phases.

9 Technical Note for the Mathematically Inclined

Describing monads as trees with grafting is slightly inaccurate. Apart from in the *Tree* case, in the examples above $\gg\!=$ doesn't just graft, it also performs a reduction operation of some sort. However, it is accurate to talk of elements of these monads as representing equivalence classes of trees. In this case $\gg\!=$ is grafting lowered to the space of trees modulo the equivalence.