Understanding Haskell Monads

Version 1.01 (2009-02-01)

Haskell is a modern purely functional programming language, which is easy to learn, has a beautiful syntax and is very productive. However, one of the greatest barriers for learning it are monads. Although they are quite simple, they can be very confusing for a beginner. As I have deep interest in extending Haskell's deployment in the real world, I'm writing this introduction for you.

1. Preamble

I have written this tutorial for Haskell newcomers, who have some basic understanding of the language and probably attempted to understand one of the key concepts of it before, namely *monads*. If you had difficulties understanding them, or you did understand them, but want some deeper insight into the motivation and background, then this tutorial is for you.

Haskell is a functional programming language. There is nothing special about this, but its design makes it easy to learn and comprehend and very effective and efficient in practice. A very special feature of Haskell is the concept of generalization. That means, instead of implementing an idea directly, you rather try to find a more general idea, which implies your idea as a special case. This has the advantage that if you find other special cases in the future, you don't need to implement them, or at least not fully from scratch.

However, the traditional programmer never had to face generalization. At most they faced abstraction, for example in the concept of object-oriented programming. They preferred to work with concrete and specialized concepts, i.e. the *tool for the job*. Unfortunately this attitude is still ubiquitous. The concept of monads is a particularly sad example, as monads are extremely useful, but Haskell newcomers often give up understanding them properly, because they are a very abstract structure, which allows implementing functionality at an incredibly general level.

Some of you may have read Brent Yorgey's Abstraction, intuition, and the "monad tutorial fallacy" [5], which explains very evidently why writing yet another interpretation of monads is useless for a newcomer. When I look around, I see the above as an additional problem: Many people try to avoid abstract concepts, despite their convenience, once you understand them. So my main intention with this tutorial is to defeat the fear of abstract concepts and to shed some light on monads as what they really are: An abstraction for a certain style of combining computations.

I hope, it is helpful to you and I would be very grateful about any constructive feedback from you.

2. Motivation

Haskell [1] is a purely functional programming language. Functions written in it are *referentially transparent*. Intuitively that means that a function called with the same arguments always gives the same result. More formally, given you have a function `f`, replacing its call by its result has no effect on the meaning of the program. So if `f 3 = foo`, then you can safely replace any occurence of `f 3` by `foo` and vice versa. *Purely functional* means that the language doesn't allow side-effects to happen in a way that destroys referential transparency. That way, the result of a function depends solely on its arguments, hence it has no side effects.

This resembles the mathematical notion of a function, which makes reasoning about the code much easier and in many cases enables the compiler to optimize much better than code, that is not referentially transparent. Furthermore the order of evaluation becomes meaningless. So for an expression `(x,y)` the compiler is free to evaluate `x` or `y` first or even skip evaluation of either one, if it isn't needed. This gives flexibility (as you can have infinite data structures or computations, as long as only a finite portion is used) and high performance. Finally, the compiler is free to take any possible path to the result. This makes Haskell programs insanely parallelizable, as a compiler can decide to take multiple paths in parallel. It can decide to calculate `x` and `y` at the same time.

The opposite of referentially transparent is *referentially opaque*. A referentially opaque function is a function that may mean different things and return different results each time, even if all arguments are the same. The canonical example of this is a random number generator. In most languages a random number function takes no arguments at all. Although it may sound counterintuitive, even a function that just prints a fixed text to the screen and always returns 0, is referentially opaque, because you cannot replace the function call with 0 without changing the meaning of the program.

As pointed out above, an obvious consequence is that in Haskell you can't write a function `random` without arguments, which returns a pseudorandom number, because it would not be referentially transparent. In fact, a function, which doesn't take any arguments, isn't even a function in Haskell. It's simply a value. A number of simple solutions to this problem exist. One is to expect a state value as an argument and produce a new state value together with a pseudorandom number:

```
random :: RandomState -> (Int, RandomState)
```

Another simple solution is to expect a seed value as an argument and produce an infinite list of pseudorandom numbers. You can write this easily in terms of the `random` function above:

```
randomList :: RandomState -> [Int]
randomList state = x : randomList newState
  where
    (x, newState) = random state
```

So the problem of deterministic sequences like the above can be solved easily, and compared to the referentially opaque `random` function, which you usually find in imperative languages, you get a useful feature: You can thread the state, so you can easily go back to an earlier state or feed two functions with the same pseudorandom sequence.

What about input/output? A general purpose language is almost useless, if you can't develop user interfaces or read files. We would like to read keyboard input or print

things to the terminal. Meet *getChar*, a hypothetical function, which reads a single character from the terminal:

```
getChar :: Char
```

You will find that this breaks referential transparency, because each reference to this function may yield a different character. We have seen that we can solve this problem by expecting a state argument. But what's our state? The state of the terminal? Well, let's get more general and just pass the state of the universe, which is of the hypothetical type *Universe*. So we adjust the type of *getChar*, and we implement a *twoChars* function to demonstrate how to use the *getChar* function:

```
getChar :: Universe -> (Char, Universe)

twoChars :: Universe -> (Char, Char, Universe)
twoChars world0 = (c1, c2, world2)
  where
    (c1, world1) = getChar world0
    (c2, world2) = getChar world1
```

We seem to have found a useful solution to our problem. Just pass the state value around. But there is a problem with this approach. Firstly, of course, that is a lot of typing for the programmer, since we need to pass the world's state around all the time. Secondly and more importantly, what is a very useful and desirable feature for functions like *random*, becomes the main obstacle for strictly impure operations like reading keyboard input or writing to the terminal:

```
strangeChars :: Universe -> (Char, Char)
strangeChars world = (c1, c2)
  where
    (c1, _) = getChar world
    (c2, _) = getChar world
```

Let's try to understand what the code above is attempting to do. We read the character *c1* from our universe, which is in state *world*. We also read the character *c2* from the universe with the same state, so we really go back in time. But when do we do that? The order in which *c1* and *c2* get computed is undefined, since we didn't sequence our world state like in the *twoChars* function. Next thing is that *strangeChars* doesn't return the new state of the universe, so after its values are demanded, the fact that it read from the terminal is forgotten, just like it never happened.

Conclusion: We can thread the state of a pseudorandom number generator without problems, but we must not thread the state of the universe! There exist a few solutions to this problem. For example, the purely functional language *Clean* uses *uniqueness types*, which is basically the above, but the language detects and thwarts attempts to thread world state, so I/O with explicit state passing like above becomes consistent. Haskell takes another approach. Instead of passing the world state explicitly, it employs a structure from category theory called a *monad*.

3. An example

As you are reading this tutorial, you have probably already found that there are many possible interpretations of monads. They are an abstract structure, and at first it can be difficult to understand where they are useful. The two main interpretations of monads are as containers and as computations. These two explain very well how existing monads are useful, but firstly in my opinion they still don't tell you how to recognize things as monads, and secondly they can look quite incompatible at times (although

they aren't).

So I'm trying to provide you with an idea of monads, that is more generic and allows you to find familiar patterns in monad usage. Particularly it should make it easy to recognize monads as such. However, I won't give you a concrete notion yet. Let's start with a motivating example instead.

Say you have a function, which may not give a result. What would be the type of that function? The exact square root over the integers is a good example, so let's write it:

```
isqrt :: Integer -> Integer
```

What is `isqrt 3`? What is `isqrt (-5)`? How do we handle the case where the computation doesn't have any result? The first idea that comes to mind originates from the imperative world. We expect the argument to be a square, otherwise we abort the program with a signal or exception. We say that if the argument is not a square, then the result is ⊥ or *bottom*.

The ⊥ value is a theoretical construct. It's the result of a function, which never returns, so you can't observe that value directly. Examples are functions, which recurse forever or which throw an exception. In both cases, there is no ordinary returning of a value.

Mathematically more correct would be an approach where invalid arguments are impossible by concept:

```
isqrt :: Square -> Integer
```

On a first glance, this seems to work, but what if we need the square root of an `Integer`? We need to convert it to a `Square` first, at which point we're facing the same problem. Also often we actually want our program to handle the case, where there is no result. So write a wrapper type `Maybe`:

```
data Maybe a = Just a | Nothing
```

A value of type `Maybe a` is either `Nothing` or `Just x`, where `x` is of type `a`. For example, a value of type `Maybe Integer` is either `Nothing` or `Just x`, where `x` is an `Integer`. So now we can change the type of our integer square root function and add a (not so optimal, but comprehensible) implementation:

```
isqrt :: Integer -> Maybe Integer
isqrt x = isqrt' x (0,0)
  where
    isqrt' x (s,r)
      | s > x     = Nothing
      | s == x    = Just r
      | otherwise = isqrt' x (s + 2*r + 1, r+1)
```

Now that `Nothing` is a valid result, our function handles all cases. Here are a few examples of `isqrt` values:

```
isqrt 4  = Just 2
isqrt 49 = Just 7
isqrt 3  = Nothing
```

What if we would like to calculate the fourth root? We now have a square root function, so wouldn't it be nice to write the fourth root function in terms of square roots? And we would like to retain the feature of returning `Nothing`, if there is no result. How could we write that function? I'm very confident you will quickly come up

with something like this:

```
i4throot :: Integer -> Maybe Integer
i4throot x = case isqrt x of
               Nothing -> Nothing
               Just y  -> isqrt y
```

Try to understand this code as well as possible. It first takes the square root of its argument. If there is no square root, then naturally there is no fourth root. If there is, then it takes the square root of the square root, which may fail again. So this is a `Maybe` computation, which has been built from two `Maybe` computations.

Do you grasp the similarity with other wrapper types like lists? You could implement the same using lists:

```
isqrt :: Integer -> [Integer]
isqrt = ...

i4throot :: Integer -> [Integer]
i4throot x = case isqrt x of
               []  -> []
               [x] -> isqrt x
```

Of course, the list type allows multiple results, too, so you could even return both square roots of the number in `isqrt`. You would need to rewrite `i4throot` to take that into account, though, as it currently supports only a single square root.

The general idea is: You have some computation with a certain type of result and a certain structure in its result (like allowing no result, or allowing arbitrarily many results), and you want to pass that computation's result to another computation. Wouldn't it be great to have a nice generic syntax for this idea of combining computations, which are built from smaller computations, and which use such a wrapper type like `Maybe` or lists? It would be even better to abstract away certain structural properties of the result, so the fact that lists allow multiple values would be taken into account implicitly. We're approaching monads.

4. Monads

You have learned the basic idea of monads at the end of the last section. A monad is a wrapper type around another type (the *inner type*), which adds a certain structure to the inner type and allows you to combine computations of the inner type in a certain way. This is really the full story.

To achieve that, a monad makes values of the inner type indirect. For example, the `Maybe` type is a monad. It adds a structure to its inner type by allowing lack of a value, and instead of having a value 3 directly, you have a *computation*, which results in 3, namely the computation `Just 3`. Further, you have a computation, which doesn't have a result at all, namely the computation `Nothing`.

Now you may want to use a computation's result to create another computation. This is one of the important features of monads. It allows you to create complex computations from simpler parts intrinsically, i.e. without requiring a notion of *running* computations.

Side note: To emphasize on how monads are nothing special or magic, I will refrain from using Haskell's nice syntactic sugar for now and implement and use them in plain.

Technically, if you find that your type is a monad, you can make it an instance of the `Monad` type class. The following is the important part of that class (there are two more

functions, but we don't need them now):

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

The `return` function, as its type suggests, takes a value of the inner type and gives a computation, which results in that value. The `(>>=)` function (usually used in the infix `>>=` notation) is more interesting: `c >>= f` is a computation, which takes the result of `c` and feeds it to `f`. You can view it as a method to incorporate an intermediate result (the result of `c`) into a larger computation (the result of `f`). How this looks like, will become clear shortly. I will call `c` the source computation and `f` the consuming function from now on.

The `(>>=)` function is usually called the *binding operator* or *bind* function, because it binds the result of a computation to a larger computation. The function `f` (being the second argument to `(>>=)` is what I will call a *monadic function* in the rest of this tutorial. It is a function, which results in a computation.

The power of the `(>>=)` comes from the fact that it leaves unspecified, in what way the result of the source computation is passed to the consuming function. In the `Maybe` monad, there may be no result at all, so the consuming function may never be actually called.

The Maybe monad

It's much easier to get started with an example. Imagine for a moment that `Maybe` doesn't exist. So let's define it:

```
data Maybe a = Just a | Nothing   deriving (Eq, Show)
```

You will quickly find that the `Maybe` type is a wrapper type around an inner type, which adds a certain structure to the inner type. The `return` function for `Maybe` is easy to implement: In the `Maybe` monad, `Just x` is a computation, which results in `x`.

More interestingly, say, you need the result of a `Maybe` computation (the source computation) in another computation, so you want to bind it to a variable `x`, then if the source computation has a result, then `x` becomes that result. If there is no result, then (because of the lack of a value for `x`), the entire computation has no result. This idea is implemented through the binding function `(>>=)`:

```
instance Monad Maybe where
  return x = Just x

  Nothing >>= f = Nothing
  Just x  >>= f = f x
```

Passing the result of the computation `Nothing` to a consuming function `f` means resulting in `Nothing` right away (since there is no result), without running `f` at all. Passing the result of `Just x` to a consuming computation `f` means passing `x` to `f` and resulting in `f`'s result.

Now let's revisit our exact integer fourth root function `i4throot`. It calculates the exact integer square root of its argument, and if there is any result, it calculates the exact integer square root of that result. Does this sound familiar? It should, because we can now exploit the monadic properties of `Maybe` (and you should definitely try this out):

```
i4throot :: Integer -> Maybe Integer
```

```
i4throot x = isqrt x >>= isqrt
```

The list monad

Similar to the `Maybe` type, you will find that the list type is a wrapper type around an inner type, which adds a list structure to that inner type, so it allows arbitrarily many results. What about binding a computation's result to a monadic function?

To answer that, just ask yourself: What is a list in the first place? It represents an arbitrary number of values, so it represents non-determinism. In other words: The computation `[2,3,7]` is one, which results in 2, 3 and 7.

Now imagine you need the result of a list computation in another computation, so you want to bind that result to the variable `x`. If there is no result, then the entire computation has no result (as there is no value for `x`), the same idea as in `Maybe`. If there is one result, `x` becomes that one result. What about two results? If there are multiple values for `x`, then there will be multiple incarnations of the consuming computation, one for each of the result values. The resulting computation collects all the individual results into one larger result.

```
[10,20,40] >>= \x -> [x+1, x+2]
```

This computation results in `[11,12,21,22,41,42]`, because you bind the result of `[10,20,40]` to `x`. But there are three results, so there will be three incarnations of `[x+1, x+2]`, one with `x == 10`, one with `x == 20` and one with `x == 40`. The result of each of the incarnations is merged together into one large end result.

Again, `return` is easy to implement: In the list monad, `[x]` is a computation, which results in `x`. Have a look at the monad instance for the list type below:

```
instance Monad [] where
  return x = [x]
  xs >>= f = concat . map f $ xs
```

Again, the `(>>=)` function is the interesting one: As its first step, it maps the consuming function `f` over all values of the source list, thereby running each of the results of `xs` through `f`. Since each time `f` results in a list in its own right, the result of this mapping is a list of lists. As its second step, it takes this result list of lists and forms one larger result list by concatenating the individual lists. The following code illustrates this again:

```
multiples :: Num a => a -> [a]
multiples x = [x, 2*x, 3*x]

testMultiples :: Num a => [a]
testMultiples = [2,7,23] >>= multiples
```

The result of `testMultiples` is `[2,4,6,7,14,21,23,46,69]`, because it takes each value of the source list, feeds it into the consuming function `multiples`, resulting in three result lists `[2,4,6]`, `[7,14,21]` and `[23,46,69]`. In other words: It maps the `multiples` function over the source list. Finally it concatenates the result lists to form a single list containing all results.

The identity monad

The *identity monad* is of little use in practice, but I'm giving it here as a basis for documenting `State` and the `IO` monad later, and of course for the sake of monad theory and completeness. Also you may find uses for the identity monad in *monad transformers*, about which I will talk later.

A computation in the identity monad simply gives a value, so it doesn't add any special structure like `Maybe` or the list monad do. Binding an identity computation's result to a consuming function simply means passing that single result verbatim.

```
data Identity a = Identity a

instance Monad Identity where
  return = Identity
  Identity x >>= f = f x
```

The computation `Identity 5` is a computation, which results in 5, so it's natural that `return x` simply gives `Identity x`, as `return x` should give a computation, which results in `x`. Also passing the result `x` of the computation `Identity x` to a consuming function `f` means simply giving that result verbatim to `f`.

5. Properties of monads

The monad laws

First of all, for something to really be a monad, it has to adhere to three laws. Besides that these are really needed by category theory, they also make sense:

1. `return x >>= f == f x`
2. `c >>= return == c`
3. `c >>= (\x -> f x >>= g) == (c >>= f) >>= g`

The first law requires `return` to be a left identity with respect to binding. This simply means that turning `x` into a computation, which results in `x` and then binding that computation's result to a consuming function `f`, is the same as passing `x` to `f` directly. Sounds obvious, doesn't it?

The second law requires `return` to be a right identity with respect to binding. That means that binding the result `x` of a source computation to the `return` function (which is supposed to give a computation, which results in `x`) is the same as the original source computation (as that one results in `x`). That should be just as obvious as the first law.

The third law requires binding to be associative. To understand it, consider a computation, which results in the 30th root of an integer or `Nothing`, if there isn't any. You can build it up by taking the square root, then the cube root and finally the fifth root of that number:

```
i30throot :: Integer -> Maybe Integer
i30throot x = isqrt x >>= icbrt >>= i5throot
```

You can take the computation of the sixth root `(isqrt x >>= icbrot)` and feed its result to `i5throot` to calculate the fifth root of it, or you can take the computation of the square root `isqrt x` and feed its result to `(\y -> icbrt y >>= i5throot)` to calculate the 15th root of it. The result is the same. This is required by the third monad law.

Support functions

As said earlier, the `Monad` class given above is not complete. There are two more functions, `(>>)` and `fail`:

```
class Monad m where
```

```
return :: a -> m a
(>>=)  :: m a -> (a -> m b) -> m b
(>>)   :: m a -> m b -> m b
fail   :: String -> m a
```

The `(>>)` function is very convenient, if the result of a computation is meaningless or not needed. You don't need to provide it in your instance definition, as it can be easily derived from `(>>=)`, and this is the default definition:

```
a >> b = a >>= const b
```

Suppose `a` and `b` are computations, and you want a computation, which runs both `a` and `b` and results in the result of `b`, ignoring the result of `a`. With the binding function, you would write something like `a >>= \_ -> b` or `a >>= const b`. With the convenience function `(>>)` you can simply write `a >> b`. Later in the section about implicit state, this will become useful.

This function is usually used with computations, which result in a value of type `()`, the unit type, which is used, when there is no meaningful result. Other than ⊥ there is only one value of type `()`, namely `()`. The unit type is analogous to the `void` type in C, although C's `void` type does not have any values, while `()` does.

`fail` is a function, which takes an error string and gives a computation, which represents failure, possibly including the given error string. The default definition gives a ⊥ computation, so calling this function aborts the program. Ignore this function for now. We will see later, where it's useful.

Interpreting monadic functions and computations

A monadic computation, which you can get using monadic functions like `return` or our `isqrt` function, or by using constructors of monads like `Just`, has a different nature than a regular value. I have stated earlier that it is correct to view monadic values as *computations* instead of values. This has the following background.

Consider `Maybe`. A monadic value `Just 3` is not the value 3 right away. It is a computation which results in 3. This may seem unnatural at first, since the result is already there, but try to interpret what the binding function `(>>=)` does. It takes a source computation and a consuming monadic function and *binds* the source computations result to the argument of the consuming function. What *binding* means is left to the monad. This gives an intrinsic idea of using the result of a computation in another computation, without requiring a notion of *running* computations.

Importantly, as long as you don't request the result, there is no result, there is just a computation. How do we request results? The most obvious method is to request results inside the corresponding monad, for example by binding. But as you just learned, binding does not yield a result. It yields a computation. Extracting a result and using it is part of that computation, but what about requesting a real end result, which is *not* itself a computation? In other words, what about *running* a computation?

Trapped in a monad

You have seen that monads are containers, which denote computations. You can create computations to result in a certain value, and you can bind the result of computations to monadic functions. But we also want results. We would like to extract the 3 from `Just 3`. So far, we have only seen monads with known constructors, like `Maybe` and lists, so you can use pattern matching or even equality to extract values from a monadic value.

Even if the constructors (`Just`, `Nothing`, `[]` and `(:)`) were unknown, you have

support functions to request results from a computation, which look different depending on the monad. For example, for the list monad, you have functions like `head` or `last`. For `Maybe` you have `fromJust` or `fromMaybe`.

But what if you don't know the constructors and there are no support functions to extract a value out of a computation? Then it is indeed not possible to do so! This is important, because it allows us to do some interesting things with monads. This property is the basis for interacting with the outside world without passing world state around. It is the basis for the `IO` monad, which I will talk about soon.

6. Implicit state

I have talked a lot about *state* in the second section. State is so integral to most other programming languages, that you likely have never bothered realizing that it even exists, since the usual imperative model of programming views the whole program as one giant state, which is continuously modified. This is certainly not surprising, since most languages have been modeled to reflect the Turing machine model of computation. Haskell is (and must be) different.

What is state in the first place? It is a modifyable environment your program or part of your program has access to throughout its life. For this to fit into the purely functional model and to retain referential transparency, state needs to be explicit, for example as a function argument. See the `random` function in the first section again to see, how this works:

```
random :: RandomState -> (Int, RandomState)
```

While in non-pure languages, a function returning a random number can get away without passing state around, for example by using global variables, this is not the case in Haskell. Of course, this is a monads tutorial, so I would not mention state, if monads could not help here. ;)

Indeed, monads do help. In fact, they are truely a virtue for everything that has to do with state. Say hello to `State` monads:

```
data State s a
```

I'm not showing you `State`'s complete definition yet, because the details can be confusing at first. For now, let's limit us to how `State` is used. Remember that you need to import `Control.Monad.State` to use it.

`State` is a type parameterized over two variables `s` and `a`. For some state type `s`, the type `State s` is a monad. A computation of type `State s a` is a computation depending on a state value of type `s` and has a result of type `a`. Throughout the computation, state changes can be applied in a purely functional way.

Looking at how the `return` and `(>>=)` functions for state monads are defined, they appear to be equivalent to the identity monad. They don't add any special structure to the inner type, so a computation in the state monad gives exactly one result. `return x` gives a computation, which results in `x`, and `c >>= f` gives a computation, which passes the one single result of `c` to `f`.

You can *run* a stateful computation using the support functions `runState`, `evalState` and `execState`:

```
runState  :: State s a -> s -> (a, s)
evalState :: State s a -> s -> a
execState :: State s a -> s -> s
```

All of them take a stateful computation and an initial state as their arguments and give (part of) the result of that computation. Although they seem to break referential transparency on a first glance, they don't, because you feed into them all information needed to produce the result, and indeed, the result is always the same, if the arguments are the same. `runState` is the basis of the other two. It produces both the result value and the end state as a tuple.

Of course, a state monad would be useless, if you couldn't actually use the state in computations, because you would have just another identity monad. So there are two monadic support functions `get` and `put` to retrieve and change the current state respectively. They make the subtle difference between state monads and the identity monad:

```
get :: State s s
put :: s -> State s ()
```

The `get` computation has a result, which is the current state (that's why the result type is the same as the state type), and the `put` function takes a new state value and replaces the current state with it. Let's try this out:

```
runState (put 3) 4 = ((), 3)
```

This example should be obvious. The computation `put 3` simply replaces the current state (which is the initial state passed to `runState` in this case) with the value 3 and returns the result value `()`. Since this computation ignores its initial state (given through the `runState` function), the result is always `((), 3)`, independent of the initial state.

```
runState (get >>= \s -> return (2*s)) 10
  = (20, 10)
```

This example is more interesting. It makes use of the initial state 10 given to it in that it returns it doubled as its result. The state itself is not changed, so the result value is 20, while the end state is 10.

```
runState (get >>= \s -> put (s+1)) 10
  = ((), 11)
```

The computation given here first takes the current state (which is the initial state in this case), binds it to `s` and puts back `s+1`. The result of that computation is `()`, but the end state is the initial state plus one.

```
runState (get >>= \s -> put (s+1) >> return (s^2)) 3
  = (9, 4)
```

Finally an example, where both the result value and the state value are significant. The state computation takes the current (in this case initial) state, binds it to `s`, then puts `s+1` back as the new state and finally returns `s^2` as the result.

Of course, such stateful computations can be further combined, so the current state doesn't have to be the initial state. Let's see how this works by writing a simple state-backed pseudo-random number generator:

```
import Data.Word

type LCGState = Word32

lcg :: LCGState -> (Integer, LCGState)
lcg s0 = (output, s1)
```

```
    where s1 = 1103515245 * s0 + 12345
          output = fromIntegral s1 * 2^16 `div` 2^32
```

This function implements a pseudo-random number generator called a *linear congruential generator*, which advances its current state *s0* to *s1* with the following formula: *s1 = (1103515245 * s0 + 12345) mod 2^32*, and then outputs the highest 16 bits of *s1* as a random number.

Using a `State` monad, we can make this passing of the `LCGState` value implicit. We write a computation `getRandom` that takes the current state value, passes it to `lcg`, then updates the state and returns a random number.

```
getRandom :: State LCGState Integer
getRandom = get >>= \s0 -> let (x,s1) = lcg s0
                           in put s1 >> return x
```

You can now use the `getRandom` computation to build larger computations. It is important that you interpret functions like `get` and `getRandom` as computations. The result of them are *not* values, but computations. You can bind the result of those computations to other computations using the `(>>=)` function. This property preserves referential transparency, because you're not dealing with results, but with the computations to get them. You can combine these computations as you like:

```
addThreeRandoms :: State LCGState Integer
addThreeRandoms = getRandom >>= \a ->
                  getRandom >>= \b ->
                  getRandom >>= \c -> return (a+b+c)
```

Replacing the `getRandom` call with its result results in the same program, because the result is not a value, but a computation. Try `runState addThreeRandoms x` with different values for `x`.

Now what is the great advantage here? Look closer at the `addThreeRandoms` function. Do you find it? The state of the generator is not mentioned anywhere. It is implicit. No explicit passing, not even to `getRandom`. Every computation of type `State s a` has implicit state of type `s`, and this state is carried along the computation through binding.

And you have seen, functions like `put` modify the state without giving any meaningful result. You could use this, for example, to implement a convenience function `modify` to modify the state (and in fact, this function is even predefined for you):

```
modify :: (s -> s) -> State s ()
modify f = get >>= \x -> put (f x)
```

7. The IO monad

Now look at `State` again, and then think back to the second section "*Motivation*" above. Do you remember the hypothetical `Universe` type, which we passed explicitly there? Couldn't we use a `State` monad to turn the state of our universe implicit? Yes, if this type existed, we could:

```
type IO = State Universe
```

This would already solve our problem, wouldn't it? Not yet, because there is a catch: Where does the initial state of the universe come from? The solution is to have a monadic `main` computation, which is the entry point of our Haskell program. Running the program means running the `main` computation with the current state of the

universe as its initial state:

```
main :: IO ()
```

In other words, the whole program is simply one *IO* computation called *main*, which may be built from smaller computations, and which, given the implicit state of the universe, can query and change this state. Now, just like the *put* function, it makes sense to have a *putStrLn* function, which is a computation altering part of this implicit state:

```
putStrLn :: String -> IO ()
```

Although, if there was really a *Universe* type, it would make sense to have a bunch of *Universe*-manipulating functions and then use a *State* monad to turn it into implicit state, there is a problem with this approach, namely threading. In the second section, we have seen that threading the state of the universe is pointless by concept. We cannot go back in time, and threading universe state *is* doing just that.

To make threading of the universe state impossible, it must be impossible to reference that state, because as long as we can give the state of the universe a name, we can use this name to refer to it arbitrarily often. To illustrate this problem, suppose there is a *Universe*-function *writeLog*, which writes a log entry:

```
writeLog :: Universe -> String -> Universe

strangeWriteLog :: String -> IO ()
strangeWriteLog msg = get >>= \world0 ->
                      put (writeLog world0 msg) >>
                      put (writeLog world0 "")
```

This is basically the same problem as in the second section. As long as we can refer to the state of the universe in some way, we can thread it. The code above takes the state of the universe using *get*, writes the log entry and puts the modified universe state back. After that, it uses a version of the universe, where the log entry has not been written yet, and writes an empty line to that. Finally it puts the modified version of the latter universe back. This code goes back in time.

Again, there seems to be a simple solution to this problem. We simply disallow using *get* and *put* with *IO*, and we disallow using *writeLog* directly. We provide a safe wrapper version instead:

```
writeLogSafe :: String -> IO ()
```

Now we have seen that, in the context of *IO* (= *State Universe*), we must artifically disallow using *get* and *put*. Further, we don't need *runState* for that monad, because the initial state comes implicitly with running the program (i.e. running the *main* computation). This takes us to the conclusion that *IO* gets along *without* a *Universe* type at all. If there is no state value to refer to, we don't need a type for it. Hence, we don't need the *State* type either. Instead, we define *IO* independently as a completely opaque type:

```
data IO a
```

This is just another state monad, but it is not written in terms of *State*, and there is no type for the state of the universe. There is also no state value, so there is no *get* or *put*, which may query or change it. Since there is no value to refer to, it's impossible to thread the state of the universe.

Taking this further, there is also no unsafe *writeLog* like the one above anymore, for

which we would need to write a safe wrapper first, but instead `writeLog` is already a safe monadic function:

```
writeLog :: String -> IO ()
```

This function takes a string and results in a computation, which takes the state of the universe and adds a log entry to it. With this approach, the threading problem is solved. So we have abstracted away all problems with impure operations in our purely functional language. For us, there is no way to *run* an `IO` computation other than by running `main`, and that can only be done through executing the program. We can only combine `IO` computations in terms of binding, and the only way to change the state of the universe is to use opaque state-changing functions like the (hypothetical) `writeLog` function. We are trapped in the `IO` monad, and this is how input/output in Haskell is consistent with referential transparency.

Properties of IO

You may find that `State` monads as well as `IO` look very similar to the identity monad. The idea of what a *value* is, is left simply as *exactly one value*, just like in the identity monad. Also, binding the result value of a computation means simply passing this one value, again just like in the identity monad. However, you can interpret both `State` monads and `IO` as implicitly and invisibly passing the current state besides the value.

`IO` is also special in that it enforces sequencing. While usually in a Haskell program, the order of evaluation is undefined, it is not so for combining `IO` computations. The `(>>=)` function always evaluates its first argument before evaluating its second. This makes input/output code much more predictable.

Further, `IO` is one of the few monads without a real internal representation. There is no *state value* at all. Think of the state value being really the state of the universe. Obviously we cannot turn the real state of the universe into some discrete value in memory, and there is also no reason to do so. The `IO` monad is rather a theoretical concept to make impure operations consistent with referential transparency, and to *think of* the state of the universe being really implicit state, just like in a `State` monad.

Predefined computations

As suggested above, computations changing the state of the universe *must* be opaque. So for a program to be useful, it must have access to a number of predefined functions for performing basic operations like accessing files, interacting with the user, networking, etc. I'm listing only the most basic operations, so you can get started. Refer to the `System.IO` module for more I/O functions.

```
putStr   :: String -> IO ()
putStrLn :: String -> IO ()
print    :: Show a => a -> IO ()
```

These three functions can be used to print a string or some arbitrary showable value to the standard output (`stdout`). `print` is mainly useful for debugging purposes. Both `putStr` and `putStrLn` write a string verbatim to `stdout`, the latter appends a line feed.

```
getChar :: IO Char
getLine :: IO String
```

These two computations read from standard input (`stdin`) one character or one line, respectively. Using these, you can already write a program, which echoes the user's input:

```
main :: IO ()
main = getLine >>= \line ->
       putStrLn line >>
       main
```

With a little extension, this program quits upon entering "quit":

```
main :: IO ()
main = getLine >>= \line ->
       if line /= "quit"
         then putStrLn line >> main
         else return ()
```

8. Syntactic sugar

The do-notation

The `(>>=)` function is associative. Its first (left) argument is a computation, of which the result is bound to the argument of the consuming function given by the second (right) argument. That function results in a computation, so the following is a very frequent and familiar pattern:

```
comp >>= \x -> ...
```

Having to write whole programs with this combinator style and lambda expressions can become quite painful and annoying. That's why Haskell supports a very beautiful and intuitive notation called the *do-notation*. The above can be written as:

```
do x <- comp
   ...
```

Since the `(>>=)` function is associative (as required by the third monad law), if we want to bind the results of two computations `comp1` and `comp2`, we usually leave the parenthesis away and just write:

```
comp1 >>= \x -> comp2 >>= \y -> ...
```

This gets increasingly ugly, but the associativity of `(>>=)` allows us to align this vertically instead, since we don't need to care about parenthesis:

```
do x <- comp1
   y <- comp2
   ...
```

Of course, more sugar exists for the convenience function `(>>)`. Instead of writing `a >> b`, you can simply align `a` and `b` vertically in do-notation. So you would write

```
c >>= \x -> f x >> g (x+1)
```

simply as:

```
do x <- c
   f x
   g (x+1)
```

Let's write the last `IO` example above using this notation:

```
main :: IO ()
main = do
  line <- getLine
  if line /= "quit"
    then putStrLn line >> main
    else return ()
```

This example should be self-explaining. Instead of using the `(>>=)` function, we have bound the result of `getLine` to a name. For another example, let's reimplement the `i4throot` function from the third section:

```
i4throot :: Integer -> Maybe Integer
i4throot x = do
  squareRoot <- isqrt x
  isqrt squareRoot
```

Please consider that the above code is an example of overusing the do-notation. The original code from the third section is much clearer. However, more interesting examples are the reimplementations of `getRandom` and `addThreeRandoms` from the section about implicit state:

```
getRandom :: State LCGState Integer
getRandom = do
  s0 <- get
  let (x,s1) = lcg s0
  put s1
  return x

addThreeRandoms :: State LCGState Integer
addThreeRandoms = do
  a <- getRandom
  b <- getRandom
  c <- getRandom
  return (a+b+c)
```

The `getRandom` function binds the current state to the name `s`. The `let` syntax without the `in` keyword is specific to the do-notation, in that it's in effect for the rest of the `do` block. Finally, two consequtive monadic computations (in this case `put s1` and `return x`) are combined using `(>>)`. This gives a notation, that looks quite semi-imperative. However, apart from looking imperative in some cases, this notation has little to do with imperative programming, so the keyword `do` may be misleading.

To make this clearer, we will have a closer look at the list monad now. As you have seen, the list monad embodies non-determinism, so binding to list computations means passing each value of the source list to a non-deterministic function. The individual results are then merged together to give the result list. Using do-notation it becomes obvious what this means:

```
do x <- [1,2,3]
   y <- [7,8,9]
   return (x + y)
```

The first line binds the list computation `[1,2,3]` to the name `x`. There is really nothing wrong with the interpretation that `x` represents all three values at the same time. It is a compact name for a non-deterministic value. Also `y` is bound to `[7,8,9]`, so it represents the values 7, 8 and 9 at the same time. So `x + y` is a non-deterministic value representing nine values altogether. The result of the above code will be:

```
[8,9,10,9,10,11,10,11,12]
```

What is important here is that the order or even strictness of evaluation of the above code is completely undefined. The compiler may choose to evaluate the `x` values first or the `y` values first or both in parallel. Although the code appears to be imperative, it's by far not.

Pattern matching in do-notation

A very useful feature of bindings in the do-notation is the fact that they can do pattern matching, just like the non-sugared lambda syntax can. Instead of writing `c >>= \(x,y) -> ...`, you can write:

```
do (x,y) <- c
   ...
```

So to do pattern matching, there is no reason to use `case ... of`. However, there is a little difference. The usual lambda-based pattern matching throws an exception, if pattern matching fails. The pattern matching in do-notation is more general: If pattern matching fails, instead of throwing an exception, the entire computation in the do-block results in `fail str`, where `str` is some implementation-dependent error string like "*Pattern matching failed*". The error string may include source code file name and line/column numbers.

```
testMaybe :: Maybe (Integer, Integer)
testMaybe = do
  [x] <- return []
  return (x-1, x+1)
```

This code fails at pattern matching in the binding. When that happens, then the part of the do-block following the binding isn't evaluated, but the entire computation results in `fail "Pattern matching failed."`. As noted, the error string is implementation-dependent.

In the `Monad` instance of `Maybe`, the `fail` function is implemented as:

```
instance Monad Maybe where
  ...
  fail = const Nothing
```

That means, if, in the `Maybe` monad, pattern matching in a do-block fails, then the entire computation is `Nothing`, notably *not* throwing an exception like `case` would. Similarly, in the list monad, if pattern matching in a do-block fails, then the entire computation is `[]`:

```
instance Monad [] where
  ...
  fail = const []
```

But this shouldn't mislead you to the conclusion that a failing pattern match in a do-block must lead to an `[]` end result. This is indeed not the case:

```
testList :: [Integer]
testList = do
  Just x <- [Just 10, Nothing, Just 20]
  [x-1, x+1]
```

This code fails at pattern matching in the binding, but only in a single incarnation of

the consuming computation `[x-1, x+1]`, namely the one, where the result is `Nothing` and hence fails to pattern-match against `Just x`. This incarnation has no result, but the other two have results. The first incarnation (the one, where `Just x` matches `Just 10`) has the results 9 and 11, the second incarnation is the computation `[]`, because pattern matching results in calling `fail`, and the third and final incarnation has the results 19 and 21. So the end result of this computation is `[9,11,19,21]`, although a pattern match failed somewhere inside of the computation.

Note: This special pattern match rule is only effective for bindings (the `<-` syntax) in a do-block. It is not effective for pattern match failures caused by `case` or `let`, even if they are parts of a do-block.

List comprehensions

List comprehensions are a very convenient shortcut to do-notation for lists. A list comprehension is a description of a list, which consists of two parts separated by a vertical bar ("|", the pipe symbol), which is written in brackets, like `[ exp | desc ]`. The right part `desc` contains bindings, `let`s and guards separated by commas. The left part `exp` is an expression using the binds and `let`s from the right part. Here is an example:

```
[ x^2 + y^2 | x <- [1..3], y <- [4..6], x+y /= 7 ]
```

The right part contains two bindings, binding the list `[1..3]` to `x` and `[4..6]` to `y`. The expression `x+y /= 7` is called a guard. Combinations of `x` and `y`, for which the guard evaluates to `False` will be filtered out. The expression `x^2 + y^2` on the left side of the vertical bar describes the value to be included in the list. This example is evaluated to:

```
[17,26,20,40,34,45]
```

Besides bindings, you can also use `let` expressions on the right side of the vertical bar:

```
[ x + sx | x <- [1..10], let sx = sqrt x, sx > 2 ]
```

To save us from having to repeat `sqrt x`, and thus to avoid common subexpressions, we used a `let` expression.

9. Too much sugar

As too much sugar isn't good for our body, it's also not too good for the body of your source code. Here is a collection of common exaggerations with syntactic sugar, which you should avoid, together with reasons *why* it's better to avoid them. After all, I decided to relay the introduction of the sugar to a late section in this tutorial for some good reason. The reason is to pull you away from imperative thinking.

Superfluous do

```
main :: IO ()
main = do putStrLn "Hello world!"
```

The *do* keyword is superfluous in this code. It's correct as `do c` is translated to simply `c`, if `c` is a computation. However, it gives the impression of having a special and isolated imperative syntax for input/output code, especially to beginners reading that code. But monads are nothing special or magic, so better write:

```
main :: IO ()
main = putStrLn "Hello world!"
```

Superfluous names

```
do line <- getLine
   putStrLn line
```

This introduces a name, where a name doesn't really add convenience. It requires you to read more code to understand the obvious meaning of what it's supposed to do, namely this:

```
getLine >>= putStrLn
```

Think of Unix pipes. Instead of redirecting the output of `ls -l` to a file and open that file for paged viewing with `less`, you would rather pipe it directly using `ls -l | less`, wouldn't you?

A particularly ugly case of the above is the following code, which is highly influenced by imperative languages, where `return` is a control construct and strictly necessary:

```
do x <- someComp
   return x
```

The above code is an extraordinarily verbose, intensely sugared variant of nothing else than `someComp >>= return`. Do you remember the second monad law? The code is equivalent to simply `someComp`, which doesn't require any sugar. So you can safely replace the above code by simply `someComp`, which is only a single unsugared word. No superfluous names and no unnecessary binding.

Incomprehensible list comprehensions

```
[ x+1 | x <- xs ]
```

Again, a piece of code, which is highly influenced by imperative programming, where you manipulate values by referring to them in formulas. Think functionally! Instead of manipulating values by a single, monolithic formula, use higher order functions:

```
map (+1) xs
```

The reason is simple. You can combine these functions arbitrarily without making your code any harder to understand. Consider the following example, which is supposed to apply `f` to all values of the list of lists `matrix`, which are not 1, and finally add them together:

```
sum [ f x | vector <- matrix, x <- vector, x /= 1 ]
```

Again, using higher order functions and function composition, the code should be much clearer:

```
sum . map f . filter (/= 1) . concat $ matrix
```

10. Backtracking monads

Sometimes the general `Monad` class is too general. Often you would like to generalize

features of particular monads like failability or backtracking. In this section, I will introduce a special monad class, which generalizes the feature of backtracking, i.e. keeping track of many results.

We have seen that the code for `i4throot` is the same for both the list monad and the `Maybe` monad. Wouldn't it be great to do the same for the `isqrt` function? Instead of a particular monad, we would like to generalize it to the following type:

```
isqrt :: Monad m => Integer -> m Integer
```

Unfortunately this is impossible. Consider our little `isqrt` implementation from above again:

```
isqrt :: Integer -> Maybe Integer
isqrt x = isqrt' x (0,0)
  where
    isqrt' x (s,r)
      | s > x     = Nothing
      | s == x    = Just r
      | otherwise = isqrt' x (s + 2*r + 1, r+1)
```

To generalize the result of this function from `Maybe Integer` to `Monad m => m Integer`, firstly we need to get rid of the `Just r`. This is easy. Just replace it by `return r`. But what about `Nothing`? What is the general idea of no result? You will find that the `Monad` class by itself supports no means to express a computation, which does not have a result. You have seen monad-specific possibilities like `Nothing` in the `Maybe` monad or `[]` in the list monad, but there is no generalization for this. There is also no generalization for many results.

The reason is simple. Not all monads support the notion of a computation, which has no or many results, so we need to introduce a special class for these *backtracking* monads. We call it `MonadPlus`:

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

This class gives us a generalized computation `mzero`, which has no result, and a way to combine the results of two computations, the `mplus` operator: Here is the instance for the `Maybe` monad:

```
instance MonadPlus Maybe where
  mzero = Nothing

  Nothing `mplus` y = y
  x       `mplus` y = x
```

The `MonadPlus` instance for `Maybe` allows us to combine two computations into one, which returns the result of the first non-`Nothing` computation:

```
Just 3  `mplus` Just 4  = Just 3
Nothing `mplus` Just 4  = Just 4
Nothing `mplus` Nothing = Nothing
```

The following is the instance for the list monad:

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

The `MonadPlus` instance for the list monad allows us to combine two computations into one, which returns all results of both computations. It concatenates the results of both computations. This allows us to generalize the `isqrt` function, and hence also the `i4throot` function from `Maybe` or the list monad to any backtracking monad (i.e. any `MonadPlus`):

```
isqrt :: MonadPlus m => Integer -> m Integer
isqrt x = isqrt' x (0,0)
  where
    isqrt' x (s,r)
      | s > x     = mzero
      | s == x    = return r `mplus` return (-r)
      | otherwise = isqrt' x (s + 2*r + 1, r+1)

i4throot :: MonadPlus m => Integer -> m Integer
i4throot x = isqrt x >>= isqrt
```

This version gives zero or one result in the `Maybe` monad and *all* results in the list monad. As can be seen above, the `isqrt` function now uses `mplus` to combine two computations, one of which returns the positive square root and the other one returns the negative. The big advantage here is that you have a single piece of code for all backtracking monads.

11. Library functions for monads

One of the greatest advantages of monads is *generalization*. Instead of implementing two things independently, look for common ideas and implement a general concept. Then make these two things special cases of the general concept. This allows you to implement a few things at a more general level, which allows you to make a third thing much more easily, if it is another special case of the concept, because you don't need to reimplement all the goodies you wrote for the general concept.

Notable examples for the success of this idea are category theory and group theory. Both define general concepts and corresponding proofs. If an object is found to fit into these concepts, the respective proofs apply automatically. That saves a lot of hard work, because you get those proofs for free.

This works for programming, too. But instead of generalizing proofs, we generalize functionality. This is a very integral concept in Haskell, which is found in very few other languages. Monads are the most important example of this. Remember the `i4throot` function? As you have seen, the codes for both the `Maybe` version and the list version were the same, just the types were different. The magic lies in the `(>>=)`. It represents the general concept of binding the result of a monadic computation to the argument of a monadic function:

```
i4throot x = isqrt x >>= isqrt
```

A rich library of support functions can be found in the `Control.Monad` module. They are general in that they work for every monad. Henk-Jan van Tuyl wrote A tour of the Haskell Monad functions [4], a comprehensive tour of the various functions with usage examples. I will document a subset of them here. Some of them are available in the `Prelude` as well, but generally you'll want to import the `Control.Monad` module. If you implement a new monad, all these support functions become available for free, because they have been implemented at a general level.

Important note: For most of these monadic support functions, I'm showing you examples from multiple monads, most notably the list monad. The results, which you get in the list monad can be quite confusing at first. If you don't grasp them right away, don't worry. For now, just read on and refer back later.

mapM and forM: mapping a monadic function over a list

```
mapM  :: Monad m => (a -> m b) -> [a] -> m [b]
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()

forM  :: Monad m => [a] -> (a -> m b) -> m [b]
forM_ :: Monad m => [a] -> (a -> m b) -> m ()
```

These functions are most useful in state monads. They allow you to apply a monadic function to a list. `forM` is the same as `mapM` with the arguments flipped. If you don't need the results, or if the results are meaningless, you can use `mapM_` or `forM_`, which are the same, but they ignore the result.

```
import System.Environment

main :: IO ()
main = getArgs >>= mapM_ putStrLn
```

The `getArgs` computation returns a list of the command line arguments given to the program. To print those arguments, each on its own line, the code above uses `mapM_`.

The type of `mapM` may give the impression that in the list monad, it is equivalent to `concatMap`. However, it isn't. Have a look at what happens in the list monad:

```
mapM (\x -> [x-1, x+1]) [10,20,30]
```

It takes a list of values, applies the monadic function and gives a computation, which results in the list of values with the function applied. This is the general idea.

Remember, we're in the list monad, which denotes non-determinism. The result of the computation is `[a,b,c]`, where `a` is the result of the computation `[10-1, 10+1]`, likewise `b` is the result of `[20-1, 20+1]` and `c` is the result of `[30-1, 30+1]`. There are two results for each of `a`, `b` and `c`, so there are in fact eight results:

```
[ [9,19,29],  [9,19,31],  [9,21,29],  [9,21,31],
  [11,19,29], [11,19,31], [11,21,29], [11,21,31] ]
```

sequence: sequencing a list of computations

```
sequence  :: Monad m => [m a] -> m [a]
sequence_ :: Monad m => [m a] -> m ()
```

The `sequence` function simply takes a list of computations and gives a computation, which results in a list of each of the results of those computations in order:

```
sequence [c0, c1, ..., cn]
= do r0 <- c0
     r1 <- c1
     ...
     rn <- cn
     return [r0, r1, ..., rn]
```

Example:

```
import System.Environment

main :: IO ()
main = sequence [getProgName, getEnv "HOME", getLine]
```

```
          >>= print
```

The above example uses `sequence` to run three computations of type `IO String`. The results are collected in the result list, which is bound to `print`. When running that program, nothing will happen at first, because the last of the three computations requests an input line from the user. After that, the resulting list of three strings is printed. The code is equivalent to:

```
import System.Environment

main :: IO ()
main = do
  results <- do
    a <- getProgName
    b <- getEnv "HOME"
    c <- getLine
    return [a,b,c]

  print results
```

Although this function is easily comprehensible for identity-like monads like `State s` or `IO`, it can give quite bizarre-looking results for monads, which add structure. Let's see, what `sequence` does in the list monad:

```
sequence [[1,2], [4,5], [6], [7,9]]
```

This gives the following result list:

```
[ [1,4,6,7], [1,4,6,9], [1,5,6,7], [1,5,6,9],
  [2,4,6,7], [2,4,6,9], [2,5,6,7], [2,5,6,9] ]
```

This may look strange at first, but reconsidering the nature of the list monad, this should make perfect sense again. Just consider that `sequence` takes a list of computations and gives a computation, which results in a list of the respective results. There are four computations, namely `[1,2]`, `[4,5]`, `[6]` and `[7,9]`.

If we name the results of those computations `a`, `b`, `c` and `d` respectively, then the result is `[a,b,c,d]`. `a` represents the values 1 and 2, as it is the result of the computation `[1,2]`. `b` represents two results as well, `c` represents a single result and `d` represents two results again. So the result list `[a,b,c,d]` represents eight results. If you pay attention to the type of `sequence`, this makes sense, because the result is returned in the list monad in this case, so there is not just a list, but a list of lists, i.e. a non-deterministic list.

The code above is equivalent to the following:

```
do a <- [1,2]
   b <- [4,5]
   c <- [6]
   d <- [7,9]
   return [a,b,c,d]
```

forever: sequencing a computation infinitely

```
forever :: Monad m => m a -> m b
```

This function takes a computation `c` and turns it into `c >> c >> ...`, so it gives a computation, which runs forever (unless there is an implicit stop in the binding

function). This function is almost only useful in state monads:

```
main :: IO ()
main = forever $ putStrLn "Hello!"
```

Although in almost all cases, the resulting computation really runs forever, there are a few cases, where `forever` gives a finite computation. Notable examples are `forever []` and `forever Nothing`. The reason is simple:

```
forever Nothing = Nothing >> forever Nothing
```

You should know from the definition of binding for `Maybe`, that if the source computation has no result, then the consuming function isn't called at all, so this computation gives `Nothing` right away. The same holds for the list version.

replicateM: sequencing a computation finitely

```
replicateM  :: Monad m => Int -> m a -> m [a]
replicateM_ :: Monad m => Int -> m a -> m ()
```

These two functions are special cases of `sequence`. They take a count `n` and a computation `c` and produce a computation, which runs `c` exactly `n` times. `replicateM` returns the results, `replicateM_` doesn't. The latter function is almost only useful in state monads.

```
main :: IO ()
main = replicateM 3 getLine >>= mapM_ putStrLn
```

The above code reads three lines and then prints them in order.

This function also makes sense in the list monad. Have a look at the following example:

```
replicateM 3 [0,1]
```

The result of this computation is `[a,b,c]`, where all of `a`, `b` and `c` are the result of the same computation `[0,1]` (because we replicate that computation three times). This gives a total of eight results (key word: non-determinism):

```
[ [0,0,0], [0,0,1], [0,1,0], [0,1,1],
  [1,0,0], [1,0,1], [1,1,0], [1,1,1] ]
```

when and unless: conditional skipping of a computation

```
when   :: Monad m => Bool -> m () -> m ()
unless :: Monad m => Bool -> m () -> m ()
```

The computation `when True c` is the same as `c`, whereas the computation `when False c` is the same as `return ()`. The `unless` function is the reverse.

```
import System.Exit

main :: IO ()
main =
  forever $ do
    line <- getLine
```

```
    when (line == "quit") exitSuccess
    putStrLn line
```

The above code repeats the following computation forever: Read a line, if that line is equal to "*quit*", then throw an `exitSuccess` exception (which simply terminates the program), then finally print the line.

liftM: applying a non-monadic function to the result

```
liftM :: Monad m => (a -> b) -> m a -> m b
```

The `liftM` function can be used to turn a computation `c` into a computation, for which a certain non-monadic function is applied to the result. So the computation `liftM f c` is the same as `c`, but the function `f` is applied to its result. One says, the function `f` is *lifted* or *promoted* to the monad.

```
main :: IO ()
main = do
  x <- liftM read getLine
  print (x+1)
```

In the above code, the computation `liftM read getLine` is the same as `getLine`, besides that the `read` function is applied to its result, so you don't have to apply `read` to `x`.

```
liftM (^2) [1,2,3]
```

For lists the `liftM` function is equivalent to `map`. So the above code results in `[1,4,9]`. This should be obvious, as the list monad adds an '*arbitrarily many results*' structure to the result, hence lifting a function `f` means applying it to each result.

```
liftM2 :: Monad m => (a1 -> a2 -> b) -> m a1 -> m a2 -> m b
```

Similarly to the `liftM` function, the `liftM2` function takes a binary function `f` and two computations. It gives a computation, which results in the result of the two individual computations passed to `f`. Likewise there are functions `liftM3`, `liftM4` and `liftM5`.

12. Monad transformers

This final section is about combining monads. The motivation is simple. You have implicit state, which you want to use while interacting with the outside world through `IO`, or you have a stateful computation and want it to be non-deterministic (including its state). You are seeking *monad transformers*.

A monad transformer for (or *to*) a particular monad is usually written with a `T` appended to its name. So the transformer version for `State` is written `StateT`. The difference between `State s` and `StateT s` is: The former is a monad right away, while the latter needs a second monad as a parameter to become a monad:

```
data State  s a
data StateT s m a
```

Where `m` is a monad, `StateT s m` is also a monad. `StateT s m` is a version of `State s`, which returns a computation in the `m` monad instead of a result. In other words, it *transforms* a `State` computation into an `m` computation. Here is an example:

```
incrReturn :: StateT Integer Maybe Integer
incrReturn = modify (+1) >> get
```

If called with the state 3, this code would increment the state to 4, but instead of resulting in 4, it results in `Just 4`, so it gives a computation in the `Maybe` monad instead of a direct result. You cannot use the usual `runState`, `evalState` or `execState` functions here. Instead there are variants of them specific to the `StateT` transformer:

```
runStateT  :: Monad m => StateT s m a -> s -> m (a, s)
evalStateT :: Monad m => StateT s m a -> s -> m a
execStateT :: Monad m => StateT s m a -> s -> m s
```

However, on a first glance, there seems to be little difference between `State s` and `StateT s m`, other than that the latter `return`s its result in the `m` monad. This would be pretty useless. The real power of monad transformers comes with the `lift` function, which allows you to encode a computation in the inner monad. Have a look at this simple example:

```
incrAndPrint :: StateT Integer IO ()
incrAndPrint = do
  modify (+1)
  x <- get
  lift (print x)
```

As you see, the `lift` function is used to encode an `IO` computation just right in the stateful computation, hence you have combined the monads `State Integer` and `IO`. This is a great feature, for example to carry application state around implicitly:

```
data AppConfig = ...

myApp :: StateT AppConfig IO ()
myApp = ...

main :: IO ()
main = do
  cfg <- getAppConfig
  evalStateT myApp cfg
```

The idea here is that you set up the application state through the `getAppConfig` computation, which may take into account command line arguments, configuration files and environment variables. Then you run the actual application computation `myApp` with this configuration as implicit state. You could go further giving the `StateT AppConfig IO` monad a convenient synonym:

```
type AppIO = StateT AppConfig IO

myApp :: AppIO ()
```

This is just a `State` transformer. Many monads can act as transformers and have a corresponding types. For example there are the `Reader` and `Writer` monads, which I don't discuss here. For them, there exist transformer variants `ReaderT` and `WriterT`. There is also the `MaybeT` transformer variant of `Maybe` (which can be found in the MaybeT package [3]). The list monad has a transformer variant, too: the `LogicT` transformer (found in the LogicT package [2]).

However, there is one notable exception: the `IO` monad. Above in this tutorial you have learned that there is no means to *run* an `IO` computation, but transforming monads needs just that. That means, even if there were an `IO` transformer, let's call it

`IOT`, you could only construct computations in the `IOT m` monad, but you could never run them. So it makes no sense to have an `IO` transformer.

As a side note, the identity monad acts as an identity with respect to monad transformation. That means, if `MT` is the transformer variant of the `M` monad, then `MT Identity` is functionally equivalent to `M`. For example, `StateT s Identity` is functionally equivalent to `State s`, in that it adds no special properties to `State s`. Some people even proposed that we define all monads in terms of their respective transformers and the identity monad.

A. Contact

You can contact me through email (es@ertes.de). If you prefer live chats, you can also reach me in the `#haskell` channel on irc.freenode.net. My nickname there is *mm_freak*. If you are interested in more from me, visit my blog, but be prepared to find informal stuff there, too. =)

B. Update history

- Version 1.00 (2008-12-26): Initial revision.
- Version 1.01 (2009-02-01): Corrected code in syntactic sugar section, which didn't compile. Thanks to Peter Hercek for reporting. Added a reference to A tour of the Haskell Monad functions [4] and to Brent Yorgey's Abstraction, intuition, and the "monad tutorial fallacy" [5]. A few minor style modifications.

References

[1] Haskell homepage – The homepage of the Haskell language.

[2] LogicT package – A package for logic programming in backtracking monads (contains a transformer variant of the list monad).

[3] MaybeT package – The transformer variant MaybeT of Maybe, unfortunately missing in the monad transformer library.

[4] A tour of the Haskell Monad functions – A comprehensive tour of the library functions for monads, together with usage examples.

[5] Abstraction, intuition, and the "monad tutorial fallacy" – Brent Yorgey's view on why writing yet another monad tutorial won't help newcomers a lot. Very interesting.

2008-12-26, Ertugrul Söylemez