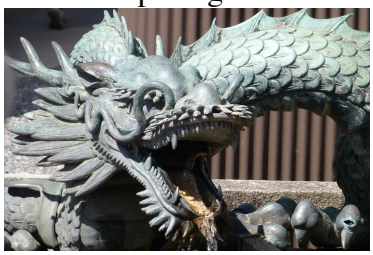
To do

6.10
7 more steps to go



Introduction to monad theory

0 comments

What is a Monad?

- A monad is an algebraic structure in category theory, and in Haskell it is used to describe computations as sequences of steps, and to handle side effects such as state and IO.
- Monads are abstract, and they have many useful concrete instances.
- Monads provide a way to structure a program.
- They can be used (along with abstract data types) to allow actions (e.g. mutable variables) to be implemented safely.

Building blocks of a monad

A monad has three building blocks:

- A type constructor that produces the type of a computation, given the type of that computation's result.
- A function that takes a value, and returns a computation that—when executed—will produce that value.
- A function that takes two computations and performs them one after the other, making the result of the first computation available to the second.

Let's restate this more precisely:

Definition of a monad

A monad consists of three objects, which must satisfy the *monad laws*. Let's look at the objects first:

- A type constructor M, such that for any type a, the type $M\,a$ is the type of a computation in the monad M that produces a result of type a.
- A function $return :: a \to M\,a$. Thus if $x :: a$, then $return\ x$ is a computation in M that, when executed, will produce a value of type a.
- A function $(>>=) :: M\,a \to (a \to M\,b) \to M\,b$.
    - The first argument is a computation that produces a value of type a.
    - The second argument is a function that requires an argument of type a and returns a computation that produces a value of type b.
    - The result is a computation that will produce a value of type b. It works by running the first computation and passing its result to the function that returns the second computation, which is then executed.

Monads and Type classes

- Monads are abstract and general and useful.
- Consequently, there are many instances of them.
- This is captured by defining a type class for monads, along with many standard instances. And you can define your own.

The *Monad* type class

```
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 `return` function takes a value and returns a monadic value, i.e. a value wrapped in the monad type constructor.
- The >>= operator (pronounced “bind”) is a crucial part of a monad. It binds a value returned from a computation to another computation.
- Sometimes the value returned by a computation is unimportant.
- The >> operator (pronounced “then”) is like >>=, but it just ignores the value returned by the computation:

```
m >> n = m >>= \_ -> n
```

- The `fail` function is used by the system to help produce error messages when there is a pattern that fails to match; normally you don’t use it directly.
- We’ll just pretend `fail` isn’t there.

The monad laws

Every monad must satisfy the following three laws. So if something looks like a monad but does not satisfy these laws, it is not a monad! The laws express properties that need to be satisfied in order to make the monadic

computations composable.

- The *right unit law*:

```
m >>= return = m
```

- The *left unit law*:

```
return x >>= f  = f x
```

- The *associativity law*:

```
(m >>= f) >>= g = m >>= (\x -> f x >>= g)
```

A monad is what the definition says it is!

- There are many metaphors or intuitions about what a monad is.
- Example: a "computation" or an "action".
- But these terms are vague—they may help you to understand, but they can also be misleading at times.
- *A monad is exactly what the definition says, no more and no less*.

The *do* notation

Writing monadic computations using the bind and then operators is still somewhat clunky. Haskell provides a very convenient syntactic sugar for monadic computations called the "do notation":

```
baz :: [a] -> Maybe a

baz xs =
  do  a <- myTail xs
      b <- myTail a
      c <- myHead b
      return c
```

Syntax rules for do

```
do { x }  -- >  x

do {x ; <xs> }  -- >  x >> do { <xs> }

do { a <- x ; <xs> }  -- >  x >>= \a -> do { <xs> }

do { let <declarations> ; xs }
  -- >
let <declarations> in do { xs }
```