Monads in Computer Science

Christina Kohl and Christina Schwaiger

Winter term 2021

1 Introduction

The concepts in functional programming are different from imperative ones. Instead of updating the running state of a program one has trees of expressions constructed by applying and composing functions. These trees are evaluated by mapping values to other values and proceeding until the whole tree is evaluated. We can further differentiate between pure languages, such as Haskell, and impure languages (like Scala or OCaml) in functional programming. Pure languages treat functions as deterministic mathematical functions - meaning a function will always return the same result when given the same input. This means that a function is not allowed to interact with the outside world in any other way than by returning a value when being called. Any other observable effect caused by a function is called a *side-effect* and makes the function impure. Examples of side-effects are modifying a global state or input and output operations. Pure programming languages convince through their easy testability, formal verification and lazy evaluation [Wad93; con21].

Of course, it may be useful to integrate the advantages of impurity in pure languages. Consider as a simple example the problem of counting how often a function has been called. In imperative programming we could have a global counter variable that can be incremented by the function itself every time it is called. That is not possible in pure programming. What we can do is pass the counter as an additional argument to the function and have the function return the updated counter in addition to its original return values. We can also deal with other desirable side-effects in a similar way, i.e., by extending functions to more arguments and return values. However this is tedious and often results in convoluted code. Solving these kinds of problems is one of the main reasons for the introduction of monads into functional programming.

Monads are a theoretical concept that can be used to simulate such impure behaviour in pure programming languages such as Haskell. With monads purity of the language is never violated but the programmer is able to deal with side-effects in a very similar way as in imperative programming. In this way monads can be applied to several problems while still being a very basic and simple concept. They are the essence of many different constructions and often we do not even realize that we are using this concept. Milewski in [Mil18,

p. 290] compares monads to duct tape in real life - it can have many different applications but in each cases it simply glues some things together. Monads in our case can be seen as a way to glue code together. Below are a few examples which can all be solved with monads [Mog91].

- exceptions
- interactive input and output
- partiality
- non-determinism
- side-effects
- continuations

The remainder of this report is structured as follows: We first give a brief historical overview of the introduction of monads into functional programming. Then in Section 3 we demonstrate the usefulness of monads using a simple example of functions with side-effects. In Section 4 the *Monad* class of Haskell will be discussed in more detail and using an extended example we show how custom types can be made instances of this class. Finally in Section 5 we give an overview of the categorical interpretation of monads and how it is connected to the programming version.

2 Some History

The term *monad* was first introduced by Saunders Mac Lane whereas its concept has already been part of Roger Godement's work. Heinrich Kleisli then formulated monads by using a *bind* function, as described in the next section below [Kle65]. At this point monads were not yet put into relation with functional programming. In the 1970s and 1980s category theory and monads started to become an inspiration for computer science. For example an effective I/O based on monadic types was designed in the programming language *Opal* [Wik22]. The first one to realize the connection between monads and functional programming was Moggi [Mog89; Mog91]. Nowadays, monads are important parts of functional programming languages such as Haskell, Scheme, Perl, Python and many more [Wik22]. One especially noteworthy development in Haskell that is due to monads is the evolution of its I/O. The concept of monads resolved the initially problematic lazy stream that was used up through v1.2 to combine I/O and lazy evaluation in Haskell [PW93].

3 Monads in Computer Science

In order to define monads in computer science we will first look at a simple example of functions where certain side-effects are desired and show how one can deal with such behaviour in a pure language like Haskell. The example has been adapted from [INF].

Consider two functions `f` and `g` with type signature `Float -> Float`. Now suppose we want to add a debugging opportunity. More specifically the function calls should also return a string which describes an (un-)expected behaviour within a function call. So the type of these functions should actually be `Float -> (Float, String)`. But this leads us to the problem that two functions `f` and `g` with such a signature cannot be directly composed anymore because the input type `Float` does not match the output type `(Float, String)`. In addition it would be desirable to concatenate the two debug strings of `f` and `g` into a single string when composing the function calls. For a concrete example imagine we want to call `f` after `g` on some input `x`. The result should be the tuple consisting of `f` applied to the `Float` returned by `g x` on the one hand, and the debugging string of `f` concatenated with the debugging string of `g x` on the other hand. In Haskell this can be computed as follows:

```
f >=> g = \x -> let (gx, gs) = g x
                    (fgx, fs) = f gx in (fgx, gs++fs)
```

Such a composition is typically denoted by the infix operator `>=>` (*fish operator*) and is called *Kleisli composition*. We will provide more background on Kleisli categories in Section 5.

The application of `f` after `g` can also be achieved with a *bind* function denoted by the infix operator `>>=`. It is called bind because it binds the `Float` value of the `(Float, String)` pair obtained by `g x` to the argument of `f`.

```
(>>=) :: (Float, String) -> (Float -> (Float, String))
        -> (Float, String)
(gx, gs) >>= f = let (fx, fs) = f gx in (fx, gs++fs)
```

With that we have the following equality:

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

The standard implementation of monads in Haskell is based on the `>>=` operator instead of `>=>`. One benefit is that it allows us to write composition of several functions as a chain `g x >>= f >>=` . . . , where we can easily append more function applications on the right.

Considering the connection to category theory we should ask the question whether there exists some kind of identity function such that

```
f >=> id = f = id >=> f
```

for all functions `f` of appropriate type signature. For historical reasons the name `return` has been established for such a function in Haskell. In our example `return` can be defined by

```
return :: Float -> (Float, String)
return x = (x, "")
```

It takes a number and transforms it into the pair consisting of the same number and the empty string. Composing any function `f` with it will leave the output of `f` unchanged and append or prepend the empty string, so the result will be the same as `f` itself.

3.1 Defintion

By considering the example above we will define a monad in programming manners.

Definition (Monad)**.** *A monad is defined as the triple* `(m, return, >>=)` *where* `m` *is a monadic constructor*[1] *denoting some side-effect or impure behaviour,* `return` *represents the identity function and* `>>=` *is used for monadic composition. They have type signatures*

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

and need to satisfy the following three laws:

```
return x >>= f     =   f x                     -- left identity
m >>= return       =   m                       -- right identity
(m >>= f) >>= g    =   m >>= (\x -> f x >>= g) -- associativity
```

Left identity states that first turning a plain value `x` into a monadic value using the `unit` function and then feeding the result to some monadic function `f`, is the same as applying `f` to `x` directly. Right identity states that feeding a monadic value `m` to the `unit` function results in `m` again. Finally associativity ensures that the order in which several nested binds are computed does not change the result. On the left of the equation the monadic value `m` is first passed to the function `f` and the resulting value is then passed to `g`. On the right the monadic value `m` is directly passed to the composition of `f` and `g` which can be expressed as the function `(\x -> f x >>= g)` as we already saw in the example above.

3.2 Alternative Definitions

In the example above we had introduced the composition operator `>=>` in addition to `>>=` and showed that they can be used interchangeably. This holds in general. So monads can just as well be defined as a triple `(m, return, >=>)` where `>>=` has been replaced by `>=>`:

```
>=> :: (a -> m b) -> (b -> m c) -> (a -> m c)
```

Then the three monad laws can be written down even more succinctly [Mil18, pp. 292–295].

```
return >=> f      =   f                 -- left identity
f >=> return      =   f                 -- right identity
(f >=> g) >=> h   =   f >=> (g >=> h)   -- associativity
```

Last but not least there is still a third option for defining monads. As we will see in Section 5 a monad in category theory is a functor at its core. So we can assume that a `map` function

[1]Think of a type constructor as a kind of container or context.

```
map :: (a -> b) -> (m a -> m b)
```

exists, which transforms a function `f :: a -> b` into a function over the monadic types. Then instead of `>>=` or `>=>` we can use `map` together with the so-called `join` function

```
join :: m (m a) -> m a
```

to define monads, since `>>=` can be expressed using the following equality [Mil18, p. 295].

```
ma >>= f = join (map f ma)
```

4 Haskell

In this section we will give more details on the actual implementation of monads in Haskell and give a few more programming examples.

In Haskell the functionality of monads is available via the `Monad` type class. Type classes in Haskell can be seen as interfaces that define certain behaviours of the types that instantiate them. The definitions of many type classes in the standard Haskell libraries have been inspired by category theory. `Monad` and also `Functor` are examples of such type classes. As of GHC 8.8, `Monad` is defined as [Wikb]:

```
class Applicative m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  m >> n = m >>= \_ -> n
```

Here the operator `(>>)` is a specialized version of `(>>=)` and will not be discussed in this report. Since 2014 `Monad` is an instance of `Applicative` which is in turn an instance of `Functor`. This means that all monads are applicatives, all applicatives are functors, and therefore all monads are also functors [Wika]. An overview of the standard Haskell types that are instances of `Monad` is given in Table 1.

Monad	Imperative behaviour	Description
`Maybe`	Exception (anonymous)	Used for computations that may not return a result, i.e., partial functions.
`Either`	Exception with error description	Can be used for partial functions where the location and cause of the failure should be recorded, e.g. as a string.

`IO`	Input/Output	Used for all computations that perform I/O actions. See Subsection 4.3.
`[]` (`List`)	Non-determinism	Used for non-deterministic computations. See Subsection 4.2.
`Reader`	Environment	See [Mil18, pp. 307–308].
`Writer`	Logger/Debugger	See [Mil18, pp. 308–309].
`Cont`	Continuations	See [Mil18, pp. 311–313].
`State`[2]	Global state	For computations that maintain some kind of state. Here functions take the state as an additional argument and produce state-value pairs as a result.

Table 1: Monads in Haskell [Wik21]

4.1 The do-Notation

Haskell provides the `do` notation as syntactic sugar for dealing with monadic values and composition of monadic functions. When using the `do` notation Haskell code almost looks like imperative code, but under the hood we still have only pure functions and monads.

`do` notation	Translation
`do action`	`action`

```
do x <- y
    more code
```
	```
	y >>= (\x -> do
	    more code)
	```

```
do
    let x = y
    more code
```
	```
	(\x -> do
	        more code) y
	```

Table 2: The `do`-notation in Haskell [INF]

4.2 Example 1: The List Monad

A non-deterministic function that can return several different results (or none at all) is semantically equivalent to a function returning a list of results. Laziness even allows to have infinite lists of results. To be able to compose multi-valued functions the list constructor `[]` has been turned into a monad in Haskell[3].

[2]State as a standalone monad does no longer exist in Haskell. It is now defined in terms of `StateT`, the transformer version of state. See `https://hackage.haskell.org/package/transformers-0.6.0.2/docs/Control-Monad-Trans-State-Lazy.html` for details.

[3]The actual implementation in `GHC.Base` does not explicitly implement `return` since `pure` of `Applicative` is provided. In addition it uses list comprehensions for the implementation

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
```

Here `return` creates a singleton list. The bind operator `>>=` has been implemented in terms of the `join` and `map` version we have seen in Subsection 3.2. Here `concat` corresponds to the `join` function and flattens a list of lists into a single list.

Below is a very simple example of how several lists, or functions that produce lists, can be combined [Lip11].

```
[1,2] >>= \n -> ['a','b'] >>= \ch -> return (n, ch)
```

which outputs

```
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

When we write the above code in `do`-notation

```
do  n <- [1,2]
    ch <- ['a','b']
    return (n, ch)
```

it becomes a bit more obvious that `n` takes on every value from `[1,2]` and `ch` takes on every value from `['a','b']`.

Here is another nice example of how Pythagorean triples can be computed.

```
pythagoreanTriples :: Integer -> [(Integer, Integer, Integer)]
pythagoreanTriples n = do x <- [1 .. n]
                          y <- [x+1 .. n]
                          z <- [y+1 .. n]
                          if x^2 + y^2 == z^2 then return (x,y,z)
                            else []
```

Here `x` takes on all values between `1` and `n`. Since there can be no Pythagorean triple with `x = y` we let `y` take on values from `x+1` to `n`. Finally we know that the hypotenuse `z` must be greater than both `x` and `y` so we let `z` take values from `y+1` to `n`. The condition $x^2+y^2 = z^2$ is checked via the `if .. then .. else`. If the condition is fulfilled then the triple `(x,y,z)` is turned into a list `[(x,y,z)]` via the `return` and concatenated together with all other valid triples via the implicit `>>=`.

4.3 Example 2: IO

Looking at the example of interactive input and output makes the challenges of pure functional programming very apparent. How can a function depend on non-deterministic user input and still return the same result for identical arguments?

of `>>=`. In this report we present a semantically equivalent version that fits better into the framework presented so far.

Imagine there was a function `getChar :: Char`. If this function produced the same character every time it was called, it would be quite useless for interactive input. In addition, when asking for multiple input characters, the compiler needs to know in which order they should be processed. That is where monads start to shine. Instead of simply returning `Char` the function `getChar` has type `IO Char` where `IO` is a monad. Conceptually one can think of the `IO` monad as a container that contains the superposition of all possible characters. Inside the monad we can do computations with concrete values but from the outside we see only the container. Since `main` in Haskell has the signature `main :: IO ()` it is possible to write programs like the one below [Mil18, pp. 313–315].

```
main = do putStr "Enter your year of birth: "
          input1 <- getLine
          let y = (read input1 :: Int)
          let age = 2022 - y
          putStr "your age is "
          print age
```

Here, after the user input is read, it is parsed to an `Int` so that we can do the subtraction. Of course the parsing might fail, in which case an exception will be printed.

4.4 An Example of Instantiating Monad

So far we have seen several examples of predefined monads in Haskell. In this section we will define a new type, make it an instance of `Monad`, and prove the three monad laws for it. In this example we view a list as a way to represent non-deterministic values where each value in the list also gets assigned a probability of its occurrence. So each element in the list is actually a value-probability-pair and we assume that the probabilities sum up to 1. This example has been adapted from [Lip11].

First we define a new data type representing a list of values of some arbitrary type `a` together with their probabilities.

```
data ProbList a = ProbList [(a, Rational)]
```

Here the data type `Rational` of `Data.Ratio` is used to represent probabilities as rational numbers, like `1 % 4`, without losing precision (as would be the case if we used `Float` or `Double`). Now some uniformly distributed values like `'a'`,`'b'`,`'c'` can be represented by the following instance of `ProbList`:

```
ProbList [('a',1 % 3), ('b',1 % 3), ('c',1 % 3)]
```

Implementing `return` is easy: we can turn any value `x` into a valid probability list by creating the singleton list with value `x` and probability `1`.

```
return :: a -> ProbList a
return x = ProbList [(x, 1)]
```

The plan is to define `>>=` in terms of `map` and `join`. To avoid confusion with the predefined function `map` defined on normal lists we will call the mapping function `pmap`90=. This function should take some function `f` and a probability list as input and apply `f` to each value in the probability list, while leaving the probabilities themselves untouched.

```
pmap :: (a -> b) -> ProbList a -> ProbList b
pmap f (ProbList xs) = ProbList $ map (\(x,p) -> (f x,p)) xs
```

For the implementation of `join` first recall its type:

```
join :: ProbList (ProbList a) -> ProbList a
```

It is supposed to transform a probability list of probability lists into a single probability list while preserving the property that the probabilities sum up to 1 (provided the initial lists also satisfied this property). The result of computing `fmap f xs` is such a nested list. For example

```
xs = ProbList [(1, 1%3),(2, 1%3),(3, 1%3)]
f x = ProbList [(x, 1%2), (x+10,1%2)]
fmap f xs = ProbList [(ProbList [(1,1 % 2),(11,1 % 2)],1 % 3),
                      (ProbList [(2,1 % 2),(12,1 % 2)],1 % 3),
                      (ProbList [(3,1 % 2),(13,1 % 2)],1 % 3)]
```

Transforming this into a single probability list can be achieved by multiplying each probability value of the outer list with the probability values of all elements in its inner list. In our example

```
flatten (fmap f xs) = ProbList [(1,1 % 4),(2,1 % 4),(2,1 % 8),
                                (3,1 % 8),(3,1 % 8),(4,1 % 8)]
```

So the implementation of `join` can be obtained as follows.

```
join (ProbList xs) = ProbList $ concat $ map multAll xs
  where multAll (ProbList innerxs, p) =
                 map (\(x, r) -> (x, p*r)) innerxs
```

Finally `Monad` can be instantiated[4]

```
instance Monad ProbList where
  xs >>= f = flatten (fmap f xs)
  return x = ProbList [(x, 1)]
```

Lemma. *`ProbList` satisfies the monad laws.*

Proof.

[4]For this code to actually compile we also need to instantiate `Functor` and `Applicative`. `Functor` requires only a mapping function which we already implemented as `pmap`. `Applicative` requires implementations of `pure` and `<*>` (called *apply*) which can be derived from `Monad`.

```
-- Left identity:
return x >>= f = ProbList [(x,1)] >>= f
               = flatten (fmap f (ProbList [(x,1)]))
               = flatten (ProbList [(f x,1)]))
               = f x
-- Right  identity
ProbList xs >>= return
    = flatten (fmap return (ProbList xs))
    -- unfold the definitions of flatten, fmap, and return:
    = ProbList $ concat $
        map multAll (map (\(x,p) -> (ProbList [(x,1)],p)) xs))
    = ProbList $ concat $ [xs]
    = ProbList xs
-- Associativity
(m >>= f) >>= g = (flatten (fmap f m)) >>= g
                = flatten (fmap g (flatten (fmap f m)))
                = flatten (fmap (\x -> flatten (fmap g (f x))) m)
                = m >>= (\x -> flatten (fmap g (f x)))
                = m >>= (\x -> f x >>= g)
```
□

5 Monads in category theory

In the following we will write T instead of m for the endofunctor of the monad. The characterization of monads in category theory differs a bit to the one used in programming.

Definition. *For some category $\mathcal{C}$ a monad on $\mathcal{C}$ is defined by the following properties:*

- *a functor $T : \mathcal{C} \to \mathcal{C}$*
- *a natural transformation $\eta : \mathrm{id}_{\mathcal{C}} \Longrightarrow T$*
- *a natural transformation $\mu : TT \Longrightarrow T$*

such that the following diagrams commute

$$\begin{tikzcd} T \arrow[r, Rightarrow, "\eta T"] \arrow[dr, Rightarrow, "id"] & TT \arrow[d, Rightarrow, "\mu"] \\ & T \end{tikzcd} \qquad \begin{tikzcd} T \arrow[r, Rightarrow, "T\eta"] \arrow[dr, Rightarrow, "id"] & TT \arrow[d, Rightarrow, "\mu"] \\ & T \end{tikzcd} \qquad \begin{tikzcd} TTT \arrow[r, Rightarrow, "T\mu"] \arrow[d, Rightarrow, "\mu T"] & TT \arrow[d, Rightarrow, "\mu"] \\ TT \arrow[r, Rightarrow, "\mu"] & T \end{tikzcd}$$

where η is referred to as unit and μ as composition. TT denotes the composition of the functor T with T.

One can also interpret a monad as monoid in the monoidal category $[\mathcal{C}, \mathcal{C}]$ with identity $\mathrm{id}_{\mathcal{C}}$ and the composition of functors [Uus21].

A monad (T, η, μ) on a category $\mathcal{C}$ induces a category called the Kleisli category of T

- The objects are the objects of $\mathcal{C}$.
- The morphism from A to B of $\mathcal{C}$ are the Kleisli morphisms defined as maps $k : A \to TB$.
- The identities are given by η from the monad for each object componentwise $\mathrm{id}_A = A \xrightarrow{\eta_A} TX$.
- The composition is given by the so called Kleisli composition which is defined for two Kleisli morphism $k : A \to TB$ and $h : B \to TC$ as

$$A \xrightarrow{k} TB \xrightarrow{Th} TTC \xrightarrow{\mu} TC$$

An alternative to the definition of monads inspired by the concept of Kleisli categories is those of the so called Kleisli triples.

Definition (Kleisli Triple)**.** *A Kleisli triple $(T, \eta, \_^*)$ is given by*

- $T : \mathit{Obj}(\mathcal{C}) \to \mathit{Obj}(\mathrm{C})$
- $\eta_A : A \to TA$ *for* $A \in \mathit{Obj}(\mathcal{C})$
- $f^* : TA \to TB$ *for* $f : A \to TB$

such that the following holds

- $\eta_A^* = id_{TA}$
- $f^* \circ \eta_A = f$ *for* $f : A \to TB$
- $g^* \circ f^* = (g^* \circ f)^*$ *for* $f : A \to TB$ *and* $g : B \to TC$.

These three conditions ensure the naturality of η and $\_^*$ as well as the functoriality of T, meaning for $f : A \to B$ in $\mathcal{C}$ if $Tf : TA \to TB$ is given by $Tf = (\eta_b \circ f)^*$, then

$$T\mathrm{id}_A = (\eta_A \circ \mathrm{id}_A)^* = \eta_A^* = \mathrm{id}_{TA}$$

and for $f : A \to B$ in $\mathcal{C}$, $g : B \to C$ in $\mathcal{C}$

$$Tg \circ Tf = (\eta_C \circ g)^* \circ (\eta_B \circ f)^* = ((\eta_C \circ g)^* \circ \eta_B \circ f)^* = (\eta_C \circ g \circ f)^* = T(g \circ f).$$

Therefore we have less conditions to fulfill than in the monad case. By taking a closer look, we can obtain a one-to-one correspondence between monads and Kleisli triples for the same $T : |\mathcal{C}| \to |\mathcal{C}|$ and η [Mog91; Uus21].

We can also define the relationship between the monads in computer science introduced in Section 3 and the monads in category theory. If we formalize the programming language i.e. Haskell as a syntactic category by

- The objects are the types of the language.
- The morphisms $X \to Y$ are the functions that take a value of type X and return a value of type Y.

If we consider this relation between programming languages and categories, monads in computer science equal those in category theory as being endofunctors $T : \mathcal{C} \to \mathcal{C}$ on this syntactic category [nLa].

References

[Kle65] Heinrich Kleisli. "Every standard construction is induced by a pair of adjoint functors". In: 1965.

[Mog89] E. Moggi. "Computational Lambda-Calculus and Monads". In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. Pacific Grove, California, USA: IEEE Press, 1989, pp. 14–23. ISBN: 0818619546.

[Mog91] Eugenio Moggi. "Notions of computation and monads". In: *Information and Computation* 93.1 (1991). Selections from 1989 IEEE Symposium on Logic in Computer Science, pp. 55–92. ISSN: 0890-5401. DOI: `https://doi.org/10.1016/0890-5401(91)90052-4`. URL: `https://www.sciencedirect.com/science/article/pii/0890540191900524`.

[PW93] Simon L. Peyton Jones and Philip Wadler. "Imperative Functional Programming". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '93. Charleston, South Carolina, USA: Association for Computing Machinery, 1993, pp. 71–84. ISBN: 0897915607. DOI: `10.1145/158511.158524`. URL: `https://doi.org/10.1145/158511.158524`.

[Wad93] Philip Wadler. "Monads for functional programming". In: *Broy M. (eds) Program Design Calculi. NATO ASI Series (Series F: Computer and Systems Sciences), vol 118* 118 (1993). DOI: `\url{https://doi.org/10.1007/978-3-662-02880-3_8}`.

[Lip11] Miran Lipovaca. *Learn You a Haskell for Great Good! A Beginner's Guide*. 1st. USA: No Starch Press, 2011. ISBN: 1593272839.

[Mil18] B. Milewski. *Category Theory for Programmers*. Blurb, Incorporated, 2018. ISBN: 9780464825081. URL: `https://books.google.at/books?id=ZaP-swEACAAJ`.

[con21] Wikipedia contributors. *Functional programming — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-December-2021]. 2021. URL: `https://en.wikipedia.org/w/index.php?title=Functional_programming&oldid=1059406385`.

[Uus21] Tarmo Uustalu. *Monads and Interaction - Lecture 1.* [Online; accessed 27-December-2021]. 2021. URL: https://cs.ioc.ee/~tarmo/mgs21/mgs1.pdf.

[Wik21] Wikibooks. *Haskell/Understanding monads — Wikibooks, The Free Textbook Project.* [Online; accessed 27-December-2021]. 2021. URL: https://en.wikibooks.org/w/index.php?title=Haskell/Understanding_monads&oldid=3833254.

[Wik22] Wikipedia contributors. *Monad (functional programming) — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Monad_(functional_programming)&oldid=1063669207. [Online; accessed 5-January-2022]. 2022.

[INF] A NEIGHBORHOOD OF INFINITY. *You Could Have Invented Monads! (And Maybe You Already Have.)* [Online; accessed 10-December-2021]. URL: http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html.

[nLa] nLab authors. *monad (in computer science).* [Online; accessed 27-December-2021]. URL: https://ncatlab.org/nlab/show/monad%20(in%20computer%20science).

[Wika] Wiki. *Functor-Applicative-Monad Proposal.* [Online; accessed 19-December-2021]. URL: https://wiki.haskell.org/Functor-Applicative-Monad_Proposal.

[Wikb] Wiki. *Typeclassopedia.* [Online; accessed 18-December-2021]. URL: https://wiki.haskell.org/Typeclassopedia.