In search of a Monad for system call abstractions

Taesoo Kim
(Instructor: David Spivak)
MIT CSAIL

Abstract

For programmers, Monads are a well-known way to represent an abstract data structure, but without knowing its true nature. Admittedly, the definition of Monads in Category Theory is too subtle and obscure so makes people to avoid from trying to understand them. In this paper, we attempt to explore popular Monads in a purely functional programming language, Haskell, with a Category-theoretic mindset. Starting from the definition of Monads, we describe how they are defined and implemented in Haskell. In particular, we classify Monads by their implementation style, and it turns out that a common fallacy of using Monad metaphors can be understood in this vein. Also, we explain why IO Monad is particularly important in Haskell and what its role is, in a programmer's perspective. Furthermore, we design a new Monad, called S Monad, in order to represent the system call abstractions, and shed light on a future Haskell OS using S Monad to represent the system call abstractions.

1 Introduction

Monads are too subtle to grasp their true meaning. To explain Monads, people proposed various metaphors like composable computations [3, 6], boxes [2], convey belts [11], or even burrito [19] and lemonade [14]. Those metaphors seem to help in understanding one aspect of Monads, especially for beginners. Alas, when they start using Monads in depth, what they understood via metaphors contradicts to each other, and even doesn't make sense anymore. In this perspective, some people consider using metaphors in teaching Monads as a bad practice [17], or even as a common fallacy [19].

After learning the basics of Category Theory, I want to write down own interpretation of Monads for working programmers, so they can understand the heart of Monads. In this paper, we won't use metaphors to explain Monads; that makes people construct an abstract concept from example metaphors. Instead, this paper starts from an abstract knowledge and applies it to the concrete use cases. The main reason is as follows. Suppose you are learning about ducks. You might first want to find some species of ducks 1. After researching what they eat, where they live, and how they look like, you might have an idea of how they differ from each other and be able to explain what duck is. Suddenly, you might wonder about swans and geese, whether they are duck or not? or what are the differences from birds? In fact, duck is nothing but a name of animals who walk and quark like a duck!

(a) Wild Duck (b) Mandarin Duck (c) Pacific Black Duck

Figure 1: If an object walks like a duck and quacks like a duck, we say it is a duck. Likewise, if a data type follows Monad rules, the data type is an instance of the Monad type class.

The Monad is a term of abstractions that satisfy certain, but general, rules. Since rules are so general, Monads in fact are not very interesting until you describe a specific instance of Monads, say Maybe or State Monad for example. As you can distinguish ducks from tigers (tigers do not walk like a duck nor quack like a duck, for sure), you can differentiate Monads from Set (yes, apples and oranges!). Also, it doesn't help in understanding Monads in details. In this paper, we rather admit the Monad as an abstract structure first and then dive into the specific instances of them. Then, what are the rules and constituents for Monads? They at least give you a sense of "not weird", "non-sense" and so "natural", as a user. However, it is always fun to see how each Monad satisfies those rules and what are the constituents, as we will discuss in §2.

Outlilne This paper consists of three parts. First, we provide a way of understanding Monads, especially for system programmers, so hope to bridge some gaps between Category Theory and a practice in Haskell (§2). Second, we articulate the necessity of IO Monad in Haskell (§3). Last, but most importantly, we propose a new Monad, called S Monad, to solve specific (yet practical) problems in the operating system, representing the system call abstractions (§4).

2 Monads

For programmers, a Monad is a consistent way of giving a semantic representation to an abstraction. We say "consistent" because a Monad consists of certain constituents and follow specific rules, "semantic" because each Monad has own goal in handing situations such as uncertainty and side-effects, and "abstraction" because Monads (more precisely, the Kleisli category of a Monad) abstract the idea of composition from users. In other words, Monads generalize the idea by encapsulating the internal detail; an interface as a simple term. Let's first see its definition in Category Theory [18] and compare it with the realized code in Haskell.

```haskell
-- Monad type class: 'm' is a specific instance of Monads, like Maybe and State
class Monad m where
  return :: a -> m a                -- unit: construct a Monad
  (>>=)  :: m a -> (a -> m b) -> m b -- bind: bind a Monad to a function

  -- useful functions (not strictly follow the definition of Monads in CT)
  (>>)   :: m a -> m b -> m b       -- weaving two instances of Monads
  fail   :: String -> m a           -- plan for failures of Monads
```

Figure 2: Definition of the Monad type constructor in Haskell. In addition to `unit` and `bind`, the Monad class provides two more functions, a `fail` to escape out of the composed computations, and » to enforce orders. In particular, » can be implemented as `ma » mb = ma >>= λ → mb` in any Monad. Note that each Monad, `m`, should first be an instance of the `Functor` class in Haskell.

2.1 Definition

Definition 1. *A* **Monad** *on* **Set** *consists of three constituents (A. fuctor, B. unit map, C. multiplication map) conforming to two laws (1. unit laws, 2. associativity laws) as follows:*

A. *a functor* $T : Set \to Set$
B. *a natural transformation* $\eta : id_{Set} \to T$
C. *a natural transformation* $\mu : T \circ T \to T$

(See. two commutative diagrams for unit laws and associativity laws in the book [18])

A functor First of all, Haskell is a category, called Hask in the literature [5]: objects are types (small sets per se) and homomorphisms are functions in Haskell. Interestingly enough, a polymorphic data type (kinds of $* \to *$ in Haskell) can be considered as a functor [1] For example, the Maybe and State are functors (in fact, they are instances of the Functor class in Haskell), so we can use both for the first constituent of a Monad, trivially.

Two natural transformations Second, given a functor, we still need to provide two more constituents to make it as a Monad. Here is the place where two functions, `return` (unit) and »= (bind) in the Figure 2 play a role. In the Category-theoretic definition, there is no bind function, rather it has a multiplication map, called `join` in Haskell. Although they are equivalent [13], we found a few reasons why Haskell prefers to use `bind` instead; free variables in Haskell's `do`-notation can be constructed by passing them as arguments for subsequently composed functions; the implementation of `bind` is shorter than the one of `join` (we don't have to unwrap Monads for `join`); and some Monads such as the State Monad and the Cont Monad do not fit into the definitions of `join` very well. Note that Maybe and Exception Monads align to the definition of `join` because threading two Monads is a good analogy in understanding Maybe and Exception. However, the State and Cont Monads are, in fact, a function so threading two functions is possible, but it is hard to get it intuitively.

[1] Disclaimer. This is my personal opinion so it might be controversial or even incorrect.

Monad	Semantics	TA	Describtion
Identity	Identity	A	Return as it is, useful in Monad Transformers.
Maybe	Partiality	$A + \{\bot\}$	Return may not contain a value.
Error	Exception	$A + E$	Return may contain some errors.
List	Non-determinism	$P(A)$	Return can be one of multiple values.
Reader	Input	$C^* + A$	Computations can read shared input.
Writer	Output	$C^* \times A$	Computations can write some output.
Cont	Continuation	$R^{(R^A)}$	Computations can be interrupted.
State	Side-effects	$(A \times S)^S$	Computations maintain state.
IO	Realworld State	$(A \times S)^S$	Computations maintain IO state.

A = Object P = Power-set E = Exceptions C = Set of characters S = Set of states

Table 1: Short summary of example Monads available in the `mtl` Haskell library [1] with their Category Theoretic semantics [15]. Monads are ordered based on their implementation styles of the composition code, which we described in §2.2 more details.

Monad Laws Third, Haskell doesn't have any safe mechanism (verification in compile time) to catch violations of Monad laws. If a Monad instance syntactically provides all constituents, Haskell considers it as a Monad. By convention, however, all Monads in the standard libraries conform to the Monad laws, and also programmers expect it from any custom Monad as well.

1. Left identity law: `return x >>= f == f x`
2. Right identity law: `c >>= return == c`
3. Associativity law: `c >>= ( λ x → f x >>= g) == (c >>= f) >>= g`

Left and right identity laws are basically the unit law in Category Theory. Those rules are trivially satisfied in most of "sane" data structures that represent an abstraction for actual uses. Then, what do the rules mean for programmers in Haskell? It means that the data type does not work weirdly when using it as a `do`-notation.

2.2 Examples

Each Monad is deserved for a few pages of an explanation, but in this paper, we will explore Monads in terms of their implementation structures. In particular, metaphors of one type of structures never fit well into the other types of structures, and importantly we believe this is the reason why people think using metaphors in understanding Monads is a common fallacy to learn Monads.

In Table 1, we classified Monads based on their implementation styles of the `bind` function. The first type of Monads, from Identity to List, implements the composition as a value (kinds of $* \to *$ in Haskell), and the second type of Monads, from Reader to IO, implements them as a function (kinds of $* \to * \to *$ in Haskell). For example, the constructor of the Maybe Monad (`Maybe a` in Figure 3(a)) abstracts a type `a` as a

value, the State Monad (`State s a` in Figure 3(b)) embeds a type `a` in a function getting a state `s` as an argument, even though its type signature is just `State s a` [2].

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

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

(a) Maybe Monad

```haskell
data State s a = State (s -> (a, s))

instance Monad (State s) where
  return x = State $ \s -> (x, s)
  ...
```

(b) State Monad

Figure 3: Maybe and State Monad in Haskell (simplified for clarity).

Why do I differentiate two types of Monads? It enables somewhat subtle implementation tricks. The first type of Monads uses a composition logic to combine two values, Monad instances, but the second type utilizes a composition logic to enforce orders. The latter purpose becomes particularly important in IO Monad that we will see in the next section, §3.

3 IO Monad

Among various interesting features in Haskell, one that is remarkably different from other program languages is IO Monad. In fact, IO Monad is exactly same as the State Monad that enables computations to maintain state in the Monad compositions. Then, why Haskell has the definition of the IO Monad separately from the State Monad? Answering this question is one of our goals in this section, and we will conclude it at the end. Before going further, let's understand the role of the IO Monad in Haskell.

Haskell is a purely functional programming language, in which there are no side-effects allowed. However, purely functional programs are useless in the real world; they can not print out the computed results at all. For this reason, Haskell uses the IO Monad to interact with the real world, where there are full of dirty side-effects. A common way to understand benefits of using the IO Monad is with a tagging metaphor, thereby enforcing the clear separation of pure functions from the real world. With the IO Monad, all inputs and outputs from the real world are tagged with the IO type constructor. For example,

```haskell
getLine :: IO String
```

`getLine()` inputs a line from a user and returns the line with the IO tag attached, `IO String` instead of `String` along. Since there is no way to drop the `IO` tag and extract the actual `String` value, Haskell programs that want to access the `String` inside the `IO String` should use the standard Monad interfaces, »= (bind) and `return` (unit). In other words, without using the Monad interfaces, functions can not interact with the external worlds, and thus can maintain their pure functions that are separated from the real world.

IO Monad as tagging is not an incorrect way of describing its useful properties in general, but it does not answer an important question, why IO Monad is particularly

[2] For the purpose of juxtaposing two Monads, I simplified the code a lot because Haskell libraries implement each Monad by using a corresponding Monad Transformer with the Identity Monad.

```haskell
-- IO data constructor
data IO a = IO (RealWorld -> (RealWorld, a))

-- IO Monad instance
instance Monad IO where
  -- unit: return the state with the input
  return x = IO $ \s -> (s, x)

  -- bind: run the first computation and return the next one
  (>>=) (IO m) f = IO $ \s ->
      case m s of (s', a) ->
          unIO (f a) s'
```

Figure 4: The implementation of IO Monad in Haskell (simplified for clarity).

useful, or even required, in Haskell. As we stated before, Haskell is purely functional so there are no side-effects allowed. Since every function does not have side-effects, it should have the same return value whenever we call it with the same input arguments. Then, how can we implement the `getLine()` function in Haskell? `getLine()` might return different values whenever we call, depending on what users typed. The magic behind the `getLine()` function is its type signature, `IO String`.

As we discussed in §2, the IO Monad is the second type as we classified, returning a function in its composition, so the return type, `IO String`, of `getLine()`, is a function. In fact, given an argument (none!), `getLine()` consistently returns the same output value, the function, `IO String`. In other words, if we run `getLine()` with a state of the real world, it will return one line that the user typed, along with the state of another world. Let's see the actual implementation of the IO Monad in §3. In fact, the definition is exactly same as the one of the State Monad in Figure 3(b), except the IO Monad gets `RealWorld` instead of `State` as a state argument.

What is `RealWorld` and how it differs from `State`? Surprisingly, unlike the `State` type contains an arbitrary type `s`, the `RealWorld` type is nothing but its name, implemented as below:

```haskell
data RealWorld
```

This kind of types is called a phantom type because it contains nothing, and used to enforce evaluation orders in the IO Monad. Since Haskell is a lazy (non-strict) language, it never evaluates expressions until they are actually used. The laziness enables Haskell programs to skip unused expressions and thus gives some performance benefits. However, it makes a programmer embarrassingly hard to write an imperative code, which is, sadly, often required in system software programs.

How does Haskell enforce orders with the execution of Monads? Suppose we are implementing a function, `three()`, adding two return values of `one()` and `two()` functions, as in Figure 5(a) and Figure 5(d). Both implementations are clean and pure, but for some reasons, we want to switch the evaluation orders of `one()` and `two()`. In C, the program runs as written down in code (imperative), so swapping both statements directly changes the invocation orders, as in Figure 5(b). In Haskell, however, we need to use a Monad to thread the sequence of two computations (declarative and functional). In Figure 5(e), the IO Monad of `two()` depends on the `RealWorld` state returned by the

```c
int one(void) {
  return 1;
}
int two(void) {
  return 2;
}
int three(void) {
  int a = one();
  int b = two();
  return a + b;
}
```

(a) Pure functions

```c
int one(void) {
  return 1;
}
int two(void) {
  return 2;
}
int three(void) {
  int b = two();
  int a = one();
  return a + b;
}
```

(b) Enforced orders

```c
int one(void) {
  printf("1"); return 1;
}
int two(void) {
  printf("2"); return 2;
}
int three(void) {
  int a = one();
  int b = two();
  return a + b;
}
```

(c) Side-effects (IO)

```haskell
one = 1
two = 2
three = a + b
  where a = one
        b = two
```

(d) Pure functions

```haskell
one = return 1
two = return 2
three = do
  b <- two
  a <- one
  return (a + b)
```

(e) Enforced orders

```haskell
one = print 1 >> return 1
two = print 2 >> return 2
three = do
  a <- one
  b <- two
  return (a + b)
```

(f) Side-effects (IO)

Figure 5: Examples of code in C and Haskell to demonstrate three constructions; composing pure functions in (a) and (d); enforcing orders in (b) and (e); and embedding side-effects in (c) and (f). In all examples, `three()` invoke `one()` and `two()`, and returns the sum of their return values.

IO Monad of `one()`, and thus the sequence of invocations is enforced. In other worlds, all of the IO Monads are threaded via the `RealWorld` state that each Monad gets as an argument and returns as an output, thereby keeping the sequence of side-effects in the Haskell program, Figure 5(f).

In summary, the IO Monad is nothing but the State Monad but composing with the shared `RealWorld` state. Whereas the `State` can change its internal state, `RealWorld` doesn't have an interface to change its state. Therefore, IO Monad can separate the pure functions from the real world. In fact, `RealWorld` of the IO Monad is nothing (a phantom type) but to enforce the execution orders in a lazy language.

4 S Monad

Since Haskell has been defined and used for nearly 30 years now, there are a variety of attempts to use Haskell for writing system software programs: x86 assembler [4], web framework [7, 8], even OSes [9] or embedded systems [12]. All system programs use Monads, one way or another, to enable imperative style programmings in the purely functional and lazy Haskell programming language [16].

However, their main purpose of using Monads is to make them easy to translate a already-written imperative program to a working piece of Haskell code. For example, HouseOS [9], whose large body of the kernel logic is written in Haskell, defines `H` Monad to access hardware such as memory and registers in the x86 machine, but its body in fact is just the IO Monad. Although `H` provides a proof of the memory safety, ensuring that all memory operations never corrupt the Haskell heap, it was failed to

escape the traditional interfaces that every commodity OSes strictly follows, the system call abstraction.

The main purpose of using the system call abstraction in OSes is to provide the clear separation of privileges between user processes (ring 3) and the kernel (ring 0), thereby protecting the system from the failures of the user processes. In other words, the system call abstraction is an interface to interact with the operating system. In the perspective of user programs, it is nothing but a list of available APIs that they can freely invoke. Here is a running example that we will explore in this section.

4.1 Motivation

```
1 // exit if file is not writable
2 if (access("file", W_OK) != 0) {
3     err(1, "access");
4 }
5
6 // open file and write buf to the file
7 fd = open("file", O_WRONLY);
8 write(fd, "hello world", 12);
9 close(fd);
```

Figure 6: A code snippet to open a file and write a string to it, if the file is writable.

The example code invokes four system calls in order; `access()` to check if a file path is writable; `open()` to create a corresponding file descriptor; `write()` to write a string buffer to a filesystem; and finally `close()` to close the opened file descriptor. This is a very simple code snippet, but there are a few interesting properties to be explained before going further.

First of all, any system call can fail; they are dealing with a fragile hardware for us. For example, `access()` can fail if the path is not writable, and `write()` can also fail if the disk is full. In the above code, the error condition of `access()` is properly handled, but the code is still vulnerable to the failures of `open()`, `write()` and `close()`. It is common to ignore error checking in software programs, because humans are lazy and importantly it rarely occurs in practice, like out-of-memory or lack of disk storage for example.

Second, those system calls encode error conditions inconsistently: `access()` returns -1 on errors, and `open()` returns a negative integer on errors. It is hardly consistent in embedding a failure (called a bottom condition, $\perp$) in the return type, `Int`, of each system call, so they use a special global variable `errno` to inform user programs. For example, `errno` of `open()` can contain `EEXIST` if a file already exists, and `ENOMEM` if the OS is out of system resources.

Third, the system state can change during the execution of a user program, thereby making it vulnerable to the time-of-check-to-time-of-use (TOCTOU) attacks. The above code, in fact, is insecure, so should not be used in a production code. For example, if an attacker can interleave between `access()` and `open()` system calls, and link a private file to the one that this program is going to overwrite, then the attacker can successfully delete any user's private file, without acquiring user's permission.

4.2 Properties

```
main = do
  err <- runS $ do
    access "file" W_OK
    fd <- open "file" O_WRONLY
    write fd "hello world"
```

Figure 7: An ideal implementation of the running example in Haskell.

Suppose we have an imaginary Monad to solve above problems, and call it the S Monad. Using the S Monad, the example code becomes simple and clean—the second `do` block runs S Monad and returns `err` to abstract possible errors. In the syntax wise, we at least need neither trivial error handling, nor manual de-allocation of the opened file descriptor in the S Monad. Here is a list of benefits that we expect from the S Monad.

Unified error handling. The S Monad should abstract the possible failures in system calls, by using an Error Monad Transformer, `ErrorT e m a`. To handle errors uniformly, all system calls should return `ErrorT SysErr S a`, where `SysErr` represents an error type when it fails, and a represents a return type of the system call, like a file descriptor. Since we embed the S Monad in the context of `ErrorT`, all computations stop immediately after any failure, unless programmers explicitly check errors from the system calls.

Privileged execution. When executing the S Monad with `runS`, it changes its privilege from the user mode (ring 3) to the kernel mode (ring 0), and unwraps the Monad with two states, `OSState` and UProcState. In other words, the composed S Monads executes in the context of the kernel mode (ring 0), and thus can avoid the cost of context switching between privileged code and the user space. Since the S Monad is well-managed and all errors immediately short-circuit to the user mode, user programs can not exploit the privileged execution at any moment.

Transactional execution. The S Monad executes as a single unit, whether all succeed or all fail (as one transaction). By using the S Monad, we can avoid the TOCTOU security problem. If one of system calls failed, `OSState` is restored to the previous state in which `runS` initiated, like Software Transcational Memory (STM) [10]. If the whole S Monad succeed, `OSState` will be updated atomically, so other user processes will see `OSState` after that.

Automatically de-allocating resources. Instead of de-allocating the opened file descriptors in C code, the S Monad should manage the life time of resources that the current process created. This technique has been explored in terms of region-based analysis, meaning that the life time of each resource is same as the life time of the region it allocate that resource. By keeping track of the ownership of each resource and corresponding regions, the S Monad can automatically de-allocate process-specific resources like file descriptors [12].

4.3 Toward implementation

First two properties are immediately achievable with minor modifications in the Haskell standard library, because `ErrorT` provides a standard way to lift a Monad for handling errors and `MonadIO` allows any computation to embed `IO` with `liftIO`. However, last two properties require to combine two discrete projects, namely Position [12] for reclaiming resources and STM [10] for enabling the transactional execution. In particular, Position encodes a number into the type as a type label (tagging), which is not known widely but deserved to emphasize its importance. To adopt its structure in the S Monad, we first need to verify what Position proposed can be applied in managing multiple resources simultaneously in the OS context. In addition, we need to verify that all of data structures that STM provides, such as `TArray` and `TQueue`, are enough for constructing basic primitives to represent `OSState` too. Here are the type signatures of the S Monad and related data structures that I expected from S Monad.

```haskell
-- types for system IO and return value
data SysIO a  = StateT (UProcState, OSState) IO a
data SysRet a = ErrorT SysErr (SysIO a) a

-- S Monad
data S a = S {unS :: STM SysRet a}

-- types definitions of each system call
access :: Path -> IOMode -> SysRet Bool
open   :: Path -> IOMode -> SysRet Handle
write  :: Handle -> Path -> SysRet Int

-- run S Monad atomically
runS :: S a -> SysRet a
runS m = atomically $ unS m
```

Figure 8: The S Monad and related data types.

This paper never tells you about how to implement them, but realizing the idea of the S Monad is my next project; it probably takes longer than a half year. Over the course of this journey, it was a great opportunity for me to formalize the goals, benefits, and possibility of the S Monad.

5 Summary

Starting from the Category-theoretic definition of Monads, we explored how they are implemented in a purely functional programming language, Haskell. We also classified Monads by their implementation styles and it turns out that a common fallacy of using metaphors for teaching Monads can be understood in this vein. Furthermore, we have figured out why the IO Monad is particularly important in Haskell and what its roles are. Lastly, we proposed a new Monad, the S Monad, to represent the system call abstractions, and shed some light on the possibility in implementing a new Haskell OS in the future.

References

[1] mtl-2.1.2: Monad classes, using functional dependencies, 2013. http://hackage.haskell.org/package/mtl-2.1.2.

[2] HaskellWiki: Monads as containers, 2013. http://www.haskell.org/haskellwiki/Monads_as_containers.

[3] HaskellWiki: Monad, 2013. http://www.haskell.org/haskellwiki/Monad.

[4] The Potential Programming Language: A type-safe x86-64 systems programming language, 2013. http://potential-lang.org.

[5] Haskell/Category theory, 2013. http://en.wikibooks.org/wiki/Haskell/Category_theory.

[6] Haskell/Understanding monads, 2013. http://en.wikibooks.org/wiki/Haskell/Understanding_monads.

[7] Yesod Web Framework for Haskell, 2013. http://www.yesodweb.com.

[8] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo. Hails: protecting data privacy in untrusted web applications. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 47–60, Berkeley, CA, USA, 2012. USENIX Association.

[9] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A principled approach to operating system construction in haskell. In *Proceedings of the 10th ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 116–128, New York, NY, USA, 2005. ACM.

[10] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM.

[11] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 1998.

[12] O. Kiselyov and C. chieh Shan. Position: Lightweight Static Resources: Sexy types for embedded and systems programming. In *Proceedings of the 8th Symposium on Trends in Functional Programming*, TFP '07, 2007.

[13] C. Mcbride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, Jan. 2008. ISSN 0956-7968.

[14] T. McGuire. Quote o' the day: Bestest monad metaphor ever!, 2013. http://maniagnosis.crsr.net/2012/06/quote-o-day-bestest-monad-metaphor-ever.html.

[15] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991. ISSN 0890-5401.

[16] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 71–84, New York, NY, USA, 1993. ACM. ISBN 0-89791-560-7. doi: 10.1145/158511.158524. URL http://doi.acm.org/10.1145/158511.158524.

[17] D. Spiewak. Monads are not metaphors, 2013. http://www.codecommit.com/blog/ruby/monads-are-not-metaphors.

[18] D. Spivak. *Category theory for scientists*. 2013.

[19] B. Yorgey. Abstraction, intuition, and the "monad tutorial fal- lacy", 2013. http://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy.

6 &DWWJRUV 7KHRUV IRU6FLHQWLVWV
6SULQJ 201