The Agent's new Cloths
Towards functional Agent-Based Simulation

[Anonymised review copy]

Abstract: TODO: parallelism for free because all isolated e.g. running multiple replications or parameter-variations

TODO: it is paramount not to write against the established approach but for the functional approach. not to try to come up with arguments AGAINST the object-oriented approach but IN FAVOUR for the functional approach. In the end: dont tell the people that what they do sucks and that i am the saviour with my new method but: that i have a new method which might be of interest as it has a few nice advantages.

So far, the pure functional paradigm hasn't got much attention in Agent-Based Simulation (ABS) where the dominant programming paradigm is object-orientation, with Java, Python and C++ being its most prominent representatives. We claim that pure functional programming using Haskell is very well suited to implement complex, real-world agent-based models and brings with it a number of benefits. To show that we implemented the seminal Sugarscape model in Haskell in our library *FrABS* which allows to do ABS the first time in the pure functional programming language Haskell. To achieve this we leverage the basic concepts of ABS with functional reactive programming using Yampa. The result is a surprisingly fresh approach to ABS as it allows to incorporate discrete time-semantics similar to Discrete Event Simulation and continuous time-flows as in System Dynamics. In this paper we will show the novel approach of functional reactive ABS through the example of the SIR model, discuss implications, benefits and best practices.

Keywords: Agent-Based Simulation, Functional Programming, Haskell

Introduction

1.1 The traditional approach to Agent-Based Simulation (ABS) has so far always been object-oriented techniques, due to the influence of the seminal work of Epstein et al Epstein & Axtell (1996) in which the authors claim "[..] object-oriented programming to be a particularly natural development environment for Sugarscape specifically and artificial societies generally [..]" (p. 179). This work established the metaphor in the ABS community, that *agents map naturally to objects* (North & Macal 2007) which still holds up today.

1.2 In this paper we challenge this metaphor and explore ways of approaching ABS using the functional programming paradigm as in the language Haskell. By doing this we expect to leverage the benefits of it (Hudak et al. 2007) to become available when implementing ABS functionally: expressing *what* a system is instead of *how* it works through declarative code, being explicit about the interactions of the program with the real world, explicit data-centric programming resulting in less sources of bugs and a strong static type system making type-errors at run-time obsolete.

1.3 We show that these functional concepts result in an approach to implementing ABS where it is harder to make mistakes and allow to implement simulations which are guaranteed to be reproducible, have less sources of bugs, are easier to verify and thus more likely to be correct which is of paramount importance in high-impact scientific computing.

1.4 As a use-case throughout the paper we employ the well known SugarScape model (Epstein & Axtell 1996) to demonstrate our case-studies, because it can be seen as one of the most influential models in ABS and it laid the foundations of object-oriented implementation of agent-based models.

1.5 The aim of this paper is show *how* to implement ABS in functional programming as in Haskell and *why* it is of benefit of doing so. Further, we give the reader a good understanding of what functional programming is, what the challenges are in applying it to ABS and how we solve these in our approach.

1.6 The paper makes the following contributions:

- It is the first to *systematically* introduce the functional programming paradigm, as in Haskell, to ABS, identifying its benefits, difficulties and drawbacks.

- We show how functional ABS can be scaled up to massively large-scale without the problems of low level concurrent programming using Software Transactional Memory (STM). Although there exist STM implementations in non-functional languages like Java and Python, due to the nature of Haskells type-system, the use of STM has unique benefits in this setting.
- Further we introduce a powerful and very expressive approach to testing ABS implementations using property-based testing. This allows a much more powerful way of expressing tests, shifting from unit-testing towards specification-based testing. Although property-based testing has been brought to non-functional languages like Java and Python as well, it has its origins in Haskell and it is here where it truly shines.

Background

2.1 In this background section we will introduce the concepts behind functional programming and functional reactive programming as they form the very foundation of our approach to pure functional Agent-Based Simulation. Note that both fields build on each other and are vast and complex. A more in-depth handling of both subjects are out of scope of this paper and we refer the reader to additional resources where appropriate. In this paper we try to avoid too much technical details and present only the fundamental concepts behind our approach. [1].

Established Concepts

2.2 The established approach to implement ABS falls into three categories:

1. Programming from scratch using object-oriented languages where Java and Python are the most popular ones.
2. Programming using a 3rd party ABS library using object-oriented languages where RePast and DesmoJ, both in Java, are the most popular one.
3. Using a high-level ABS tool-kit for non-programmers, which allow customization through programming if necessary. By far the most popular one is NetLogo with an imperative programming approach followed by AnyLogic with an object-oriented Java approach.

In general one can say that these approaches, especially the 3rd one, support fast prototyping of simulations which allow quick iteration times to explore the dynamics of a model. Unfortunately, all of them suffer the same problems when it comes to verifying and guaranteeing the correctness of the simulation.

The established way to test software in established object-oriented approaches is writing unit-tests which cover all possible cases. This is possible in approach 1 and 2 but very hard or even impossible when using an ABS tool-kit, as in 3, which is why this approach basically employs manual testing. In general writing those tests or conducting manual tests is necessary because one cannot guarantee the correct working at compile-time which means testing ultimately tests the correct behaviour of code at run-time. The reason why this is not possible is due to the very different type-systems and paradigm of those approaches. Java has a strong but very dynamic type-system whereas Python is completely dynamic not requiring the programmer to put types on data or variables at all. This means that due to type-errors and data-dependencies run-time errors can occur which origins might be difficult to track down.

Functional Programming

2.3 In his 1977 ACM Turing Award Lecture, John Backus [2] fundamentally critizied imperative programming for its deep flaws and proposed a functional style of programming to overcome the limitations of imperative program-

[1] For interested readers, we refer to our companion paper TODO: cite / refer to existing repository. In it we introduce the functional programming community to ABS by deriving our approach step-by-step by implementing an agent-based implementation of the SIR model. In this companion paper we assume good knowledge and beyond of the fundamental concepts of FP and FRP and provide in-depth discussions of technical details.

[2] One of the giants of Computer Science, a main contributor to Fortran - an imperative programming language.

ming Backus (1978). The main criticism is its use of *state-transition with complex states* and the inherent semantics of state-manipulation. In the end an imperative program consists of a number of assign-statements resulting in side-effects on global mutable state which makes reasoning about programs nearly impossible. Backus proposes the so called *applicative* computing, which he termes *functional programming* which has its foundations in the Lambda Calculus Church (1941). The main idea behind it is that programming follows a declarative rather than an imperative style of programming: instead of describing *how* something is computed, one describes *what* is computed. This concept abandons variables, side-effects and (global) mutable state and resorts to the simple core of function application, variable substitution and binding of the Lambda Calculus. Although possible and an important step to understand the very foundations, one does not do functional programming in the Lambda Calculus Michaelson (2011), as one does not do imperative programming in a Turing Machine.

2.4 MacLennan MacLennan (1990) defines Functional Programming as a methodology and identifies it with the following properties (amongst others):

1. It is programming without the assignment-operator.
2. It allows for higher levels of abstraction.
3. It allows to develop executable specifications and prototype implementations.
4. It is connected to computer science theory.
5. Suitable for Parallel Programming.
6. Algebraic reasoning.

Allen & Moronuki (2016) defines Functional Programming as "a computer programming paradigm that relies on functions modelled on mathematical functions." Further they explicate that it is

- in Functional programming programs are combinations of expressions
- Functions are *first-class* which means the can be treated like values, passed as arguments and returned from functions.

MacLennan (1990) makes the subtle distinction between *applicative* and *functional* programming. Applicative programming can be understood as applying values to functions where one deals with pure expressions:

- Value is independent of the evaluation order.
- Expressions can be evaluated in parallel.
- Referential transparency.
- No side effects.
- Inputs to an operation are obvious from the written form.
- Effects to an operation are obvious from the written form.

Note that applicative programming is not necessarily unique to the functional programming paradigm but can be emulated in an imperative language e.g. C as well. Functional programming is then defined by MacLennan (1990) as applicative programming with *higher-order* functions. These are functions which operate themselves on functions: they can take functions as arguments, construct new functions and return them as values. This is in stark contrast to the *first-order* functions as used in applicative or imperative programming which just operate on data alone. Higher-order functions allow to capture frequently recurring patterns in functional programming in the same way like imperative languages captured patterns like GOTO, while-do, if-then-else, for. Common patterns in functional programming are the map, fold, zip, operators. So functional programming is not really possible in this way in classic imperative languages e.g. C as you cannot construct new functions and return them as results from functions [3].

The equivalence in functional programming to to the *;* operator of imperative programming which allows to compose imperative statements is function composition. Function composition has no side-effects as opposed

[3] Object-Oriented languages like Java let you to partially work around this limitation but are still far from *pure* functional programming.

to the imperative ; operator which simply composes destructive assignment statements which are executed after another resulting in side-effects. At the heart of modern functional programming is monadic programming which is polymorphic function composition: one can implement a user-defined function composition by allowing to run some code in-between function composition - this code of course depends on the type of the Monad one runs in. This allows to emulate all kind of effectful programming in an imperative style within a pure functional language. Although it might seem strange wanting to have imperative style in a pure functional language, some problems are inherently imperative in the way that computations need to be executed in a given sequence with some effects. Also a pure functional language needs to have some way to deal with effects otherwise it would never be able to interact with the outside-world and would be practically useless. The real benefit of monadic programming is that it is explicit about side-effects and allows only effects which are fixed by the type of the monad - the side-effects which are possible are determined statically during compile-time by the type-system. Some general patterns can be extracted e.g. a map, zip, fold over monads which results in polymorphic behaviour - this is the meaning when one says that a language is polymorphic in its side-effects.

TODO: explain closures TODO: explain continuations TODO: explain monads (explicit about side-effects), what are effects TODO: explain the term 'pure'

In our research we selected Haskell as our functional programming language. [4]. The paper of Hudak et al. (2007) gives a comprehensive overview over the history of the language, how it developed and its features and is very interesting to read and get accustomed to the background of the language. A widely used introduction to programming in Haskell is Hutton (2016). The main points why we decided to go for Haskell are

- Pure, Lazy Evaluation, Higher-Order Functions and Static Typing - these are the most important points for the decision as they form the very foundation for composition, correctness, reasoning and verification.
- Real-World applications - the strength of Haskell has been proven through a vast amount of highly diverse real-world applications [5] Hudak et al. (2007) and is applicable to a number of real-world problems O'Sullivan et al. (2008).
- Modern - Haskell is constantly evolving through its community and adapting to keep up with the fast changing field of computer science e.g. parallelism & concurrency.
- In-house knowledge - the School of Computer Science of the University of Nottingham has a large amount of in-house knowledge in Haskell which can be put to use and leveraged in my thesis.

The main conclusion of the classical paper Hughes (1989) is that *modularity* is the key to successful programming and can be achieved best using higher-order functions and lazy evaluation provided in functional languages like Haskell. The author argues that the ability to divide problems into sub-problems depends on the ability to glue the sub-problems together which depends strongly on the programming-language. He shows that laziness and higher-order functions are in combination a highly powerful glue and identifies this as the reason why functional languages are superior to structure programming. Another property of lazy evaluation is that it allows to describe infinite data-structures, which are computed as currently needed. This makes functions possible which produce an infinite stream which is consumed by another function - the decision of *how many* is decoupled from *how to*.

In the paper Wadler (1992) Wadler describes Monads as the essence of functional programming (in Haskell). Originally inspired by monads from category-theory (see below) through the paper of Moggi Moggi (1989), Wadler realized that monads can be used to structure functional programs Wadler (1990). A pure functional language like Haskell needs some way to perform impure (side-effects) computations otherwise it has no relevance for solving real-world problems like GUI-programming, graphics, concurrency,... . This is where monads come in, because ultimately they can be seen as a way to make effectful computations explicit [6]. In Wadler (1992) Wadler shows how to factor out the error handling in a parser into monads which prevents code to be cluttered by cross-cutting concerns not relevant to the original problem. Other examples Wadler gives are the propagating of mutable state, (debugging) text-output during execution, non-deterministic choice. Further applications of monads are given in Wadler (1992), Wadler (1995), Wadler (1997) where they are used for array updating,

[4] Although we did a bit of research using Scala (a mixed paradigm functional language) in ABS (see Appendix **??**), we deliberately ignored other functional languages as it is completely out-of-scope of this thesis to do an in-depth comparison of functional languages for their suitability to implement ABS.

[5] `https://wiki.haskell.org/Applications_and_libraries`

[6] This is seen as one of the main impacts of Haskell had on the mainstream programming Hudak et al. (2007)

interpreting of a language formed by expressions in algebraic data-types, filters, parsers, exceptions, IO, emulating an imperative-style of programming. This seems to be exactly the way to go, tackling the problems mentioned in the introduction: making data-flow explicit, allowing to factor out cross-cutting concerns and encapsulate side-effects in types thus making them explicit. It may seem that one runs into efficiency-problems in a pure functional programming language when using algorithms which are implemented in imperative languages through mutable data which allows in-place update of memory. The seminal work of Okasaki (1999) showed that when approaching this problem from a functional mind-set this does not necessarily be the case. The author presents functional data structures which are asymptotically as efficient as the best imperative implementations and discusses the estimation of the complexity of lazy programs.

The concept of monads was further generalized by Hughes in the concept of arrows Hughes (2000). The main difference between Monads and Arrows are that where monadic computations are parameterized only over their output-type, Arrows computations are parametrised both over their input- and output-type thus making Arrows more general. In Hughes (2005) Hughes gives an example for the usage for Arrows in the field of circuit simulation. Streams are used to advance the simulation in discrete steps to calculate values of circuits thus the implementation is a form of *discrete event simulation* - which is in the direction we are heading already with ABS. As will be shown below, the concept of arrows is essential for Functional Reactive Programming a potential way to do ABS in pure functional programming.

Functional Reactive Programming

2.5 TODO: explain streams TODO: Yampa, BearRiver, Dunai

2.6 Functional Reactive Programming (FRP) is a paradigm for programming hybrid systems which combine continuous and discrete components. Time is explicitly modelled: there is a continuous and synchronous time flow.

2.7 there have been many attempts to implement FRP in frameworks which each has its own pro and contra. all started with fran, a domain specific language for graphics and animation and at yale FAL, Frob, Fvision and Fruit were developed. The ideas of them all have then culminated in Yampa which is the reason why it was chosen as the FRP framework. Also, compared to other frameworks it does not distinguish between discrete and synchronous time but leaves that to the user of the framework how the time flow should be sampled (e.g. if the sampling is discrete or continuous - of course sampling always happens at discrete times but when we speak about discrete sampling we mean that time advances in natural numbers: 1,2,3,4,... and when speaking of continuous sampling then time advances in fractions of the natural numbers where the difference between each step is a real number in the range of [0..1])

2.8 **?** give a good overview of Yampa and FRP. Quote: "The essential abstraction that our system captures is time flow". Two *semantic* domains for progress of time: continuous and discrete.

2.9 The first implementations of FRP (Fran) implemented FRP with synchronized stream processors which was also followed by **?**. Yampa is but using continuations inspired by Fudgets. In the stream processors approach "signals are represented as time-stamped streams, and signal functions are just functions from streams to streams", where "the Stream type can be implemented directly as (lazy) list in Haskell...":

2.10 "A major design goal for FRP is to free the programmer from 'presentation' details by providing the ability to think in terms of 'modeling'. It is common that an FRP program is concise enough to also serve as a specification for the problem it solves" **?**. This quotation describes exactly one of the strengths using FRP in ACE

Related Research

2.11 TODO: paper by James Odell "Objects and Agents Compared"

A functional approach

3.1 Due to the fundamentally different approaches of pure Functional Programming (pure FP) an ABS needs to be implemented fundamentally different as well compared to traditional object-oriented approaches (OO). We face the following challenges:

1. How can we represent an Agent?
   In OO the obvious approach is to map an agent directly onto an object which encapsulates data and provides methods which implement the agents actions. Obviously we don't have objects in pure FP thus we need to find a different approach to represent the agents actions and to encapsulate its state.

2. How can we represent state in an Agent?
   In the classic OO approach one represents the state of an Agent explicitly in mutable member variables of the object which implements the Agent. As already mentioned we don't have objects in pure FP and state is immutable which leaves us with the very tricky question how to represent state of an Agent which can be actually updated.

3. How can we implement proactivity of an Agent?
   In the classic OO approach one would either expose the current time-delta in a mutable variable and implement time-dependent functions or ignore it at all and assume agents act on every step. At first this seems to be not a big deal in pure FP but when considering that it is yet unclear how to represent Agents and their state, which is directly related to time-dependent and reactive behaviour it raises the question how we can implement time-varying and reactive behaviour in a purely functional way.

4. How can we implement the agent-agent interaction?
   In the classic OO approach Agents can directly invoke other Agents methods which makes direct Agent interaction *very* easy. Again this is obviously not possible in pure FP as we don't have objects with methods and mutable state inside.

5. How can we represent an environment and its various types?
   In the classic OO approach an environment is almost always a mutable object which can be easily made dynamic by implementing a method which changes its state and then calling it every step as well. In pure FP we struggle with this for the same reasons we face when deciding how to represent an Agent, its state and proactivity.

6. How can we implement the agent-environment interaction?
   In the classic OO approach agents simply have access to the environment either through global mechanisms (e.g. Singleton or simply global variable) or passed as parameter to a method and call methods which change the environment. Again we don't have this in pure FP as we don't have objects and globally mutable state.

7. How can we step the simulation?
   In the classic OO approach agents are run one after another (with being optionally shuffled before to uniformly distribute the ordering) which ensures mutual exclusive access in the agent-agent and agent-environment interactions. Obviously in pure FP we cannot iteratively mutate a global state.

Agent representation, state and proactivity

3.2 Whereas in imperative programming (the OO which we refer to in this paper is built on the imperative paradigm) the fundamental building block is the destructive assignment, in FP the building blocks are obviously functions which can be evaluated. Thus we have no other choice than to represent our Agents using a function which implements their behaviour. This function must be time-aware somehow and allow us to react to time-changes and inputs. Fortunately there exists already an approach to time-aware, reactive programming which is termed Functional Reactive Programming (FRP). This paradigm has evolved over the year and current modern FRP is built around the concept of a signal-function which transforms an input-signal into an output-signal. An input-signal can be seen as a time-varying value. Signal-functions are implemented as continuations which allows to capture local state using closures. Modern FRP also provides feedback functions which provides convenient methods to capture and update local state from the previous time-step with an initial state provided at time = 0.

3.3 - pure functions don't have a notion of communication as opposed to method calls in object-oriented languages like java

3.4 - time is represented using the FRP concept: Signal-Functions which are sampled at (fixed) time-deltas, the dt is never visible directly but only reflected in the code and read-only. - no method calls => continuous data-flow instead

3.5 Viewing agent-agent interaction as simple method calls implies the following: - it takes no time - it has a synchronous and transactional character - an agent gives up control over its data / actions or at least there is always the danger that it exposes too much of its interface and implementation details. - agents equals objects, which is definitely NOT true. Agents

3.6 data-flow synchronous agent transactions

3.7 - still need transactions between two agents e.g. trading occurs over multiple steps (makeoffer, accept/refuse, finalize/abort) -> exactly define what TX means in ABS -> exclusive between 2 agents -> state-changes which occur over multiple steps and are only visible to the other agents after the TX has commited -> no read/write access to this state is allowed to other agents while the TX is active -> a TX executes in a single time-step and can have an arbitrary number of tx-steps -> it is easily possible using method-calls in OOP but in our pure functional approach it is not possible -> parallel execution is being a problem here as TX between agents are very easy with sequential -> an agent must be able to transact with as many other agents as it wants to in the same time-step -> no time passes between transactions => what we need is a 'all agents transact at the same time' -> basically we can implement it by running the SFs of the agents involved in the TX repeatedly with dt=0 until there are no more active TXs -> continuations (SFs) are perfectly suited for this as we can 'rollback' easily by using the SF before the TX has started

Environment representation and interaction

3.8 no global shared mutable environment, having different options: - non-active read-only (SIR): no agent, as additional argument to each agent - pro-active read-only (?): environment as agent, broadcast environment updates as data-flow - non-active read/write (?): no agent, StateT in agents monad stack - pro-active read/write (Sugarscape): environment as agent, StateT in agents monad stack

3.9 care must be taken in case of agent-transactions: when aborting/refusing all changes to the environment must be rolled back => instead of StateT use a transactional monad which allows us to revert changes to a save point at the start of the TX. if we drag the environment through all agents then we could easily revert changes but that then requires to hard-code the environment concept deep into the simulation scheduling/stepping which brings lots of inconveniences, also it would need us to fold the resulting multiple environments back into a single. If we had an environment-centric view then probably this is what we want but in ABS the focus is on the agents

3.10 question is if the TX sf runs in the same monad aw the agent or not. i opt for identity monad which prevents modification of the Environment in a transaction

3.11 also need to motivate the dt=0 in all TX processing: conceptually it all happens instantaneously (although arbitration is sequential) but agents must act time-sensitive

3.12 for environment we need transactional and shared state behaviour where we can have mutual exclusive access to shared data but also roll back changes we made. it should run deterministic when running agents not truly parallel. solution: run environment in a transactional state monad (TX monad). although the agents are executed in parallel in the end it (map) runs sequentially. this passes a mutable state through all agents which can act on it an roll back actions e.g. in case of a failed agent TX. if we dont need transactional behaviour then just use StateT monad. this ensures determinism. pro active environment is also easily possible by writing to the state. this approach behaves like sequential transactional although the agents run in parallel but how is this possible when using mapMSF ?

Stepping the simulation

3.13 - parallel update only, sequential is deliberately abandoned due to: -> reality does not behave this way -> if we need transactional behaviour, can use STM which is more explicit -> it is translates directly to a map which is very easy to reason about (sequential is basically a fold which is much more difficult to reason about) -> is more natural in functional programming -> it exists for 'transactional' reasons where we need mutual exclusive access to environment / other agents -> we provide a more explicit mechanism for this: Agent Transactions

References

Allen, C. & Moronuki, J. (2016). *Haskell Programming from First Principles*. Allen and Moronuki Publishing. Google-Books-ID: 5FaXDAEACAAJ

Backus, J. (1978). Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs. *Commun. ACM*, *21*(8), 613–641. doi:10.1145/359576.359579
URL http://dx.doi.org/10.1145/359576.359579

Church, A. (1941). *The Calculi of Lambda-conversion*. Princeton University Press. Google-Books-ID: KCOuGztKVgcC

Epstein, J. M. & Axtell, R. (1996). *Growing Artificial Societies: Social Science from the Bottom Up*. Washington, DC, USA: The Brookings Institution

Hudak, P., Hughes, J., Peyton Jones, S. & Wadler, P. (2007). A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, (pp. 12–1–12–55). New York, NY, USA: ACM. doi:10.1145/1238844.1238856
URL http://dx.doi.org/10.1145/1238844.1238856

Hughes, J. (1989). Why Functional Programming Matters. *Comput. J.*, *32*(2), 98–107. doi:10.1093/comjnl/32.2.98
URL http://dx.doi.org/10.1093/comjnl/32.2.98

Hughes, J. (2000). Generalising Monads to Arrows. *Sci. Comput. Program.*, *37*(1-3), 67–111. doi:10.1016/S0167-6423(99)00023-4
URL http://dx.doi.org/10.1016/S0167-6423(99)00023-4

Hughes, J. (2005). Programming with Arrows. In *Proceedings of the 5th International Conference on Advanced Functional Programming*, AFP'04, (pp. 73–129). Berlin, Heidelberg: Springer-Verlag. doi:10.1007/11546382_2
URL http://dx.doi.org/10.1007/11546382_2

Hutton, G. (2016). *Programming in Haskell*. Cambridge University Press. Google-Books-ID: 1xHPDAAAQBAJ

MacLennan, B. J. (1990). *Functional Programming: Practice and Theory*. Addison-Wesley. Google-Books-ID: JqhQAAAAMAAJ

Michaelson, G. (2011). *An Introduction to Functional Programming Through Lambda Calculus*. Courier Corporation. Google-Books-ID: gKvwPtvsSjsC

Moggi, E. (1989). Computational Lambda-calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, (pp. 14–23). Piscataway, NJ, USA: IEEE Press
URL http://dl.acm.org/citation.cfm?id=77350.77353

North, M. J. & Macal, C. M. (2007). *Managing Business Complexity: Discovering Strategic Solutions with Agent-Based Modeling and Simulation*. Oxford University Press, USA. Google-Books-ID: gRATDAAAQBAJ

Okasaki, C. (1999). *Purely Functional Data Structures*. New York, NY, USA: Cambridge University Press

O'Sullivan, B., Goerzen, J. & Stewart, D. (2008). *Real World Haskell*. O'Reilly Media, Inc., 1st edn.

Wadler, P. (1990). Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, (pp. 61–78). New York, NY, USA: ACM. doi:10.1145/91556.91592
URL http://dx.doi.org/10.1145/91556.91592

Wadler, P. (1992). The Essence of Functional Programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, (pp. 1–14). New York, NY, USA: ACM. doi:10.1145/143165.143169
URL http://dx.doi.org/10.1145/143165.143169

Wadler, P. (1995). Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, (pp. 24–52). London, UK, UK: Springer-Verlag
URL http://dl.acm.org/citation.cfm?id=647698.734146

Wadler, P. (1997). How to Declare an Imperative. *ACM Comput. Surv.*, *29*(3), 240–263. doi:10.1145/262009.262011
URL http://dx.doi.org/10.1145/262009.262011