The Agent's new Cloths

Towards functional programming in Agent-Based Simulation

JONATHAN THALER and THORSTEN ALTENKIRCH, University of Nottingham, United Kingdom

TODO: implement synchronous agent interaction

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: don't 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 functional programming using Haskell is very well suited to implement complex, real-world agent-based models and brings with it a number of benefits. In this paper we will introduce the reader to the functional programming paradigm and explain how it can be applied to implementing ABS. Further we discuss benefits and advantages. As use-case we implemented the seminal Sugarscape model in Haskell.

Additional Key Words and Phrases: Agent-Based Simulation, Functional Programming, Haskell

ACM Reference Format:
Jonathan Thaler and Thorsten Altenkirch. 2019. The Agent's new Cloths: Towards functional programming in Agent-Based Simulation. 1, 1 (July 2019), 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

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 [8] 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* [27] which still holds up today.

In this paper we challenge this metaphor and explore ways of approaching ABS using the functional programming paradigm with the language Haskell. We present fundamental concepts and advanced features of functional programming and we show how to leverage the benefits of it [13] to become available when implementing ABS functionally.

We claim that by using functional programming for implementing ABS it is easier to add parallelism and concurrency, it is harder to make mistakes, the resulting simulations are easier to test and verify, guaranteed to be reproducible, have less potential sources of bugs and are ultimately more likely to be correct. Still we need solid testing to verify and validate our software for which we will introduce property-based testing. All of this is of paramount

Authors' address: Jonathan Thaler, jonathan.thaler@nottingham.ac.uk; Thorsten Altenkirch, thorsten.altenkirch@nottingham.ac.uk, University of Nottingham, 7301 Wollaton Rd, Nottingham, NG8 1BB, United Kingdom.

2019. XXXX-XXXX/2019/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

importance in scientific computing, in which results need to be reproducible and correct while simulations should be able to massively scale-up as well.

Also we show that the concepts can be also applied to Discrete Event Simulation (DES) and System Dynamics (SD), making functional programming an attractive alternative for multi-method implementations.

The aim of this paper is to conceptually show *how* to implement ABS in functional programming using 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. Although functional programming is a highly technical subject, we avoid technical discussions and follow a very high-level approach, focusing on the concepts instead of their implementation. For readers which are interested in learning functional programming and go into technical details we refer to relevant literature in the respective parts of the paper.

The paper makes the following contributions:

- To the best of our knowledge, we are the first to introduce the functional programming paradigm using Haskell to ABS on a *conceptual* level, identifying benefits, difficulties and drawbacks.
- We show that functional programming concepts can be used to create simulation software which is easier to parallelise and add concurrency, has less sources of bugs, is more likely to be correct and guaranteed to be reproducible already at compile-time.
- We show that functional programming concepts can be used to test ABS implementations using property-based testing, which allows a very expressive way of testing, shifting from unit- towards specification-based testing.
- We show that the functional programming concepts can be applied to implement DES and SD simulations as well, making it an attractive multi-method implementation approach.

2 BUGS AND ERRORS IN AGENT-BASED SIMULATION

TODO: general introduction

The problem of correctness in agent-based simulations became more apparent in the work of Ionescu et al [18] which tried to replicate the work of Gintis [9]. In his work Gintis claimed to have found a mechanism in bilateral decentralized exchange which resulted in walrasian general equilibrium without the neo-classical approach of a tatonement process through a central auctioneer. This was a major break-through for economics as the theory of walrasian general equilibrium is non-constructive as it only postulates the properties of the equilibrium [5] but does not explain the process and dynamics through which this equilibrium can be reached or constructed - Gintis seemed to have found just this process. Ionescu et al. [18] failed and were only able to solve the problem by directly contacting Gintis which provided the code - the definitive formal reference. It was found that there was a bug in the code which led to the "revolutionary" results which were seriously damaged through this error. They also reported ambiguity between the informal model description in Gintis paper and the actual implementation. TODO: it is still not clear what this bug was, find out! look at the master thesis

This is supported by a talk [36], in which Tim Sweeney, CEO of Epic Games, discusses the use of main-stream imperative object-oriented programming languages (C++) in the context of Game Programming. Although the fields of games and ABS seem to be very different, in the end they have also very important similarities: both are simulations which perform

numerical computations and update objects in a loop either concurrently or sequential [10]. Sweeney reports that reliability suffers from dynamic failure in such languages e.g. random memory overwrites, memory leaks, accessing arrays out-of-bounds, dereferencing null pointers, integer overflow, accessing uninitialized variables. He reports that 50% of all bugs in the Game Engine Middleware Unreal can be traced back to such problems and presents dependent types as a potential rescue to those problems.

TODO: list common bugs in object-oriented / imperative programming TODO: java solved many problems TODO: still object-oriented / imperative ultimately struggle when it comes to concurrency / parallelism due to their mutable nature.

TODO: [39]

TODO: software errors can be costly TODO: bugs per loc

3 FUNCTIONAL PROGRAMMING

The roots of functional programming lie in the Lambda Calculus which was first described by Alonzo Church [2]. This is a fundamentally different approach to computation than imperative and object-oriented programming which roots lie in the Turing Machine [37]. Rather than describing *how* something is computed as in the more operational approach of the Turing Machine, due to the more declarative nature of the Lambda Calculus, code in functional programming describes *what* is computed.

In our research we are using the functional programming language Haskell. The paper of [13] 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. The main points why we decided to go for Haskell are:

- Rich Feature-Set - it has all fundamental concepts of the pure functional programming paradigm of which we explain the most important below.
- Real-World applications - the strength of Haskell has been proven through a vast amount of highly diverse real-world applications [13], is applicable to a number of real-world problems [29] and has a large number of libraries available [1].
- Modern - Haskell is constantly evolving through its community and adapting to keep up with the fast changing field of computer science. Further, the community is the main source of high-quality libraries.

As a short example we give an implementation of the factorial function in Haskell:

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

When looking at this function we can already see the central concepts of functional programming:

(1) Declarative - we describe *what* the factorial function is rather than how to compute it. This is supported by *pattern matching* which allows to give multiple equations for the same function, matching on its input.

(2) Immutable data - in functional programming we don't have mutable variables - after a variable is assigned, it cannot change its contents. This also means that there is no destructive assignment operator which can re-assign values to a variable. To change values, we employ recursion.

[1] https://wiki.haskell.org/Applications_and_libraries

(3) Recursion - the function calls itself with a smaller argument and will eventually reach the case of 0. Recursion is the very meat of functional programming because they are the only way to implement loops in this paradigm due to immutable data.
(4) Static Types - the first line indicates the name and the types of the function. In this case the function takes one Integer as input and returns an Integer as output. Types are static in Haskell which means that there can be no type-errors at run-time e.g. when one tries to cast one type into another because this is not supported by this kind of type-system.
(5) Explicit input and output - all data which are required and produced by the function have to be explicitly passed in and out of it. There exists no global mutable data whatsoever and data-flow is always explicit.
(6) Referential transparency - calling this function with the same argument will *always* lead to the same result, meaning one can replace this function by its value. This means that when implementing this function one can not read from a file or open a connection to a server. This is also known as *purity* and is indicated in Haskell in the types which means that it is also guaranteed by the compiler.

It may seem that one runs into efficiency-problems in Haskell when using algorithms which are implemented in imperative languages through mutable data which allows in-place update of memory. The seminal work of [28] 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.

For an excellent and widely used introduction to programming in Haskell we refer to [17]. Other, more exhaustive books on learning Haskell are [1, 21]. For an introduction to programming with the Lambda-Calculus we refer to [24]. For more general discussion of functional programming we refer to [13, 14, 22].

3.1 Side-Effects

One of the fundamental strengths of functional programming and Haskell is their way of dealing with side-effects in functions. A function with side-effects has observable interactions with some state outside of its explicit scope. This means that the behaviour it depends on history and that it loses its referential transparency character, which makes understanding and debugging much harder. Examples for side-effects are (amongst others): modifying a global variable, await an input from the keyboard, read or write to a file, open a connection to a server, drawing random-numbers,...

Obviously, to write real-world programs which interact with the outside-world we need side-effects. Haskell allows to indicate in the *type* of a function that it does or does *not* have side-effects. Further there are a broad range of different effect types available, to restrict the possible effects a function can have to only the required type. This is then ensured by the compiler which means that a program in which one tries to e.g. read a file in a function which only allows drawing random-numbers will fail to compile. Haskell also provides mechanisms to combine multiple effects e.g. one can define a function which can draw random-numbers and modify some global data. The most common side-effect types are:

- IO - Allows all kind of I/O related side-effects: reading/writing a file, creating threads, write to the standard output, read from the keyboard, opening network-connections, mutable references,...
- Rand - Allows to draw random-numbers.

- Reader - Allows to read from an environment.
- Writer - Allows to write to an environment.
- State - Allows to read and write an environment.

A function with side-effects has to indicate this in their type e.g. if we want to give our factorial function for debugging purposes the ability to write to the standard output, we add IO to its type: factorial :: Integer -> IO Integer. A function without any side-effect type is called *pure*. A function with a given effect-type needs to be executed with a given effect-runner which takes all necessary parameters depending on the effect and runs a given effectful function returning its return value and depending on the effect also an effect-related result. For example when running a function with a State-effect one needs to specify the initial environment which can be read and written. After running such a function with a State-effect the effect-runner returns the changed environment in addition with the return value of the function itself. Note that we cannot call functions of different effect-types from a function with another effect-type, which would violate the guarantees. Calling a *pure* function though is always allowed because it has by definition no side-effects. An effect-runner itself is a *pure* function. The exception to this is the IO effect type which does not have a runner but originates from the *main* function which is always of type IO.

Although it might seem very restrictive at first, we get a number of benefits from making the type of effects we can use explicit. First we can restrict the side-effects a function can have to a very specific type which is guaranteed at compile time. This means we can have much stronger guarantees about our program and the absence of potential errors already at compile-time which implies that we don't need test them with e.g. unit-tests. Second, because effect-runners are themselves *pure*, we can execute effectful functions in a very controlled way by making the effect-context explicit in the parameters to the effect-runner. This allows a much easier approach to isolated testing because the history of the system is made explicit.

For a technical, in-depth discussion of the concept of side-effects and how they are implemented in Haskell using Monads, we refer to the following papers: [20, 25, 40–42].

4 ADVANCED CONCEPTS

In this section we give a brief overview over advanced concepts found in functional programming.

4.1 Parallelism and Concurrency

TODO: write this section

TODO: in haskell we can distinguish between parallelism and concurrency in the types: parallelism is pure, concurrency is impure

TODO: explain STM, Problem: live locks, For a technical, in-depth discussion on Software Transactional Memory in Haskell we refer to the following papers: [11, 29].

4.2 Functional Reactive Programming

Functional Reactive Programming (FRP) is a way to implement systems with continuous and discrete time-semantics in functional programming. The central concept in FRP is the Signal Function which can be understood as a process over time which maps an input- to an output-signal. A signal in turn, can be understood as a value which varies over time. Thus, signal functions have an awareness of the passing of time by having access to a Δt which are positive time-steps with which the system is sampled. In general, signal functions can be

understood to be computations that represent processes, which have an input of a specific type, process it and output a new type. Note that this is an important building block to represent agents in functional programming: by implementing agents as signal functions allows us to implement them as processes which act continuously over time, which implies a time-driven approach to ABS. We have also applied the concept of FRP to event-driven ABS [23].

FRP provides a number of functions for expressing time-semantics, generating events and making state-changes of the system. They allow to change system behaviour in case of events, run signal functions, generate deterministic (after fixed time) and stochastic (exponential arrival rate) events and provide random-number streams.

For a technical, in-depth discussion on FRP in Haskell we refer to the following papers: [6, 12, 15, 16, 26, 31, 32, 43]

4.3 Property-Based Testing

TODO: write this section

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.

We found property-based testing particularly well suited for ABS. Although it is now available in a wide range of programming languages and paradigms, propert-based testing has its origins in Haskell [3, 4] and we argue that for that reason it really shines in pure functional programming. Property-based testing allows to formulate *functional specifications* in code which then the property-testing library (e.g. QuickCheck [3]) tries to falsify by automatically generating random test-data covering as much cases as possible. When an input is found for which the property fails, the library then reduces it to the most simple one. It is clear to see that this kind of testing is especially suited to ABS, because we can formulate specifications, meaning we describe *what* to test instead of *how* to test (again the declarative nature of functional programming shines through). Also the deductive nature of falsification in property-based testing suits very well the constructive nature of ABS.

For a technical, in-depth discussion on property-based testing in Haskell we refer to the following papers: [3, 4].

5 RELATED RESEARCH

The amount of research on using pure functional programming with Haskell in the field of ABS has been moderate so far. Most of the papers are related to the field of Multi Agent Systems and look into how agents can be specified using the belief-desire-intention paradigm [7, 19, 35].

A multi-method simulation library in Haskell called *Aivika 3* is described in the technical report [34]. It is not pure, as it uses the IO under the hood and comes with very basic features for event-driven ABS, which allows to specify simple state-based agents with timed transitions. TODO: it can do much much more, be supportive of it but make clear that it uses IO

Using functional programming for DES was discussed in [19] where the authors explicitly mention the paradigm of FRP to be very suitable to DES.

A domain-specific language for developing functional reactive agent-based simulations was presented in [38]. This language called FRABJOUS is human readable and easily understandable by domain-experts. It is not directly implemented in FRP/Haskell but is compiled to Haskell code which they claim is also readable. This supports that FRP is a suitable approach to implement ABS in Haskell. Unfortunately, the authors do not discuss

their mapping of ABS to FRP on a technical level, which would be of most interest to functional programmers.

6 A FUNCTIONAL APPROACH

Due to the fundamentally different approaches of functional programming (FP) an ABS needs to be implemented fundamentally different as well compared to established object-oriented (OO) approaches. We face the following challenges:

(1) How can we represent an Agent, its internal state and its interface?
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 FP thus we need to find a different approach to represent the agents actions and to encapsulate its state. 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 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. In the established OO approach, agents have a well-defined interface through their public methods through which one can interact with the agent and query information about it. Again we don't have this in FP as we don't have objects and globally mutable state.

(2) How can we implement pro-activity 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 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.

(3) 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 FP we struggle with this for the same reasons we face when deciding how to represent an Agent, its state and proactivity.

(4) How can we implement agent-agent and agent-environment interactions?
In the classic OO approach Agents can directly invoke other Agents methods which makes direct Agent interaction straight forward. Again this is obviously not possible in FP as we don't have objects with methods and mutable state inside. 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 FP as we don't have objects and globally mutable state.

(5) 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 FP we cannot iteratively mutate a global state.

The fundamental building blocks to solve these problems are *recursion* and *continuations*. In recursion a function is defined in terms of itself: in the process of computing the output it *might* call itself with changed input data. Continuations in turn allow to encapsulate the

execution state of a program including local variables and pick up computation from that point later on. We present an example for continuations and recursions:

```
newtype Cont a = Cont (a -> (a, Cont a))

adder :: Int -> Cont Int
adder x = Cont (\x' -> (x + x', adder (x + x')))

runCont :: Int
        -> Cont Int
        -> IO ()
runCont 0 _ = return ()
runCont n (Cont cont) = do
  let (x, cont') = cont 1
  print x
  runCont (n-1) cont'

test :: IO ()
test = runCont 100 (adder 0)
```

Fortunately FRP (see Section 4.2) provides us already with a suitable abstraction, the signal function, which are built on *recursion* and *continuations*. Using signal functions and FRP allows us to solve the presented problems.

6.1 Agent representation, internal state and interface

Continuation: It is apparent that functional programming supports very strong encapsulation of local state which is not accessible and mutable from outside. This makes testing in some respect easier, in some harder.

6.2 Agent pro-activity

TODO: time-driven vs. event-driven 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.

6.3 Agent-Agent interactions

Agent-transactions are necessary when an arbitrary number of interactions between two agents need to happen instantaneously without time-lag. The use-case for this are price negotiations between multiple agents where each pair of agents needs to come to an agreement in the same time-step [8]. In object-oriented programming, the concept of synchronous communication between agents is trivially implemented directly with method calls.

6.4 Environment representation and Agent-Environment interactions

TODO: basically we follow the same approach as in agent representation, depending on what kind of environments we need

Generally, there exist four different types of environments in agent-based simulation:

(1) Passive read-only - implemented in the previous steps, where the environment never changes and is passed as static information, e.g. list of neighbours, to each agent.
(2) Passive read/write - implemented in this step. The environment itself is not modelled as an active process but just as shared data which can be accessed and manipulated by the agents.

(3) Active read-only - can be implemented by adding an environment agent which broadcasts changes in the environment to all agents using the data-flow mechanism.
(4) Active read/write - can be implemented as in this step plus adding an environment agent which reads/writes the environment e.g. regrowing some resources.

6.5 Stepping the simulation

TODO: time-driven vs. event-driven This allows to implement feedback by recursively stepping a simulation and feeding in the output of the last step into the next step.

TODO: depending on the time-semantics of the model we approach it different. TODO: SIR maps nicely to continuous time-semantics and state-transitions provided by FRP, property-testing can be used to directly express parts of the SD specification TODO: Sugarscape: no need for continuous time-semantics as agents act in all time-step, main difficulty: synchronous agent-interactions, property-based testing: can we express hypotheses?

7 MULTI-METHOD SIMULATION

7.1 System Dynamics

7.2 Discrete Event Simulation

8 DISCUSSION

TODO: show why the claim of the introduction holds up

The restrictions functional programming imposes, directly removes serious sources of bugs which leads to simulation which is more likely to be correct. These restrictions force us to solve the fundamental concepts in ABS implementation differently. Note that we could fall back to using IO throughout all the simulation in which case we have access to mutable references but then we lose important compile-time guarantees and introduce those serious sources of bugs we want to get rid of - also testing becomes more complicated and not as strong any more because we cannot guarantee at compile time that no random IO stuff is happening within the agents. Also note that obviously no one would do random IO stuff in an agent (e.g. read from a file, open connection to server...) but one must not understimate the value of guaranteeing its absence at compile-time.

9 CONCLUSIONS

- being explicit and polymorph about side-effects: can have 'pure' (no side-effects except state), 'random' (can draw random-numbers), 'IO' (all bets are off), STM (concurrency) agents - hybrid between SD and ABS due to continuous time AND parallel dataFlow (parallel update-strategy)

Our approach is radically different from traditional approaches in the ABS community. First it builds on the already quite powerful FRP paradigm. Second, due to our continuous time approach, it forces one to think properly of time-semantics of the model and how small Δt should be. Third it requires one to think about agent interactions in a new way instead of being just method-calls.

Because no part of the simulation runs in the IO Monad and we do not use unsafePerformIO we can rule out a serious class of bugs caused by implicit data-dependencies and side-effects which can occur in traditional imperative implementations.

Also we can statically guarantee the reproducibility of the simulation, which means that repeated runs with the same initial conditions are guaranteed to result in the same dynamics. Although we allow side-effects within agents, we restrict them to only the Random and State

Monad in a controlled, deterministic way and never use the IO Monad which guarantees the absence of non-deterministic side effects within the agents and other parts of the simulation.

Determinism is also ensured by fixing the Δt and not making it dependent on the performance of e.g. a rendering-loop or other system-dependent sources of non-determinism as described by [33]. Also by using FRP we gain all the benefits from it and can use research on testing, debugging and exploring FRP systems [30, 33].

Issues

Reasoning about space-leaks due to lazyness.

Currently, the performance of the system is not comparable to imperative implementations but our research was not focusing on this aspect. We leave the investigation and optimization of the performance aspect of our approach for further research.

In our pure functional approach, agent identity is not as clear as in traditional object-oriented programming, where an agent can be hidden behind a polymorphic interface which is much more abstract than in our approach. Also the identity of an agent is much clearer in object-oriented programming due to the concept of object-identity and the encapsulation of data and methods.

We can conclude that the main difficulty of a pure functional approach evolves around the communication and interaction between agents, which is a direct consequence of the issue with agent identity. Agent interaction is straight-forward in object-oriented programming, where it is achieved using method-calls mutating the internal state of the agent, but that comes at the cost of a new class of bugs due to implicit data flow. In pure functional programming these data flows are explicit but our current approach of feeding back the states of all agents as inputs is not very general. We have added further mechanisms of agent interaction which we had to omit due to lack of space.

10 FURTHER RESEARCH

- we have also implemented an event-driven approach - can we do DES? e.g. single queue with multiple servers? also specialist vs. generalist - can we do SD?

ACKNOWLEDGMENTS

The authors would like to thank

REFERENCES

[1] Christopher Allen and Julie Moronuki. 2016. *Haskell Programming from First Principles*. Allen and Moronuki Publishing. Google-Books-ID: 5FaXDAEACAAJ.

[2] Alonzo Church. 1936. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics* 58, 2 (April 1936), 345–363. https://doi.org/10.2307/2371045

[3] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[4] Koen Claessen and John Hughes. 2002. Testing Monadic Code with QuickCheck. *SIGPLAN Not.* 37, 12 (Dec. 2002), 47–59. https://doi.org/10.1145/636517.636527

[5] Andreu Mas Colell. 1995. *Microeconomic Theory*. Oxford University Press. Google-Books-ID: dFS2AQAACAAJ.

[6] Antony Courtney, Henrik Nilsson, and John Peterson. 2003. The Yampa Arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell (Haskell '03)*. ACM, New York, NY, USA, 7–18. https://doi.org/10.1145/871895.871897

[7] Tanja De Jong. 2014. *Suitability of Haskell for Multi-Agent Systems*. Technical Report. University of Twente.

[8] Joshua M. Epstein and Robert Axtell. 1996. *Growing Artificial Societies: Social Science from the Bottom Up.* The Brookings Institution, Washington, DC, USA.

[9] Herbert Gintis. 2006. The Emergence of a Price System from Decentralized Bilateral Exchange. *Contributions in Theoretical Economics* 6, 1 (2006), 1–15. https://doi.org/10.2202/1534-5971.1302

[10] Jason Gregory. 2018. *Game Engine Architecture, Third Edition.* Taylor & Francis.

[11] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05)*. ACM, New York, NY, USA, 48–60. https://doi.org/10.1145/1065944.1065952

[12] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2003. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming*, Johan Jeuring and Simon L. Peyton Jones (Eds.). Number 2638 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 159–187. https://doi.org/10.1007/978-3-540-44833-4_6

[13] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*. ACM, New York, NY, USA, 12–1–12–55. https://doi.org/10.1145/1238844.1238856

[14] J. Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (April 1989), 98–107. https://doi.org/10.1093/comjnl/32.2.98

[15] John Hughes. 2000. Generalising Monads to Arrows. *Sci. Comput. Program.* 37, 1-3 (May 2000), 67–111. https://doi.org/10.1016/S0167-6423(99)00023-4

[16] John Hughes. 2005. Programming with Arrows. In *Proceedings of the 5th International Conference on Advanced Functional Programming (AFP'04)*. Springer-Verlag, Berlin, Heidelberg, 73–129. https://doi.org/10.1007/11546382_2

[17] Graham Hutton. 2016. *Programming in Haskell.* Cambridge University Press. Google-Books-ID: 1xHPDAAAQBAJ.

[18] Cezar Ionescu and Patrik Jansson. 2012. Dependently-Typed Programming in Scientific Computing. In *Implementation and Application of Functional Languages (Lecture Notes in Computer Science)*, Ralf Hinze (Ed.). Springer Berlin Heidelberg, 140–156. https://doi.org/10.1007/978-3-642-41582-1_9

[19] Peter Jankovic and Ondrej Such. 2007. *Functional Programming and Discrete Simulation.* Technical Report.

[20] Simon Peyton Jones. 2002. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction.* Press, 47–96.

[21] Miran Lipovaca. 2011. *Learn You a Haskell for Great Good!: A Beginner's Guide* (1 edition ed.). No Starch Press, San Francisco, CA.

[22] Bruce J. MacLennan. 1990. *Functional Programming: Practice and Theory.* Addison-Wesley. Google-Books-ID: JqhQAAAAMAAJ.

[23] Ruth Meyer. 2014. Event-Driven Multi-agent Simulation. In *Multi-Agent-Based Simulation XV (Lecture Notes in Computer Science)*. Springer, Cham, 3–16. https://doi.org/10.1007/978-3-319-14627-0_1

[24] Greg Michaelson. 2011. *An Introduction to Functional Programming Through Lambda Calculus.* Courier Corporation. Google-Books-ID: gKvwPtvsSjsC.

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

[26] Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 51–64. https://doi.org/10.1145/581690.581695

[27] Michael J. North and Charles M. Macal. 2007. *Managing Business Complexity: Discovering Strategic Solutions with Agent-Based Modeling and Simulation.* Oxford University Press, USA. Google-Books-ID: gRATDAAAQBAJ.

[28] Chris Okasaki. 1999. *Purely Functional Data Structures.* Cambridge University Press, New York, NY, USA.

[29] Bryan O'Sullivan, John Goerzen, and Don Stewart. 2008. *Real World Haskell* (1st ed.). O'Reilly Media, Inc.

[30] Ivan Perez. 2017. Back to the Future: Time Travel in FRP. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. ACM, New York, NY, USA, 105–116. https://doi.org/10.1145/3122955.3122957

[31] Ivan Perez. 2017. *Extensible and Robust Functional Reactive Programming*. Doctoral Thesis. University Of Nottingham, Nottingham.

[32] Ivan Perez, Manuel Baerenz, and Henrik Nilsson. 2016. Functional Reactive Programming, Refactored. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, New York, NY, USA, 33–44. https://doi.org/10.1145/2976002.2976010

[33] Ivan Perez and Henrik Nilsson. 2017. Testing and Debugging Functional Reactive Programming. *Proc. ACM Program. Lang.* 1, ICFP (Aug. 2017), 2:1–2:27. https://doi.org/10.1145/3110246

[34] David Sorokin. 2015. *Aivika 3: Creating a Simulation Library based on Functional Programming*.

[35] Martin Sulzmann and Edmund Lam. 2007. *Specifying and Controlling Agents in Haskell*. Technical Report.

[36] Tim Sweeney. 2006. The Next Mainstream Programming Language: A Game Developer's Perspective. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, New York, NY, USA, 269–269. https://doi.org/10.1145/1111037.1111061

[37] A. M. Turing. 1937. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* s2-42, 1 (1937), 230–265. https://doi.org/10.1112/plms/s2-42.1.230

[38] Ivan Vendrov, Christopher Dutchyn, and Nathaniel D. Osgood. 2014. Frabjous: A Declarative Domain-Specific Language for Agent-Based Modeling. In *Social Computing, Behavioral-Cultural Modeling and Prediction*, William G. Kennedy, Nitin Agarwal, and Shanchieh Jay Yang (Eds.). Number 8393 in Lecture Notes in Computer Science. Springer International Publishing, 385–392. https://doi.org/10.1007/978-3-319-05579-4_47

[39] V. Vipindeep and Pankaj Jalote. 2005. *List of Common Bugs and Programming Practices to avoid them*. Technical Report. Indian Institute of Technology, Kanpur.

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

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

[42] Philip Wadler. 1997. How to Declare an Imperative. *ACM Comput. Surv.* 29, 3 (Sept. 1997), 240–263. https://doi.org/10.1145/262009.262011

[43] Zhanyong Wan and Paul Hudak. 2000. Functional Reactive Programming from First Principles. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 242–252. https://doi.org/10.1145/349299.349331

Received May 2018