The Agent's new Cloths

Towards functional programming in Agent-Based Simulation

JONATHAN THALER, PEER-OLAF SIEBERS, and THORSTEN ALTENKIRCH, University of Nottingham, United Kingdom

TODO: currently writing for TOMACS - Papers should not be longer than 25 pages - Also, it may be appropriate to place content in supplementary electronic material to accompany the submission, e.g. information and software required for reproducing the results presented in the paper. - If the paper is accepted, paper and supplementary material will be made available in the ACM Digital Library.

TODO: select journals - ACM Transactions on Modeling and Computer Simulation (TOMACS): https://tomacs.acm.org/ - Journal of Simulation: https://www.tandfonline.com/toc/tjsm20/current (https://www.tandfonline.com/doi/10.1057/jos.2010 - JASSS: http://jasss.soc.surrey.ac.uk/JASSS.html

TODO: although we approach it from a high level, we have to present a bit of code at various points otherwise its too vague.

IMPORTANT: don't over-exaggerate: instead of harder, say hard. instead of easier, say easy

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, Peer-Olaf Siebers, and Thorsten Altenkirch. 2019. The Agent's new Cloths: Towards functional programming in Agent-Based Simulation. 1, 1 (July 2019), 16 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 [9] 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* [31] 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 [14] to become available when implementing ABS functionally.

Authors' address: Jonathan Thaler, jonathan.thaler@nottingham.ac.uk; Peer-Olaf Siebers, peer-olaf.siebers@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

We claim that the community needs functional programming in ABS because of its *scientific computing* nature where results need to be reproducible and correct while simulations should be able to massively scale-up as well. The established object-oriented approaches need considerably high effort and might even fail to deliver these objectives due to its conceptually different approach to computing. In contrast, we claim that by using functional programming for implementing ABS it is easy to add parallelism and concurrency, the resulting simulations are easy to test and verify, guaranteed to be reproducible already at compile-time, have few potential sources of bugs and are ultimately very likely to be correct.

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. Still, it is necessary to present a bit of code to make things more clear. We always explain the code on a conceptual level, not necessarily being very precise and technical correct in terms of functional programming but here we aim for readability and clarity instead of technical perfection and detail. 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. Also this paper can be seen as a mini review paper on this new topic as we cite the relevant literature to get into it.
- 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.

2 RELATED RESEARCH

TODO: re-read them all and describe them more constructively

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, 20, 41].

A multi-method simulation library in Haskell called *Aivika 3* is described in the technical report [40]. 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 [20] 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 [45]. 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.

3 THE ESTABLISHED APPROACH TO ABS

TODO: we need citiations here to support our claims!

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.

It is no coincidence that JavaScript, the most widely used language for programming client-side web-applications, originally a completely dynamically typed language like Python, got additions for type-checking developed by the industry through TypeScript. This is an indicator that the industry acknowledges types as something important as they allow to rule out certain classes of bugs at run-time and express guarantees already at compile-time. We expect similar things to happen with Python as its popularity is surging and more and more people become aware of that problem. Summarizing, due to the highly dynamic nature of the type-system and imperative nature, run-time errors and bugs are possible both in Python and Java which absence must be guaranteed by exhaustive testing.

The problem of correctness in agent-based simulations became more apparent in the work of Ionescu et al [19] which tried to replicate the work of Gintis [10]. 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. [19] 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 [42], 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 [11]. 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: general introduction

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: [46]

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

4 CONCEPTS OF FUNCTIONAL PROGRAMMING

The reason why functional programming is called *functional* is because because it makes functions the main concept of programming, promoting them to first-class citizens as we will describe later on. Its roots 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 [44]. 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 [14] 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 [14], is applicable to a number of real-world problems [33] 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.

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

(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.
(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 [32] 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 [18]. Other, more exhaustive books on learning Haskell are [1, 22]. For an introduction to programming with the Lambda-Calculus we refer to [28]. For more general discussion of functional programming we refer to [14, 15, 25].

4.1 Lazy evaluation, Higher Order Functions and Currying

TODO:

4.2 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: [21, 29, 47–49].

4.3 Parallelism and Concurrency

TODO: write this section

Also Haskell makes a very clear distinction between parallelism and concurrency. Parallelism is always deterministic and thus pure without side-effects because although parallel code runs concurrently, it does by definition not interact with data of other threads. This can be indicated through types: we can run pure functions in parallel because for them it doesn't matter in which order they are executed, the result will always be the same due to the concept of referential transparency. Concurrency is potentially non-deterministic because of non-deterministic interactions of concurrently running threads through shared data. Although data in functional programming is immutable, Haskell provides primitives which allow to share immutable data between threads. Accessing these primitives is but only possible from within an IO or STM context which means that when we are using concurrency in our program, the types of our functions change from pure to either IO or STM effect context.

Spawning thousands of threads in Haskell is no problem and has very low memory footprint because they are lightweight user-space threads, managed by the Haskell Runtime System which maps them to physical operating-system threads.

TODO: in haskell we can distinguish between parallelism and concurrency in the types: parallelism is pure, concurrency is impure TODO: parallelism for free because all isolated e.g. running multiple replications or parameter-variations

TODO: For a technical, in-depth discussion on parallelism and concurrency in Haskell we refer to the excellent book [26].

TODO: explain STM, Problem: live locks, For a technical, in-depth discussion on Software Transactional Memory in Haskell we refer to the following papers: [8, 12, 33, 38]. TODO: need much more papers on STM, parallelism and concurrency

4.4 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 [27].

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.

TODO: libraries Yampa and Dunai

For a technical, in-depth discussion on FRP in Haskell we refer to the following papers: [6, 13, 16, 17, 30, 35, 36, 50]

4.5 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 A FUNCTIONAL APPROACH TO ABS

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 underestimate the value of guaranteeing its absence at compile-time. Thus, 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 local state and its interface? How can be make it proactive?
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 established 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. In the established 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.
(2) How can we implement agent-agent interactions?
In the established 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 established 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.
(3) How can we represent an environment and its various types? How can we implement agent-environment interactions
In the established 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 step the simulation?
In the established 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.

5.1 Agent representation, local state, interface and pro-activity

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. TODO: explain

```
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)
```

From the continuation example it becomes apparent that we can encapsulate local state which which is not accessible and mutable from outside but only through explicit inputs and outputs to the continuation. The source of pro-activity in ABS comes always from observing the time - when an agent can observe the flow of time, it can become pro-active and initiate actions on its own without external stimuli like events [43]. Thus we need to make the flow of time available to our agents as well.

FRP (see Section 4.4) provides us with an interesting abstraction for a flow of time, the signal function, which are built on *recursion* and *continuations*. A signal function can be understood as a *process over time* which maps an input- to an output-signal. A signal 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 Δt which are positive time-steps with which the system is sampled. Also some FRP implementations allow to execute signal functions within a context which can have side-effects [36] of which we can make use as well (see below).

$$Signal\ \alpha \approx Time \longrightarrow \alpha$$
$$SF\ \alpha\ \beta \approx Signal\ \alpha \longrightarrow Signal\ \beta$$

Using signal functions and FRP allows us to solve the presented problems, thus we make an Agent a signal function. Our agent-interface is defined in terms of the input $Signal\ \alpha$ and output $Signal\ \beta$ and the type of side-effects the context allows. Further it allows us to encapsulate local state on a very strong level: there is now way to access or mutate locale state outside of the control of an agent, it all has to go through the inputs and outputs and running the signal function. Pro-activity is not a problem as well as signal functions have an awareness of time which can be used to e.g. emit events *after* a given time-out.

We present a short code-example of an infected agent of the agent-based SIR model [23] which recovers after a given time. The first line with the double semi-colons (::) defines the type of a function. We see that *infectedAgent* is a signal function (SF) which has as input the empty tuple (can be seen as void / no input) and outputs the SIR state it is currently in. Also this signal-function is pure and does not run within a side-effect context. By looking at the types we see no explicit Δt as input (it is hidden in the signal function and FRP implementation) thus there is no way to access it explicitly, meaning we removed a potential source of bugs.

The infected agent behaves as infected until the recovery-event happens - from that moment on it will behave as a recovered agent - which is implemented using the *switch* function provided by FRP. The *infected* function returns a tuple with the Event in addition to the SIR state which indicates if the recovery-event has happened or not. If it has, then the *switch* function will detect this and switch into *recoveredAgent*. While infected, the agent 'waits' for the recovery-event which is generated using the *occasionally* function, provided by FRP. It generates on average an event after *illnessDuration*, meaning it generates stochastic events from an exponential distribution. Depending whether the event has occurred, the infected agents outputs Infected or Recovered.

Note that *occasionally* is a stochastic function which means it makes use of a random-number stream, which is passed to *infectedAgent* in its first argument *RandomGen g => g*.

```
-- an agent in the SIR model is either Susceptible, Infected or Recovered
data SIRState = Susceptible | Infected | Recovered

infectedAgent :: RandomGen g => g -> SF () SIRState
infectedAgent g
    -- behave as infected until recovery-event, then behave as
    -- recoveredAgent from that moment on
    = switch infected (const recoveredAgent)
  where
    infected :: SF () (SIRState, Event ())
    infected = proc _ -> do
      -- awaiting the recovery-event
      recEvt <- occasionally g illnessDuration () -< ()
      -- if event occurred a is Recovered, otherwise Infected
      let a = event Infected (const Recovered) recEvt
      -- return the state and event
      returnA -< (a, recEvt)

-- a recovered agent stays Recovered forever
recoveredAgent :: SF () SIRState
recoveredAgent = arr (const Recovered)
```

Note that this approach to ABS is inherently time-driven. This means that depending on the time-semantics of the model, we need to select the right time-deltas by which to sample the simulation. If it is too large we might not arrive at the correct solution, if it is too small, we run into performance problems. An alternative is to follow an event-driven approach [27], where agents schedule events in a Discrete Event Simulation fashion. To implement such an approach is possible using signal functions and FRP as well by running within a State context in which one manages an event queue. The simulation stepping (see below) then advances the simulation through processing the events instead of time-deltas. Although the event-driven approach can emulate the time-driven approach and is thus more general, it might be more natural to implement the simulation in a time-driven way due to the model semantics fit more natural to it.

5.2 Agent-Agent interactions

Agent-agent interactions are trivial in object-orientation: one either makes a direct method call or send an event, mutating the internal state of the receiving agent. In functional programming we need to come up with alternatives because neither method-calls nor globally mutable state is available.

A simple solution is to implement *asynchronous* interactions, which can be seen as a message sent to the target agent which will arrive in the next time step, passed in through the signal function input. If the receiving agent doesn't handle the event, it will be lost if we are in a pure context because there is no concept of a persistent message-box which would require mutable data. In a effectful context e.g. STM or State, we could implement stateful message-boxes. Whether an asynchronous approach is suitable, depends entirely on model semantics - it might work for some models or parts of a model, but not for others.

The alternative are *synchronous* interactions which are necessary when an arbitrary number of interactions between two agents need to happen instantaneously without any time-steps in between. 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 [9]. In object-oriented programming, the concept of synchronous communication between agents is trivially implemented directly with method calls but it can get tricky to get right in an functional programming setting. The only option one has, is to dynamically find the target agents signal function and run it within the source agent. This would imply some effectful context which allows read/write to all signal functions in the system: we need to read it to find the target and write it to put the continuation back in because it has locally encapsulated state. This is active research we conduct at the moment and we leave this for further research as it is out of the scope of this paper.

TODO: implement synchronous agent interaction

TODO: discuss macals 4 classifications of his paper [24]

5.3 Environment representation and Agent-Environment interactions

Depending on which kind of environment we are using we have different approaches on how to solve these problems. We distinguish between four different environment types:

(1) Passive read-only e.g. a static neighbour network - The environment is passed as additional input to each agent.
(2) Passive read/write e.g. a shared 2D grid like in the schelling model [39] - The environment is shared through an effectful State context which can be accessed and manipulated by the agents.
(3) Active read-only e.g. ? - The environment is made a signal function too which broadcasts asynchronous messages about changes in the environment to all agents.
(4) Active read/write e.g. Sugarscapes environment in which agents move around and harvest resources but where the environment regrows them - The environment is made a signal function which acts on a shared state which is made available to the environment *and* the agents using an effectful State context.

5.4 Stepping the simulation

An FRP library generally provides functions to either run signal functions with or without any effectful context. Further they might also provide a looping function which runs within the IO context to e.g. continuously render outputs to a window. All of it is built on the concept of recursion and continuations as we have introduced earlier which allows to feed the output of the current step into the next one, generating a closed feedback-loop.

6 DISCUSSION

TODO: maybe we find a catchier title of this section e.g. "Functional programming to the rescue (?)"

After having presented *how* to implement ABS in the functional programming, we now discuss *why* it is of benefit of doing so. We re-visit the claims made in the introduction, discuss each of them, show why they hold and connect them to ABS.

6.1 Easy to add parallelism and concurrency

The main problems of parallelism and concurrency in imperative programming languages stem from mutable data which require locks to avoid race-conditions. Further in imperative languages the distinction between a parallel (no interactions between threads) and a concurrent (interaction between threads explicitly happening) program is not always clear as it is difficult to guarantee that threads don't mutate shared data. As already pointed out, in functional programming data is immutable and it makes a very clear distinction between parallelism and concurrency. This means that we can easily apply parallelism and concurrency in our ABS approach.

An example of parallelism is running multiple replications of a pure simulation because they can each be run in isolation from each other without guaranteed no interference between each of them. Another example of parallelism is running a collection of pure signal functions in parallel.

Scaling up of ABS to massively large-scale is possible using Software Transactional Memory (STM). In this case each agent runs in its own thread and the signal function is executed within an STM effect context which makes transactional variables and transactional persistent message-boxes available. If additional effect context are needed on top of STM e.g. Random Number Streams, they can be added as well but they are always local to the respective thread, guaranteeing isolation of effects between the agents. Of course, this isolation does not apply to the STM effect context where we share mutable data through STM variables or queues - the building blocks for concurrency.

Unfortunately when using concurrency one loses the compile-time guarantee of reproducibility of the simulation: same initial conditions don't lead to the same results any more because of non-deterministic influence of concurrency. Still, by restricting the possible effects to STM and not to the unrestricted IO, we can still guarantee that the non-deterministic influence will only stem from STM and no other IO actions e.g. we can guarantee that the agents wont create additional threads, communicate through shared mutable variables, read/write files,...

We have prototyped the use of STM for massively scaling-up concurrent ABS for the agent-based SIR [23] and Sugarscape [9] model. Both show good results and demonstrate that mapping ABS to STM for concurrency is a highly promising approach to be researched more in-depth in future research.

6.2 Easy to test and verify

One of the main strengths of functional programming is its composability. Pure functions can be easily composed if their input/output types match, further effectful functions can be easily composed as well due to the explicit way side-effects are handled. Also the ability to run effectful functions isolated with given initial parameters makes effectful functions highly composable. All this allows a much straight-forward and easier approach to testing because we can isolate tests extremely well.

Also this make verification easier. TODO: why?

By utilizing property-based testing we can even leverage this further. TODO:

For a more in-depth and technical discussion of testing of FRP applications, we refer to [34, 37].

6.3 Guaranteed to be reproducible

When we restrict the agents to operate either purely or in deterministic effect contexts (ruling out IO or STM) we can guarantee that the simulation will always result in the same dynamics given same initial starting conditions - already at compile-time. This is a very strong and powerful guarantee one can make as it is highly important for scientific computing in general and simulations in particular. One can argue that one would never make the mistake of implementing non-deterministic agents (e.g. reading/writing a file) when wants deterministic behaviour but in Haskell we can ensure this through the compiler.

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 [37].

6.4 Few potential sources of bugs and Very likely to be correct

The type system of Haskell and the nature of functional programming removes many pr - immutable data: no hidden data-dependencies - static type system: much less run-time errors because no dynamic types TODO: refer to bugs in ABS

6.5 Problems

Despite the dramatic reduction of potential sources of bugs, in Haskell we potentially can have run-time errors if we us partial functions. A partial function is a function which is not defined for all input-cases and should be avoided under all circumstances. Still parts of the Haskell basic library builds on partial functions. Calling a partial function with an argument for which the function is not defined results in a run-time error. Haskell provides many compile-time options to avoid such partial functions but in the end it is up to the programmer to follow this more or less strictly. The functional language Idris, which has a more sophisticated type-system allows to enforce this property by the compiler - we leave this for further research.

Although lazy evaluation can be seen as a major strength of Haskell, because it allows to separate consumption from production, it is also a serious weakness as it makes reasoning about space-leaks very hard even for experienced programmers. This can be alleviated by forcing strictness of some parts and by the excellent profiling tools freely available but the problem is still there and especially difficult for beginners.

Performance is another issue. Although we haven't conducted proper performance measurements it is clear that the performance of functional ABS is currently not comparable to imperative implementations. We don't see this as a serious problem for now because of the benefits we get from functional programming, including its convenient abilities to parallelise and go concurrent. Also performance can be potentially improved with future research.

Agent identity is not as clear as in established object-oriented approaches, 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 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 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.

7 CONCLUSIONS

- its very much about types which guide and restrict us

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.

- we are also very well aware that choosing a programming language does not solely depends on its features but we hope that this paper might inspire implementors who use established object-oriented approaches to use functional concepts.

8 FURTHER RESEARCH

- STM - multi-method simulation - can we do DES? e.g. single queue with multiple servers? also specialist vs. generalist use MSFs and event-queue - can we do SD? towards paper: sd is nearly correct by construction - emphasis that aivika is multi-method, gis, distributed. - idris with dependent type system arrive at potentially correct-by-construction - other languages: Scala seems to be very promising and also supports very lightweight message-passing concurrency but has side-effects, erlang has no side-effects and is more centered around lightweight message-passing concurrency with distribution support, clojure / scheme are LISP dialects (lisp the first functional programming language) but also are implicitly effectful but support homoiconicity, F# ?, ...

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] Anthony Discolo, Tim Harris, Simon Marlow, Simon Peyton Jones, and Satnam Singh. 2006. Lock Free Data Structures Using STM in Haskell. In *Proceedings of the 8th International Conference on Functional and Logic Programming (FLOPS'06).* Springer-Verlag, Berlin, Heidelberg, 65–80. https://doi.org/10.1007/11737414_6

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

[10] 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

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

[12] 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

[13] 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

[14] 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

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

[16] 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

[17] 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

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

[19] 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

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

[21] 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.

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

[23] Charles M. Macal. 2010. To Agent-based Simulation from System Dynamics. In *Proceedings of the Winter Simulation Conference (WSC '10)*. Winter Simulation Conference, Baltimore, Maryland, 371–382. http://dl.acm.org/citation.cfm?id=2433508.2433551

[24] C. M. Macal. 2016. Everything you need to know about agent-based modelling and simulation. *Journal of Simulation* 10, 2 (May 2016), 144–156. https://doi.org/10.1057/jos.2016.7

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

[26] Simon Marlow. 2013. *Parallel and Concurrent Programming in Haskell*. O'Reilly. Google-Books-ID: k0W6AQAACAAJ.

[27] 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

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

[29] 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

[30] 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

[31] 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.

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

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

[34] 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

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

[36] 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

[37] 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

[38] Cristian Perfumo, Nehir SÃűnmez, Srdjan Stipic, Osman Unsal, AdriÃąn Cristal, Tim Harris, and Mateo Valero. 2008. The Limits of Software Transactional Memory (STM): Dissecting Haskell STM Applications on a Many-core Environment. In *Proceedings of the 5th Conference on Computing Frontiers (CF '08)*. ACM, New York, NY, USA, 67–78. https://doi.org/10.1145/1366230.1366241

[39] Thomas Schelling. 1971. Dynamic models of segregation. *Journal of Mathematical Sociology* 1 (1971).

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

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

[42] 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

[43] Jonathan Thaler and Peer-Olaf Siebers. 2017. The Art Of Iterating: Update-Strategies in Agent-Based Simulation. Dublin.

[44] 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

[45] 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

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

[47] 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

[48] 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

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

[50] 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