Pure functional epidemics

An Agent-Based Approach

Jonathan Thaler
Thorsten Altenkirch
Peer-Olaf Siebers
jonathan.thaler@nottingham.ac.uk
thorsten.altenkirch@nottingham.ac.uk
peer-olaf.siebers@nottingham.ac.uk
University of Nottingham
Nottingham, United Kingdom

Abstract

Agent-Based Simulation (ABS) is a methodology in which a system is simulated in a bottom-up approach by modelling the micro interactions of its constituting parts, called agents, out of which the global system behaviour emerges.

So far mainly object-oriented techniques and languages have been used in ABS. Using the SIR model of epidemiology, which allows to simulate the spreading of an infectious disease through a population, we show how to use Functional Reactive Programming to implement ABS. With our approach we can guarantee the reproducibility of the simulation already at compile time, which is not possible with traditional object-oriented languages. Also, we claim that this representation is conceptually very clean and opens the way to formally reason about ABS. Further we verify the correctness of our implementation through property-testing using QuickCheck and white-box code verification through informal reasoning. Unfortunately the performance of our approach is far behind traditional OO approaches but this is not the main focus of our research and we outline alternatives to alleviate this problem in further research.

Keywords Functional Reactive Programming, Monadic Stream Functions, Agent-Based Simulation

ACM Reference Format:
Jonathan Thaler, Thorsten Altenkirch, and Peer-Olaf Siebers. 2018. Pure functional epidemics: An Agent-Based Approach. In *Proceedings of Haskell Symposium (HS18).* ACM, New York, NY, USA, 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 [7] 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* [17] which still holds up today.

In this paper we fundamentally challenge this metaphor and explore ways of approaching ABS in a pure functional way using Haskell. By doing this we expect to leverage the benefits of pure functional programming [9]: higher expressivity through declarative code, being polymorph and explicit about side-effects through monads, more robust and less susceptible for bugs due to explicit data flow and lack of implicit side-effects.

As use case we introduce the simple SIR model of epidemiology with which one can simulate epidemics, that is the spreading of an infectious disease through a population, in a realistic way.

Over the course of four steps, we derive all necessary concepts required for a full agent-based implementation. We start from a very simple solution running in the Random Monad which has all general concepts already there and then refine it in various ways, making the transition to Functional Reactive Programming (FRP) [30] and to Monadic Stream Functions (MSF) [20].

The aim of this paper is to show how ABS can be done in *pure* Haskell and what the benefits and drawbacks are. By doing this we give the reader a good understanding of what ABS is, what the challenges are when implementing it and how we solve these in our approach.

The contributions of this paper are:

- To the best of our knowledge, we are the first to *systematically* introduce the concepts of ABS to the *pure* functional programming paradigm in a step-by-step approach. It is also the first paper to show how to apply Arrowized FRP to ABS on a technical level, presenting a new field of application to FRP.
- Our approach shows how robustness can be achieved through purity which guarantees reproducibility at compile time, something not possible with traditional object-oriented approaches.

HS18, 2018, 09
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- The result of using Arrowized FRP is a conceptually clean approach to ABS. It allows expressing continuous time-semantics in very clean, compositional and declarative way, without having to deal with low-level details related to the progress of time.
- Using property-testing with QuickCheck in combination with white-box verification (informal reasoning about the code) we can ensure the correctness of our implementation to a very high degree.

Section 7 discusses related work. In section 2 we introduce functional reactive programming, arrowized programming and monadic stream functions, because our approach builds heavily on these concepts. Section 3 defines agent-based simulation. In section 4 we introduce the SIR model of epidemiology as an example model to explain the concepts of ABS. The heart of the paper is section 5 in which we derive the concepts of a pure functional approach to ABS in four steps, using the SIR model. Finally, we draw conclusions and discuss issues in section 8 and point to further research in section 9.

2 Background

2.1 Functional Reactive Programming

Functional Reactive Programming (FRP) is a way to implement systems with continuous and discrete time-semantics in pure functional languages. There are many different approaches and implementations but in our approach we use *Arrowized* FRP [10], [11] as implemented in the library Yampa [8], [5], [16].

The central concept in arrowized FRP is the Signal Function (SF) which 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.

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

Yampa provides a number of combinators for expressing time-semantics, events and state-changes of the system. They allow to change system behaviour in case of events, run signal functions and generate stochastic events and random-number streams. We shortly discuss the relevant combinators and concepts we use throughout the paper. For a more in-depth discussion we refer to [8], [5], [16].

Event An event in FRP is an occurrence at a specific point in time which has no duration e.g. the the recovery of an infected agent. Yampa represents events through the *Event* type which is programmatically equivalent to the *Maybe* type.

Dynamic behaviour To change the behaviour of a signal function at an occurrence of an event during run-time, the combinator *switch :: SF a (b, Event c) -> (c -> SF a b) -> SF a b* is provided. It takes a signal function which is run until it generates an event. When this event occurs, the function in the second argument is evaluated, which receives the data of the event and has to return the new signal function which will then replace the previous one.

Randomness In ABS one often needs to generate stochastic events which occur based on e.g. an exponential distribution. Yampa provides the combinator *occasionally :: RandomGen g => g -> Time -> b -> SF a (Event b)* for this. It takes a random-number generator, a rate and a value the stochastic event will carry. It generates events on average with the given rate. Note that at most one event will be generated and no 'backlog' is kept. This means that when this function is not sampled with a sufficiently high frequency, depending on the rate, it will loose events.

Yampa also provides the combinator *noise :: (RandomGen g, Random b) => g -> SF a b* which generates a stream of noise by returning a random number in the default range for the type *b*.

Running signal functions To *purely* run a signal function Yampa provides the function *embed :: SF a b -> (a, [(DTime, Maybe a)]) -> [b]* which allows to run a SF for a given number of steps where in each step one provides the Δt and an input *a*. The function then returns the output of the signal function for each step. Note that the input is optional, indicated by *Maybe*. In the first step at $t = 0$, the initial *a* is applied and whenever the input is *Nothing* in subsequent steps, the last *a* which was not *Nothing* is re-used.

2.2 Arrowized programming

Yampa's signal functions are arrows, requiring us to program with arrows. Arrows are a generalisation of monads which, in addition to the already familiar parameterisation over the output type, allow parameterisation over their input type as well [10], [11].

In general, arrows can be understood to be computations that represent processes, which have an input of a specific type, process it and output a new type. This is the reason why Yampa is using arrows to represent their signal functions: the concept of processes, which signal functions are, maps naturally to arrows.

There exists a number of arrow combinators which allow arrowized programing in a point-free style but due to lack of space we will not discuss them here. Instead we make use of Paterson's do-notation for arrows [18] which makes code more readable as it allows us to program with points.

To show how arrowized programming works, we implement a simple signal function, which calculates the acceleration of a falling mass on its vertical axis as an example [21].

```
fallingMass :: Double -> Double -> SF () Double
fallingMass p0 v0 = proc _ -> do
  v <- arr (+v0) <<< integral -< (-9.8)
  p <- arr (+p0) <<< integral -< v
  returnA -< p
```

To create an arrow, the *proc* keyword is used, which binds a variable after which then the *do* of Patersons do-notation [18] follows. Using the signal function *integral :: SF v v* of Yampa which integrates the input value over time using the rectangle rule, we calculate the current velocity and the position based on the initial position *p0* and velocity *v0*. The <<< is one of the arrow combinators which composes two arrow computations and *arr* simply lifts a pure function into an arrow. To pass an input to an arrow, -< is used and <- to bind the result of an arrow computation to a variable. Finally to return a value from an arrow, *returnA* is used.

3 Defining Agent-Based Simulation

Agent-Based Simulation (ABS) is a methodology to model and simulate a system where the global behaviour may be unknown but the behaviour and interactions of the parts making up the system is known. Those parts, called agents, are modelled and simulated, out of which then the aggregate global behaviour of the whole system emerges.

So, the central aspect of ABS is the concept of an agent which can be understood as a metaphor for a pro-active unit, situated in an environment, able to spawn new agents and interacting with other agents in some neighbourhood by exchange of messages.

We informally assume the following about our agents [24], [31], [15]:

- They are uniquely addressable entities with some internal state over which they have full, exclusive control.
- They are pro-active which means they can initiate actions on their own e.g. change their internal state, send messages, create new agents, terminate themselves.
- They are situated in an environment and can interact with it.
- They can interact with other agents situated in the same environment by means of messaging.

4 The SIR Model

To explain the concepts of ABS and of our pure functional approach to it, we introduce the SIR model as a motivating example and use-case for our implementation. It is a very well studied and understood compartment model from epidemiology [13] which allows to simulate the dynamics of an infectious disease like influenza, tuberculosis, chicken pox, rubella and measles spreading through a population.

In this model, people in a population of size N can be in either one of three states *Susceptible*, *Infected* or *Recovered* at a particular time, where it is assumed that initially there is at least one infected person in the population. People interact *on average* with a given rate of β other people per time-unit and become infected with a given probability γ when interacting with an infected person. When infected, a person recovers *on average* after δ time-units and is then immune to further infections. An interaction between infected persons does not lead to re-infection, thus these interactions are ignored in this model. This definition gives rise to three compartments with the transitions seen in Figure 1.

Figure 1. States and transitions in the SIR compartment model.

Before looking into how one can simulate this model in an agent-based approach we first explain how to formalize it using System Dynamics (SD) [22]. In SD one models a system through differential equations, allowing to conveniently express continuous systems which change over time. The advantage of an SD solution is that one has an analytically tractable solution against which e.g. agent-based solutions can be validated. The problem is that, the more complex a system, the more difficult it is to derive differential equations describing the global system, to a point where it simply becomes impossible. This is the strength of an agent-based approach over SD, which allows to model a system when only the constituting parts and their interactions are known but not the macro behaviour of the whole system. As will be shown later, the agent-based approach exhibits further benefits over SD.

The dynamics of the SIR model can be formalized in SD with the following equations:

$$\frac{dS}{dt} = -infectionRate \tag{1}$$

$$\frac{dI}{dt} = infectionRate - recoveryRate \tag{2}$$

$$\frac{dR}{dt} = recoveryRate \tag{3}$$

$$infectionRate = \frac{I\beta S\gamma}{N} \tag{4}$$

$$recoveryRate = \frac{I}{\delta} \tag{5}$$

Solving these equations is done by numerically integrating over time which results in the dynamics as shown in Figure 2 with the given variables.

An Agent-Based approach

The SD approach is inherently top-down because the behaviour of the system is formalized in differential equations. This requires that the macro behaviour of the system is known a priori which may not always be the case. In the case of the SIR model we already have a top-down description

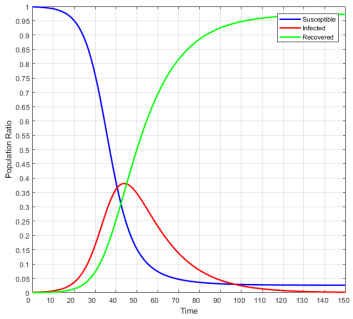


Figure 2. Dynamics of the SIR compartment model using the System Dynamics approach. Population Size N = 1,000, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time-steps.

of the system in the form of the differential equations from SD. We want now to derive an agent-based approach which exhibits the same dynamics as shown in Figure 2.

The question is whether such top-down dynamics can be achieved using ABS as well and whether there are fundamental drawbacks or benefits when doing so. Such questions were asked before and modelling the SIR model using an agent-based approach is indeed possible [14].

The fundamental difference is that SD is operating on averages, treating the population completely continuous which results in non-discrete values of stocks e.g. 3.1415 infected persons. The approach of mapping the SIR model to an ABS is to discretize the population and model each person in the population as an individual agent. The transitions between the states are no longer happening according to continuous differential equations but due to discrete events caused both by interactions amongst the agents and time-outs. Besides the already mentioned differences, the true advantage of ABS becomes now apparent: with it we can incorporate spatiality as shown in section **??** and simulate heterogenity of population e.g. different sex, age,... Note that the latter is theoretically possible in SD as well but with increasing number of population properties, it quickly becomes intractable.

According to the model, every agent makes *on average* contact with β random other agents per time unit. In ABS we can only contact discrete agents thus we model this by generating a random event on average every $\frac{1}{\beta}$ time units. We need to sample from an exponential distribution because the rate is proportional to the size of the population [3]. Note that an agent does not know the other agents' state when making contact with it, thus we need a mechanism in which agents reveal their state in which they are in *at the moment of making contact*. This mechanism is an implementation detail which we will derive in our implementation steps. For now we only assume that agents can make contact with each other somehow.

This results in the following agent behaviour:

- *Susceptible*: A susceptible agent makes contact *on average* with β other random agents. For every *infected* agent it gets into contact with, it becomes infected with a probability of γ. If an infection happens, it makes the transition to the *Infected* state.
- *Infected*: An infected agent recovers *on average* after δ time units. This is implemented by drawing the duration from an exponential distribution [3] with $\lambda = \frac{1}{\delta}$ and making the transition to the *Recovered* state after this duration.
- *Recovered*: These agents do nothing because this state is a terminating state from which there is no escape: recovered agents stay immune and can not get infected again in this model.

5 Deriving a pure functional approach

We presented a high-level agent-based approach to the SIR model in the previous section, which focused only on the states and the transitions, but we haven't talked about technical implementation.

The authors of [27] discuss two fundamental problems of implementing an agent-based simulation from a programming-language agnostic point of view. The first problem is how agents can be pro-active and the second how interactions and communication between agents can happen. For agents to be pro-active, they must be able to perceive the passing of time, which means there must be a concept of an agent-process which executes over time. Interactions between agents can be reduced to the problem of how an agent can expose information about its internal state which can be perceived by other agents.

In this section we will derive a pure functional approach for an agent-based simulation of the SIR model in which we will pose solutions to the previously mentioned problems. We will start out with a very naive approach and show its limitations which we overcome by adding FRP. Then in further steps we will add more concepts and generalisations, ending up at the final approach which utilises monadic stream functions (MSF), a generalisation of FRP [1].

As shown in the first step, the need to handle Δt explicitly can be quite messy, is inelegant and a potential source of errors, also the explicit handling of the state of an agent and its behavioural function is not very modular. We can solve both these weaknesses by switching to the functional reactive programming paradigm (FRP), because it allows to express systems with discrete and continuous time-semantics.

[1]The code of all steps can be accessed freely through the following URL: https://github.com/thalerjonathan/phd/tree/master/public/purefunctionalepidemics/code

In this step we are focusing on Arrowized FRP [10] using the library Yampa [8]. In it, time is handled implicit, meaning it cannot be messed with, which is achieved by building the whole system on the concept of signal functions (SF). An SF can be understood as a process over time and is technically a continuation which allows to capture state using closures. Both these fundamental features allow us to tackle the weaknesses of our first step and push our approach further towards a truly elegant functional approach.

5.0.1 Implementation

We start by defining an agent now as an SF which receives the states of all agents as input and outputs the state of the agent:

We start by modelling the states of the agents:

```
data SIRState = Susceptible | Infected | Recovered
```

Agents are ill for some duration, meaning we need to keep track when a potentially infected agent recovers. Also a simulation is stepped in discrete or continuous time-steps thus we introduce a notion of *time* and Δt by defining:

```
type Time      = Double
type TimeDelta = Double

type SIRAgent = SF [SIRState] SIRState
```

Now we can define the behaviour of an agent to be the following:

```
sirAgent :: RandomGen g => g -> SIRState -> SIRAgent
sirAgent g Susceptible = susceptibleAgent g
sirAgent g Infected    = infectedAgent g
sirAgent _ Recovered   = recoveredAgent
```

Depending on the initial state we return the corresponding behaviour. Most notably is the difference that we are now passing a random-number generator instead of running in the Random Monad because signal functions as implemented in Yampa are not capable of being monadic. We see that the recovered agent ignores the random-number generator which is in accordance with the implementation in the previous step where it acts as a sink which returns constantly the same state:

```
recoveredAgent :: SIRAgent
recoveredAgent = arr (const Recovered)
```

When an event occurs we can change the behaviour of an agent using the Yampa combinator *switch*, which is much more elegant and expressive than the initial approach as it makes the change of behaviour at the occurrence of an event explicit. Thus a susceptible agent behaves as susceptible until it becomes infected. Upon infection an *Event* is returned which results in switching into the *infectedAgent* SF, which causes the agent to behave as an infected agent from that moment on. Instead of randomly drawing the number of contacts to make, we now follow a fundamentally different approach by using Yampas *occasionally* function. This requires us to carefully select the right Δt for sampling the system as will be shown in results.

```
susceptibleAgent :: RandomGen g => g -> SIRAgent
susceptibleAgent g =
    switch (susceptible g) (const (infectedAgent g))
  where
    susceptible :: RandomGen g
      => g -> SF [SIRState] (SIRState, Event ())
    susceptible g = proc as -> do
      makeContact <- occasionally g (1 / contactRate) () -< ()
      if isEvent makeContact
        then (do
          a <- drawRandomElemSF g -< as
          case a of
            Infected -> do
              i <- randomBoolSF g infectivity -< ()
              if i
                then returnA -< (Infected, Event ())
                else returnA -< (Susceptible, NoEvent)
            _        -> returnA -< (Susceptible, NoEvent))
        else returnA -< (Susceptible, NoEvent)
```

We deal with randomness different now and implement signal functions built on the *noiseR* function provided by Yampa. This is an example for the stream character and statefulness of a signal function as it needs to keep track of the changed random-number generator internally through the use of continuations and closures. Here we provide the implementation of *randomBoolSF*, *drawRandomElemSF* works similar but takes a list as input and returns a randomly chosen element from it:

```
randomBoolSF :: RandomGen g => g -> Double -> SF () Bool
randomBoolSF g p = proc _ -> do
  r <- noiseR ((0, 1) :: (Double, Double)) g -< ()
  returnA -< (r <= p)
```

The infected agent behaves as infected until it recovers on average after the illness duration after which it behaves as a recovered agent by switching into *recoveredAgent*. As in the case of the susceptible agent, we use the *occasionally* function to generate the event when the agent recovers. Note that the infected agent ignores the states of the other agents as its behaviour is completely independent of them.

```
infectedAgent :: RandomGen g => g -> SIRAgent
infectedAgent g = switch infected (const recoveredAgent)
  where
    infected :: SF [SIRState] (SIRState, Event ())
    infected = proc _ -> do
      recEvt <- occasionally g illnessDuration () -< ()
      let a = event Infected (const Recovered) recEvt
      returnA -< (a, recEvt)
```

Running and stepping the simulation works now a bit differently, using Yampas function *embed*:

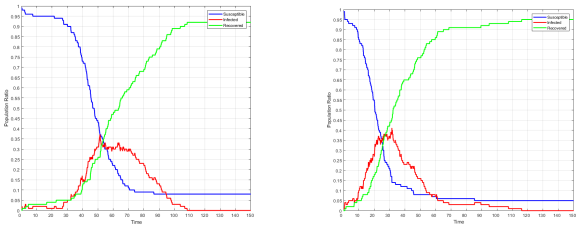
```
runSimulation :: RandomGen g
  => g -> Time -> DTime -> [SIRState] -> [[SIRState]]
runSimulation g t dt as
    = embed (stepSimulation sfs as) ((), dts)
  where
    steps     = floor (t / dt)
    dts       = replicate steps (dt, Nothing)
    n         = length as
    (rngs, _) = rngSplits g n [] -- unique rngs for each agent
    sfs       = map (\ (g', a) -> sirAgent g' a) (zip rngs as)
```

What we need to implement next is a closed feedback-loop. Fortunately, [16], [5] discusses implementing this in Yampa. The function *stepSimulation* is an implementation of such a closed feedback-loop. It takes the current signal functions and states of all agents, runs them all in parallel and returns the new agent states of this step. Yampa provides the *dpSwitch* combinator for running signal functions in parallel, which is quite involved and discussed more in-depth in section 2. It allows us to recursively switch back into the *stepSimulation* with the continuations and new states of all the agents after they were run in parallel. Note the use of *notYet* which is required because in Yampa switching occurs immediately at $t = 0$. Sometimes one needs to run a collection of signal functions in parallel and collect all of their outputs in a list. Yampa provides the combinator *dpSwitch* for it. It is quite involved and has the following type-signature:

```
dpSwitch :: Functor col
         -- routing function
         => (forall sf. a -> col sf -> col (b, sf))
         -- SF collection
         -> col (SF b c)
         -- SF generating switching event
         -> SF (a, col c) (Event d)
         -- continuation to invoke upon event
         -> (col (SF b c) -> d -> SF a (col c))
         -> SF a (col c)
```

Its first argument is the pairing-function which pairs up the input to the signal functions - it has to preserve the structure of the signal function collection. The second argument is the collection of signal functions to run. The third argument is a signal function generating the switching event. The last argument is a function which generates the continuation after the switching event has occurred. *dpSwitch* returns a new signal function which runs all the signal functions in parallel and switches into the continuation when the switching event occurs. The d in *dpSwitch* stands for decoupled which guarantees that it delays the switching until the next time-step: the function into which we switch is only applied in the next step, which prevents an infinite loop if we switch into a recursive continuation.

```
stepSimulation :: [SIRAgent] -> [SIRState] -> SF () [SIRState]
stepSimulation sfs as =
    dpSwitch
      -- feeding the agent states to each SF
      (\_ sfs' -> (map (\sf -> (as, sf)) sfs'))
      -- the signal functions
      sfs
      -- switching event, ignored at t = 0
      (switchingEvt >>> notYet)
      -- recursively switch back into stepSimulation
      stepSimulation
  where
    switchingEvt :: SF ((), [SIRState]) (Event [SIRState])
    switchingEvt = arr (\ (_, newAs) -> Event newAs)
```



(a) $\Delta t = 0.1$ **(b)** $\Delta t = 0.01$

Figure 3. FRP simulation of agent-based SIR showing the influence of different Δt. Population size of 100 with contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time-steps with respective Δt.

5.0.2 Results

The function which drives the dynamics of our simulation is *occasionally*, which randomly generates an event on average with a given rate following the exponential distribution. To arrive at the correct dynamics, this requires us to sample *occasionally*, and thus the whole system, with small enough Δt which matches the rate. If we choose a too large Δt, we loose events which will result in dynamics which do not approach the SD dynamics sufficiently enough, see Figure 3.

Clearly by keeping the population size constant and just increasing the Δt results in a closer approximation to the SD dynamics. To increasingly approximate the SD dynamics with ABS we still need a bigger population size and even smaller Δt. Unfortunately increasing both the number of agents and the sample rate results in severe performance and memory problems. A possible solution would be to implement super-sampling which would allow us to run the whole simulation with $\Delta t = 1.0$ and only sample the *occasionally* function with a much higher frequency.

5.0.3 Discussion

Reflecting on our first naive approach we can conclude that it already introduced most of the fundamental concepts of ABS

- Time - the simulation occurs over virtual time which is modelled explicitly divided into *fixed* Δt where at each step all agents are executed.
- Agents - we implement each agent as an individual, with the behaviour depending on its state.
- Feedback - the output state of the agent in the current time-step t is the input state for the next time-step $t + \Delta t$.
- Environment - as environment we implicitly assume a fully-connected network (complete graph) where every agent 'knows' every other agent, including itself and thus can make contact with all of them.

- Stochasticity - it is an inherently stochastic simulation, which is indicated by the Random Monad type and the usage of *randomBoolM* and *randomExpM*.
- Deterministic - repeated runs with the same initial random-number generator result in same dynamics. This may not come as a surprise but in Haskell we can guarantee that property statically already at compile time because our simulation runs in the Random Monad and *not* in the IO Monad. This guarantees that no external, uncontrollable sources of randomness can interfere with the simulation.
- Dynamics - with increasing number of agents the dynamics smooth out [14].

By moving on to FRP using Yampa we made a huge improvement in clarity, expressivity and robustness of our implementation. State is now implicitly encoded, depending on which signal function is active. Also by using explicit time-semantics with *occasionally* we can achieve extremely fine grained stochastics. Compared to drawing a random number of events we create only a single event or none at all. This requires to sample the system with a much smaller Δt: we are treating it as a continuous agent-based system, resulting in a hybrid SD/ABS approach.

So far we have an acceptable implementation of an agent-based SIR approach. What we are lacking at the moment is a general treatment of an environment. To conveniently introduce it we want to make use of monads which is not possible using Yampa. In the next step we make the transition to Monadic Stream Functions (MSF) as introduced in Dunai [20] which allows FRP within a monadic context.

6 Verification

TODO: explore ABS testing in pure functional Haskell - we need to distinguish between two types of testing/verification -> 1. testing/verification of models for which we have real-world data or an analytical solution which can act as a ground-truth. examples for such models are the SIR model, stock-market simulations, social simulations of all kind -> 2. testing/verification of models which are just exploratory and which are only be inspired by real-world phenomena. examples for such models are Epsteins Sugarscape and Agent_Zero

6.1 Black Box Verification

Defined as treating the functionality to test as a black box with inputs and outputs and comparing controlled inputs to expected outputs.

In Black Box Verification one generally feeds input and compares it to expected output. In the case of ABS we have two things to black-box test:

1. Isolated Agent Behaviour - test isolated agent behaviour under given inputs using unit- and property-based testing
2. Interacting Agent Behaviour - test if interaction between agents are correct
3. Simulation Dynamics - compare emergent dynamics of the ABS as a whole under given inputs to an analytical solution / real-world dynamics in case there exists some using statistical tests
4. Hypotheses- test whether hypotheses are valid / invalid using unit- and property-based testing. TODO: how can we formulate hypotheses in unit- and/or property-based tests?

- testing of the final dynamics: how close do they match the analytical solution - can we express model properties in tests e.g. quickcheck? - property-testing shines here - isolated tests: how easy can we test parts of an agent / simulation?

6.1.1 Finding optimal Δt

The selection of the right Δt can be quite difficult in FRP because we have to make assumptions about the system a priori. One could just play it safe with a very conservatively selected small $\Delta t <= 0.1$ but the smaller Δt, the lower the performance as it quickly multiplies the number of steps to calculate. Obviously one wants to select the *optimal* Δt, which in the case of ABS is the largest possible Δt for which we still get the correct simulation dynamics. To find out the *optimal* Δt one can make direct use of the Black Box tests: start with a large $\Delta t = 1.0$ and reduce it by half every time the tests fail until no more tests fail - if for $\Delta t = 1.0$ tests already pass, increasing it may be an option. It is important to note that although isolated agent behaviour tests might result in larger Δt, in the end when they are run in the aggregate system, one needs to sample the whole system with the smallest Δt found amongst all tests. Another option would be to apply super-sampling to just the parts which need a very small Δt but this is out of scope of this paper.

6.1.2 Agents as signals

Agents *might* behave as signals in FRP which means that their behaviour is completely determined by the passing of time: they only change when time changes thus if they are a signal they should stay constant if time stays constant. This means that they should not change in case one is sampling the system with $\Delta t = 0$. Of course to prove whether this will *always* be the case is strictly speaking impossible with a Black Box verification but we can gain a good level of confidence with them also because we are staying pure. It is only through white box verification that we can really guarantee and prove this property.

6.1.3 Comparison of dynamics against existing data

- utilise a statistical test with H0 "ABS and comparison is not the same" and H1 "ABS and comparison is the same" - how many replications and how do we average? - which statistical test do we implement? (steward robinson simulation book,

chapter 12.4.4) -> Normalizsed Mean Squared Error (NMSE) -> TODO: implement confidence interval -> TODO: what about chi-squared? -> TODO: what about paired-t confidence interval

IMPORTANT: this is not what we are after here in this paper, statistical tests are a science on their own and there actually exists quite a large amount of literature for conducting statistical tests on ABS dynamics: Robinson Book (TODO: find additional literature)

6.2 White Box Verification

White-Box verification is necessary when we need to reason about properties like *forever*, *never*, which cannot be guaranteed from black-box tests. Additional help can be coverage tests with which we can show that all code paths have been covered in our tests.

6.3 Property-Testing

TODO: describe what i did using quickcheck TODO: implement quickcheck test for combining susceptible and infected

6.4 White-box Verification

TODO: describe reasoning about the code

In this section we verify our agent-based implementation of the SIR model. Verification of our implementation should be fairly straight-forward and easy as the model is given in differential equations which gives us a formal specification of the model which we can use directly in our verification process. We will also conduct white-box verification and for one property it will be the only way of ensuring that our model is correct as we cannot guarantee it through black-box verification.

6.5 Black Box Verification

6.5.1 Agent Behaviour

When conducting black-box testing for the SIR model, we test if the *isolated* behaviour of an agent in all three states Susceptible, Infected and Recovered, corresponds to model specifications. The crucial thing though is that we are dealing with a stochastic system where the agents act *on averages*, which means we need to average our tests as well.

The interface of the agent behaviours are defined below. When running the SF with a given Δt one has to feed in the state of all the other agents as input and the agent outputs its state it is after this Δt.

```
data SIRState
  = Susceptible
  | Infected
  | Recovered

type SIRAgent = SF [SIRState] SIRState

susceptibleAgent :: RandomGen g => g -> SIRAgent
infectedAgent :: RandomGen g => g -> SIRAgent
recoveredAgent :: SIRAgent
```

Susceptible Behaviour A susceptible agent *may* become infected, depending on the number of infected agents in relation to non-infected the susceptible agent has contact to. To make this property testable we run a susceptible agent for 1.0 time-unit (note that we are sampling the system with small Δt e.g. 0.1) and then check if it is infected - that is it returns infected as its current state. Obviously we need to pay attention to the fact that we are dealing with a stochastic system thus we can only talk about averages and thus it does not suffice to only run a single agent but we are repeating this for e.g. $N = 10.000$ agents (all with different RNGs). We then need a formula for the required fraction of the N agents which should have become infected on average. Per 1.0 time-unit, a susceptible agent makes *on average* contact with β other agents where in the case of a contact with an infected agent the susceptible agent becomes infected with a given probability γ. In this description there is another probability hidden, which is the probability of making contact with an infected agent which is simply the ratio of number of infected agents to number not infected agents. The formula for the target fraction of agents which become infected is then: $\beta * \gamma * \frac{numberofinfected}{numberofnon-infected}$. To check whether this test has passed we compare the required amount of agents which on average should become infected to the one from our tests (simply count the agents which got infected and divide by N) and if the value lies within some small ϵ then we accept the test as passed.

Obviously the input to the susceptible agents which we can vary is the set of agents with which the susceptible agents make contact with. To save us from constructing all possible edge-cases and combinations and testing them with unit-tests we use QuickCheck which creates them randomly for us and reduces them also to all relevant edge-cases. This is an example for how to use property-based testing in ABS where QuickCheck can be of immense help generating random test-data to cover all cases.

TODO: derive the target-fraction formula from the differential equations TODO: can we encode this somehow on a type level using dependent types? then we don't need to test this property any more

Infected Behaviour An infected agent *will always* recover after a finite time, which is *on average* after δ time-units. Note that this property involves stochastics too, so to test this property we run a large number of infected agents e.g. $N = 10.000$ (all with different RNGs) until they recover, record the time of each agents recovery and then average over all recovery times. To check whether this test has passed we compare the average recovery times to δ and if they lie within some small ϵ then we accept the test as passed. We use QuickCheck in this case as well to generate the set of other

agents as input for the infected agents. Strictly speaking this would not be necessary as an infected agent never makes contact with other agents and simply ignores them - we could as well just feed in an empty list. We opted for using QuickCheck for the following reasons:

- We wanted to stick to the interface specification of the agent-implementation as close as possible which asks to pass the states of all agents as input.
- We shouldn't make any assumptions about the actual implementation and if it REALLY ignores the other agents, so we strictly stick to the interface which requires us to input the states of all the other agents.
- The set of other agents is ignored when determining whether the test has failed or not which indicates by construction that the behaviour of an infected agent does not depend on other agents.
- We are not just running a single replication over 10.000 agents but 100 of them which should give black-box verification more strength.

TODO: derive the average formula from the differential equations TODO: can we encode this somehow on a type level using dependent types? then we don't need to test this property any more

Recovered Behaviour A recovered agent will stay in the recovered state *forever*. Obviously we cannot write a black-box test that truly verifies that because it had to run in fact forever. In this case we need to resort to White Box Verification (see below).

Because we use multiple replications in combination with QuickCheck obviously results in longer test-runs (about 5 minutes on my machine) In our implementation we utilized the FRP paradigm. It seems that functional programming and FRP allow extremely easy testing of individual agent behaviour because FP and FRP compose extremely well which in turn means that there are no global dependencies as e.g. in OOP where we have to be very careful to clean up the system after each test - this is not an issue at all in our *pure* approach to ABS.

6.5.2 Simulation Dynamics

We won't go into the details of comparing the dynamics of an ABS to an analytical solution, that has been done already by [14]. What is important is to note that population-size matters: different population-size results in slightly different dynamics in SD => need same population size in ABS (probably...?). Note that it is utterly difficult to compare the dynamics of an ABS to the one of a SD approach as ABS dynamics are stochastic which explore a much wider spectrum of dynamics e.g. it could be the case, that the infected agent recovers without having infected any other agent, which would lead to an extreme mismatch to the SD approach but is absolutely a valid dynamic in the case of an ABS. The question is then rather if and how far those two are *really* comparable as it seems that the ABS is a more powerful system which presents many more paths through the dynamics. TODO: i really want to solve this for the SIR approach -> confidence intervals? -> NMSE? -> does it even make sense?

6.5.3 Finding optimal Δt

Obviously the *optimal* Δt of the SIR model depends heavily on the model parameters: contact rate β and illness duration δ. We fixed them in our tests to be $\beta = 5$ and $\delta = 15$. By using the isolated behaviour tests we found an optimal $\Delta t = 0.125$ for the susceptible behaviour and $\Delta t = 0.25$ for the infected behaviour. TODO: dynamics comparison?

6.5.4 Agents as signals

Our SIR agents *are* signals due to the underlying continuous nature of the analytical SIR model and to some extent we can guarantee this through black box testing. For this we write tests for each individual behaviour as previously but instead of checking whether agents got infected or have recovered we assume that they stay constant: they will output always the same state when sampling the system with $\Delta t = 0$. The tests are conceptual the complementary tests of the previous behaviour tests so in conjunction with them we can assume to some extent that agents are signals. To prove it, we need to look into white box verification as we cannot make guarantees about properties which should hold *forever* in a computational setting.

6.6 White Box Verification

In the case of the SIR model we have the following invariants:

- A susceptible agent will *never* make the transition to recovered.
- An infected agent will *never* make the transition to susceptible.
- A recovered agent will *forever* stay recovered.

All these invariants can be guaranteed when reasoning about the code. An additional help will be then coverage testing with which we can show that an infected agent never returns susceptible, and a susceptible agent never returned infected given all of their functionality was covered which has to imply that it can never occur!

Lets start with looking at the recovered behaviour as it is the simplest one. We then continue with the infected behaviour and end with the susceptible behaviour as it is the most complex one.

Recovered Behaviour The implementation of the recovered behaviour is as follows:

```
recoveredAgent :: SIRAgent
recoveredAgent = arr (const Recovered)
```

Just by looking at the type we can guarantee the following:

- it is pure, no side-effects of any kind can occur

- no stochasticity possible because no RNG is fed in / we don't run in the random monad

The implementation is as concise as it can get and we can reason that it is indeed a correct implementation of the recovered specification: we lift the constant function which returns the Recovered state into an arrow. Per definition and by looking at the implementation, the constant function ignores its input and returns always the same value. This is exactly the behaviour which we need for the recovered agent. Thus we can reason that the recovered agent will return Recovered *forever* which means our implementation is indeed correct.

Infected Behaviour

```
infectedAgent :: RandomGen g
              => g
              -> Double
              -> SIRAgent
infectedAgent g illnessDuration =
    switch
      infected
      (const recoveredAgent)
  where
    infected :: SF [SIRState] (SIRState, Event ())
    infected = proc _ -> do
      recEvt <- occasionally g illnessDuration () -< ()
      let a = event Infected (const Recovered) recEvt
      returnA -< (a, recEvt)
```

By looking at the types we can reason that there *may* be some stochasticity involved (the function may choose to ignore the RNG) and that we are again pure. TODO: not finished yet

Susceptible Behaviour

```
susceptibleAgent :: RandomGen g
                 => g
                 -> Double
                 -> Double
                 -> Double
                 -> SIRAgent
susceptibleAgent g contactRate infectivity illnessDuration =
    switch
      (susceptible g)
      (const (infectedAgent g illnessDuration))
  where
    susceptible :: RandomGen g => g -> SF [SIRState] (SIRState, Event ())
    susceptible g = proc as -> do
      makeContact <- occasionally g (1 / contactRate) () -< ()

      -- NOTE: strangely if we are not splitting all if-then-else into
      -- separate but only a single one, then it seems not to work,
      -- dunno why
      if isEvent makeContact
        then (do
          a <- drawRandomElemSF g -< as
          case a of
            Just Infected -> do
              i <- randomBoolSF g infectivity -< ()
              if i
                then returnA -< (Infected, Event ())
                else returnA -< (Susceptible, NoEvent)
            _             -> returnA -< (Susceptible, NoEvent))
        else returnA -< (Susceptible, NoEvent)
```

TODO: not finished yet

6.7 Interacting Agent Behaviour

TODO: haven't look into this yet

7 Related Work

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 more related to the field of Multi Agent Systems and look into how agents can be specified using the belief-desire-intention paradigm [6], [26], [12].

A library for Discrete Event Simulation (DES) and System Dynamics (SD) in Haskell called *Aivika 3* is described in the technical report [25]. It is not pure, as it uses the IO Monad under the hood and comes only with very basic features for event-driven ABS, which allows to specify simple state-based agents with timed transitions.

The authors of [12] discuss using functional programming for DES and explicitly mention the paradigm of FRP to be very suitable to DES.

The authors of [29] present a domain-specific language for developing functional reactive agent-based simulations. 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 Yampa 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.

8 Conclusions

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 hybrid approach, it forces one to think properly of time-semantics of the model and how small Δt should be. Third it requires 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. Within the agents there are no side effects possible which could result in differences between same runs. Every agent has access to its own random-number generator or the Random Monad, allowing randomness to occur in the simulation but the random-generator seed is fixed in the beginning and can never be changed within an agent. This means that after initialising the agents, which *could* run in the IO Monad, the simulation itself runs completely deterministic.

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 [21]. Also by using FRP we gain all the benefits from it and can use research on testing, debugging and exploring FRP systems [21], [19].

Issues

Unfortunately, the hybrid approach of SD/ABS amplifies the performance issues of agent-based approaches, which requires much more processing power compared to SD, because each agent is modelled individually in contrast to aggregates in SD [14]. With the need to sample the system with high frequency, this issue gets worse. We haven't investigated how to optimize the performance by using efficient functional data structures, hence in the moment our program performs much worse than an imperative implementation that exploits in-place updates.

Despite the strengths and benefits we get by leveraging on FRP, there are errors that are not raised at compile-time, e.g. we can still have infinite loops and run-time errors. This was for example investigated by [23] who use dependent types to avoid some runt-time errors in FRP. We suggest that one could go further and develop a domain specific type system for FRP that makes the FRP based ABS more predictable and that would support further mathematical analysis of its properties.

We can conclude that the main difficulty of a pure functional approach evolves around the communication and interaction between agents. This 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 and we have added further mechanisms of agent interaction which we had to omit due to lack of space. We hypothesise that MSFs allow us to conveniently express agent communication but but leave this for further research.

We started with high hopes for the pure functional approach and hypothesized that it will be truly superior to existing traditional object-oriented approaches but we come to the conclusion that this is not so. The single real benefit is the lack of implicit side-effects and reproducibility guaranteed at compile time. Still, our research was not in vain as we see it as an intermediary step towards using dependent types. Moving to dependent types would pose a unique benefit over the object-oriented approach and should allow us to express and guarantee properties at compile time which is not possible with imperative approaches. We leave this for further research.

9 Further Research

We see this paper as an intermediary and necessary step towards dependent types for which we first needed to understand the potential and limitations of a non-dependently typed pure functional approach in Haskell. Dependent types are extremely promising in functional programming as they allow us to express stronger guarantees about the correctness of programs and go as far as allowing to formulate programs and types as constructive proofs which must be total by definition [28], [2], [1].

So far no research using dependent types in agent-based simulation exists at all and it is not clear whether dependent types make sense in this context. In our next paper we want to explore this for the first time and ask more specifically how we can add dependent types to our pure functional approach, which conceptual implications this has for ABS and what we gain from doing so. We plan on using Idris [4] as the language of choice as it is very close to Haskell with focus on real-world application and running programs as opposed to other languages with dependent types e.g. Agda and Coq which serve primarily as proof assistants.

It would be of immense interest whether we could apply dependent types to the model meta-level or not - this boils down to the question if we can encode our model specification in a dependently typed way. This would allow the ABS community for the first time to reason about a model directly in code.

Acknowledgments

The authors would like to thank I. Perez, H. Nilsson, J. Greensmith, M. Baerenz, H. Vollbrecht, S. Venkatesan, J. Hey and the Haskell Symposium 2018 referees for constructive feedback, comments and valuable discussions.

References

[1] Thorsten Altenkirch, Nils Anders Danielsson, Andres Loeh, and Nicolas Oury. 2010. Pi Sigma: Dependent Types Without the Sugar. In *Proceedings of the 10th International Conference on Functional and Logic Programming (FLOPS'10)*. Springer-Verlag, Berlin, Heidelberg, 40–55. https://doi.org/10.1007/978-3-642-12251-4_5

[2] Thorsten Altenkirch, Conor Mcbride, and James Mckinna. 2005. Why dependent types matter. In *In preparation, http://www.e-pig.org/downloads/ydtm.pdf*.

[3] Andrei Borshchev and Alexei Filippov. 2004. From System Dynamics and Discrete Event to Practical Agent Based Modeling: Reasons, Techniques, Tools. Oxford.

[4] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 05 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

[5] 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

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

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

[8] 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

[9] 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

[10] 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

[11] 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

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

[13] W. O. Kermack and A. G. McKendrick. 1927. A Contribution to the Mathematical Theory of Epidemics. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 115, 772 (Aug. 1927), 700–721. https://doi.org/10.1098/rspa.1927.0118

[14] 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

[15] 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

[16] 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

[17] 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.

[18] Ross Paterson. 2001. A New Notation for Arrows. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*. ACM, New York, NY, USA, 229–240. https://doi.org/10.1145/507635.507664

[19] 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

[20] 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

[21] 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

[22] Donald E. Porter. 1962. Industrial Dynamics. Jay Forrester. M.I.T. Press, Cambridge, Mass.; Wiley, New York, 1961. xv + 464 pp. Illus. $18. *Science* 135, 3502 (Feb. 1962), 426–427. https://doi.org/10.1126/science.135.3502.426-a

[23] Neil Sculthorpe and Henrik Nilsson. 2009. Safe Functional Reactive Programming Through Dependent Types. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 23–34. https://doi.org/10.1145/1596550.1596558

[24] Peer-Olaf Siebers and Uwe Aickelin. 2008. Introduction to Multi-Agent Simulation. *arXiv:0803.3905 [cs]* (March 2008). http://arxiv.org/abs/0803.3905 arXiv: 0803.3905.

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

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

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

[28] Simon Thompson. 1991. *Type Theory and Functional Programming*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[29] 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

[30] 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

[31] Michael Wooldridge. 2009. *An Introduction to MultiAgent Systems* (2nd ed.). Wiley Publishing.

Received March 2018