Pure functional epidemics

An Agent-Based Approach

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

TODO: fill in the right metainfo TODO: select the right copyright TODO: get DOI TODO: fill in paper history TODO: generate CCSXML

TODO: find a way to deactivate pygmentize

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 macro system behaviour emerges. So far mainly object-oriented techniques and languages have been used in ABS. Using the SIR model of epidemiology as an example we show how to use Functional Reactive Programming to implement ABS. We claim that this representation is conceptually cleaner and opens the way to formally reason about ABS.

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

ACM Reference Format:
Jonathan Thaler, Thorsten Altenkirch, and Peer-Olaf Siebers. 2018. Pure functional epidemics: An Agent-Based Approach. *Proc. ACM Program. Lang.* 9, 4, Article 39 (September 2018), 35 pages. https://doi.org/0000001.0000001

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 [15] in which the authors claim that "[..] 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* [29] 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 [18]: 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 in a realistic way and over the course of five 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 but then refine it in various ways, making the transition to Functional Reactive Programming (FRP) [49] and to Monadic Stream Functions (MSF) [33]. 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

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

2475-1421/2018/9-ART39
https://doi.org/0000001.0000001

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:

- 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. Although there exist some work on ABS in Haskell they are not of the conceptual character this paper exhibits, barely scratch the surface, sacrifice purity and do focus on different problems.
- Our approach shows how robustness can be achieved through purity which guarantees reproducibility at compile time, something not possible with traditional object-oriented approaches.
- The result of using arrowized FRP is a unique, hybrid approach to ABS because FRP allows expressing continuous time-semantics not possible with the traditional imperative object-oriented approaches.

Section 2 discusses related work. In section 3 we introduce functional reactive programming, arrows and monadic stream functions because our approach builds heavily on these concepts. Section 4 defines agent-based simulation and introduces the ideas behind it. In section 5 we discuss the SIR model of epidemiology as an example model to explain the concepts of ABS using a concrete model. The heart of the paper is section 6 in which we derive the concepts of a pure functional approach to ABS in five steps, using the SIR model as example. Finally we draw conclusions and discuss issues in section 7 and point to further research 8.

2 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 (MAS) and look into how agents can be specified using the belief-desire-intention paradigm [11], [43], [21]. A library for Discrete Event Simulation (DES) and System Dynamics (SD) in Haskell called *Aivika 3* is described in the technical report [42]. It is not pure as it uses the IO Monad under the hood and comes with very basic features for event-driven ABS which allows to specify simple state-based agents with timed transitions. The authors of [21] discuss using functional programming for DES and explicitly mention the paradigm of FRP to be very suitable to DES. In his talk [44], Tim Sweeney CTO of Epic Games discussed programming languages in the development of game engines and scripting of game logic. Although the fields of games and ABS seem to be very different, Gregory [16] defines computer-games as *"[..] soft real-time interactive agent-based computer simulations"* (p. 9) and 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. In games these objects are called *game-objects* and in ABS they are called *agents* but they are conceptually the same thing. The two main points Sweeney made were that dependent types could solve most of the run-time failures and that parallelism is the future for performance improvement in games. He distinguishes between pure functional algorithms which can be parallelized easily in a pure functional language and updating game-objects concurrently using software transactional memory (STM).

The thesis of [5] constructs two frameworks: an agent-modelling framework and a DES framework, both written in Haskell. They put special emphasis on parallel and concurrency in their work. The author develops two programs with strong emphasis on parallelism: HLogo which is a clone of the NetLogo agent-modelling framework and HDES, a framework for discrete event simulation.

The authors of [38] and [47] 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.

3 BACKGROUND

3.1 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* [19], [20] FRP as implemented in the library Yampa [17], [9], [27]. 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.

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

Signal functions are implemented as continuations which don't take a Δt at $t = 0$ but then change their signature into one which takes a Δt for $t > 0$. This allows to hide Δt completely from the types which makes them much more suitable for declarative programming. Yampa provides a number of combinators for expressing time-semantics, events and state-changes of the system. They allow us 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 the papers [17], [9], [27].

Event. Yampa represents events through the *Event* type which is the same as 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.

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
         => (forall sf. a -> col sf -> col (b, sf))  -- routing function
         -> col (SF b c)                             -- SF collection
         -> SF (a, col c) (Event d)                  -- SF generating switching event
         -> (col (SF b c) -> d -> SF a (col c))      -- continuation to invoke upon event
         -> 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 (thus the p) and switching into the continuation when the switching event occurs. The d in *dpSwitch* stands for delayed which guarantees that it delays the switching until the next time e.g. 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.

Randomness. In ABS one often needs to generate stochastic events which occur based on an exponential distribution. Yampa provides the combinator *occasionally :: RandomGen g => g -> Time -> b -> SF a (Event b)* for this. It take 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 first a is applied and whenever the input is *Nothing*, the last input a which was not *Nothing* is re-used.

3.2 Arrowized programming

Yampas signal functions are arrows, requiring us to program with arrows, which is a bit different than programming with monads, so we will introduce the concept of arrows and programming with them shortly. 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 [19], [20]. 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 Patersons do-notation for arrows [30] which makes the code much more readable as it allows us to program with points instead of using only the point-free arrow combinators. 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 [34].

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

The keyword *proc* is the same as introducing a lambda abstraction but for arrows. After it the arguments to the process follow, which in this case we ignore as it is just the unit type. 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 the familiar symbol <- to assign the result of an arrow computation to a variable. Finally to return a value in an arrow the function *returnA* is used which is itself an arrow - the identity function lifted.

3.3 Monadic Stream Functions

Monadic Stream Functions (MSF) are a generalisation of Yampas signal functions with additional combinators to control and stack side effects. A MSF is a polymorphic type and an evaluation function which applies an MSF to an input and returns an output and a continuation, both in a monadic context [33], [32]:

```
newtype MSF m a b = step :: Monad m => MSF m a b -> a -> m (b, MSF m a b)
```

MSFs are also arrows which means we can apply arrowized programming with Patersons do-notation as well. The authors [33] implement the library Dunai, which is available on Hackage. It allows us to run monadic computations within a signal function by providing the combinators *arrM :: Monad m => (a -> m b) -> MSF m a b* and *arrM_ :: Monad m => m b -> MSF m a b*.

4 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 of knowledge. 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 [40], [51], [41], [10], [25]:

- 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 which are situated in the same environment by means of messaging.

Epstein [13] identifies ABS to be especially applicable for analysing *"spatially distributed systems of heterogeneous autonomous actors with bounded information and computing capacity"*. It exhibits the following properties:

- Linearity & Non-Linearity - actions of agents can lead to non-linear behaviour of the system.
- Time - agents act over time and time is also the source of pro-activity.
- States - agents encapsulate some state which can be accessed and changed during the simulation.
- Feedback-Loops - because agents act continuously and their actions influence each other and themselves in subsequent time-steps, feedback-loops are the norm in ABS.
- Heterogeneity - although agents can have same properties like height, sex,... the actual values can vary arbitrarily between agents.
- Interactions - agents can be modelled after interactions with an environment or other agents.
- Spatiality & Networks - agents can be situated within e.g. a spatial (discrete 2D, continuous 3D,...) or complex network environment.

Note that there does not exist a commonly agreed technical definition of ABS but the field draws inspiration from the closely related field of Multi-Agent Systems (MAS) [51], [50]. It is important to understand that MAS and ABS are two different fields where in MAS the focus is much more on technical details implementing a system of interacting intelligent agents within a highly complex environment with the focus primarily on solving AI problems.

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

5 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 [23] which allows to simulate the dynamics of an infectious disease like influenza, tuberculosis, chicken pox, rubella and measles [12] 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 as seen in Figure 1.

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) [35]. In SD one models a system through differential equations, allowing to conveniently express continuous systems which change over time. The advantage of a 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:
TODO: there seems to be an unnerving space after the f letters, can we get rid of them?

$$\frac{\mathrm{d}S}{\mathrm{d}t} = -infectionRate \tag{1}$$

$$\frac{\mathrm{d}I}{\mathrm{d}t} = infectionRate - recoveryRate \tag{2}$$

$$\frac{\mathrm{d}R}{\mathrm{d}t} = recoveryRate \tag{3}$$

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

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

Solving these equations is done by integrating over time. In the SD terminology, the integrals are called *Stocks* and the values over which is integrated over time are called *Flows*. At $t = 0$ a single agent is infected because if there wouldn't be any infected agents, the system would immediately reach equilibrium - this is also the formal definition of the steady state of the system: as soon as $I(t) = 0$ the system won't change any more.

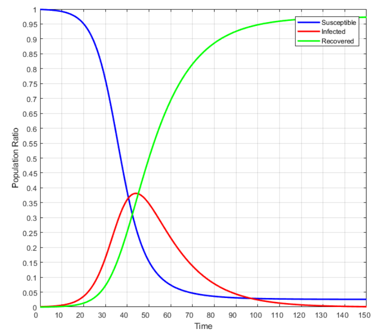


Fig. 2. Dynamics of the SIR compartment model using the System Dynamics approach. Population Size N = 1,000, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time-steps.

TODO: there seems to be an unnerving space after the f letters, can we get rid of them?

$$S(t) = N - I(0) + \int_0^t -infectionRate \, \mathrm{d}t \tag{6}$$

$$I(0) = 1 \tag{7}$$

$$I(t) = \int_0^t infectionRate - recoveryRate \, \mathrm{d}t \tag{8}$$

$$R(t) = \int_0^t recoveryRate \, \mathrm{d}t \tag{9}$$

Running the SD simulation over time 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 have already a top-down description 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 2. The questions are 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 [24]. 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 6.5 and simulate heterogenity of population e.g. different parameters for e.g. different sex, age,... Note that the later 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 β time units. We need to sample from an exponential distribution because the rate is proportional to the size of the population [6]. 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. Although in the model all agents make contact with each other, we can reduce the contacts to a few cases which have an influence on the dynamics. Contacts between agents with the same state as well as contacts with recovered agents have no influence, thus we need only to focus on contacts between susceptible and infected agents. In the agent-based approach we implement the following 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 [1]. This is implemented by drawing the duration from an exponential distribution [6] 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 [2].

For a more in-depth introduction of how to approximate an SD model by ABS see [24] who discusses a general approach and how to compare dynamics and [6] which explain the need to draw the illness-duration from an exponential-distribution. For comparing the dynamics of the SD and ABS approach to real-world epidemics see [2].

6 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 details on how to actually implement such a state-machine. The authors of [45] discuss two fundamental problems of implementing an agent-based simulation from a programming language agnostic view. The first problem is how agents can be pro-active and the second how interactions 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 mentiond problems. We will start out with a very naive approach and show its limitations which we overcome by adding FRP. Then in further

[1]Note that these agents do *not* pro-actively contact other agents. The rationale behind it is that according to the theory of epidemiology β is defined as *a contact which would be sufficient to lead to infection, were it to occur between a susceptible and an infected individual* [1]. In this theory it wouldn't make sense to work out the β value for someone who is already infected - the contact structure of the infected agents is implicit. Would we add pro-active contact making to the infected agents as well we would get very different results, not matching the SD dynamics which were validated in the past against real world epidemics [2].

[2]There exists an extended SIR model, called SIRS which adds a cycle to the state transitions by introducing a transition from recovered back to susceptible but we don't consider that here.

steps we will add more concepts and generalisations, ending up at the final approach which utilises monadic stream functions (MSF), a generalisation of FRP [3].

6.1 Naive beginnings

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
```

Now we can represent every agent simply as a tuple of its SIR state and its potential recovery time. We hold all our agents in a list:

```
type SIRAgent = (SIRState, Time)
type Agents   = [SIRAgent]
```

Next we need to think about how to actually step our simulation. For this we define a function which advances our simulation with a fixed Δt until a given time t where in each step the agents are processed and the output is fed back into the next step. This is the source of pro-activity as agents are executed in every time step and can thus initiate actions based on the passing of time. As already mentioned in previous sections, the agent-based implementation of the SIR model is inherently stochastic which means we need access to a random-number generator. We decided to use the Random Monad at this point as threading a generator through the simulation and the agents is very cumbersome. Thus our simulation stepping runs in the Random Monad:

```
runSimulation :: RandomGen g => Time -> TimeDelta -> Agents -> Rand g [Agents]
runSimulation tEnd dt as = runSimulationAux 0 as []
  where
    runSimulationAux :: RandomGen g => Time -> Agents -> [Agents] -> Rand g [Agents]
    runSimulationAux t as acc
      | t >= tEnd = return (reverse (as : acc))
      | otherwise = do
        as' <- stepSimulation dt as
        runSimulationAux (t + dt) as' (as : acc)

stepSimulation :: RandomGen g => TimeDelta -> Agents -> Rand g Agents
stepSimulation dt as = mapM (processAgent dt as) as
```

Now we can implement the behaviour of an individual agent. First we need to distinguish between the agents SIR states:

```
processAgent :: RandomGen g => TimeDelta -> Agents -> SIRAgent -> Rand g SIRAgent
processAgent _  as   (Susceptible, _) = susceptibleAgent as
processAgent dt _  a@(Infected   , _) = return (infectedAgent dt a)
processAgent _  _  a@(Recovered  , _) = return a
```

An agent gets fed the states of all agents in the system from the previous time-step so it can draw random contacts - this is one, very naive way of implementing the interactions between agents. Note that this includes also the agent itself thus we would need to omit the agent itself to prevent making contact with itself. We decided against that as it complicates the solution and for larger numbers of agent population the probability for an agent to make contact with itself is so small

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

that it can be neglected. Also making contact with the same SIR state never leads to a state change so it really makes no big difference.

From our implementation it becomes apparent that only the behaviour of a susceptible agent involves randomness and that a recovered agent is simply a sink - it does nothing and stays constant.

Lets look how we can implement the behaviour of a susceptible agent. It simply makes contact on average with a number of other agents and gets infected with a given probability if an agent it has contact with is infected. When the agent gets infected it calculates also its time of recovery by drawing a random number from the exponential distribution meaning it is ill on average for *illnessDuration*.

```
susceptibleAgent :: RandomGen g => Agents -> Rand g SIRAgent
susceptibleAgent as = do
    rc <- randomExpM (1 / contactRate)
    cs <- forM ([0..floor rc - 1] :: [Int]) (const (makeContact as))
    if elem True cs
      then infect
      else return (Susceptible, 0)
  where
    makeContact :: RandomGen g => Agents -> Rand g Bool
    makeContact as = do
      randContact <- randomElem as
      case fst randContact of
        Infected -> randomBoolM infectivity -- returns True with a given probability between 0..1
        _        -> return False

    infect :: RandomGen g => Rand g SIRAgent
    infect = do
      randIllDur <- randomExpM (1 / illnessDuration) -- draws from an exponential distribution
      return (Infected, randIllDur)
```

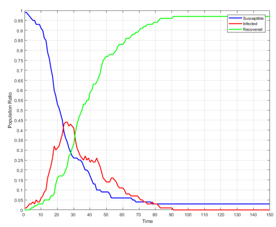
The infected agent is trivial. It simply recovers after the given illness duration which is implemented as follows:

```
infectedAgent :: TimeDelta -> SIRAgent -> SIRAgent
infectedAgent dt (_, t)
    | t' <= 0    = (Recovered, 0)
    | otherwise = (Infected, t')
  where
    t' = t - dt
```

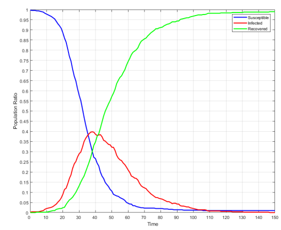
6.1.1 Results. When running our naive implementation with increasing population sizes we get the dynamics as seen in Figure 3. With increasing number of agents [24] our solution becomes increasingly smoother and approaches the SD dynamics from Figure 2 but doesn't quite match them because we are under-sampling the contact-rate. We will will address this problem in the next section.

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

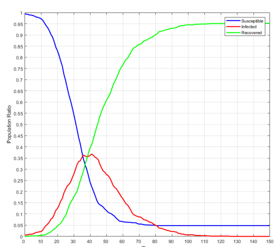
- Time - the simulation occurs over virtual time which is modelled explicitly divided into *fixed* Δt where at each the agents are executed.
- Agents - we implement each agent as an individual behaviour which depends on the agents state.
- Feedback - the output state of the agent in the current time-step t is the input state for the next time-step $t + 1$.



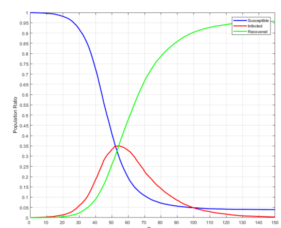
(a) 100 Agents



(b) 500 Agents



(c) 1,000 Agents



(d) 10,000 Agents

Fig. 3. Naive simulation of SIR using agent-based approach. Varying population size, 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 fixed $\Delta t = 1.0$.

- Environment - as environment we implicitly assume a fully-connected network (complete graph) where every agent 'knows' every other agents, including itself and thus can make contact all of them.
- Stochasticity - its 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 [24].

Nonetheless our approach has also weaknesses and dangers:

(1) Δt is passed explicitly as argument to the agent and needs to be dealt with explicitly.This is not very elegant and a potential source of errors - can we do better and find a more elegant solution?
(2) The way our agents are represented is not very elegant. The state of the agent is explicitly encoded in an ADT and when processing the agent, the function needs always first distinguish between the states. Can we express it in a more implicit, functional way e.g. continuations?
(3) The states of all agents of the current step are fed back into every agent in the next step so that an agent can pick its contacts. Although agents cannot change the states of others, this reveals too much information e.g. the illness duration is of no interest to the other agents. Although we could just feed in the *SIRState* without the illness duration, the problem is

more of conceptual nature - it should be the agent which decides to whom it reveals which information.

We move now to the next section in which we will address points 1 and 2. Points 3 and the under-sampling issue will be addressed in section 6.3.

6.2 Adding Functional Reactive Programming

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. In this step we are focusing on arrowized [19] FRP using the library Yampa [17]. 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). A 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 functional approach.

6.2.1 Implementation. We start by defining our agents now as a SF which receives the states of all agents as input and outputs the state of the agent:

```
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 one of three functions. 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)
```

By using the Yampa combinator *switch* we can change the behaviour of an agent when an event occurs which is much more elegant then the initial approach and much more expressive 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. It generates on average an event after the given time, so in each time-step we generate either a single event or no event. This requires a fundamental different approach in selecting the right Δt and 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 differently now and implement signal functions built on the *noiseR* function provided by Yampa. This is another example of 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:

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

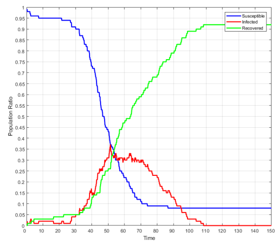
The infected agent behaves as infected until it recovers on average after the illness duration after which it behaves as a recovered agent by switching into *recoveredAgent*. As in the case of the susceptible agent, we use the *occasionally* function to generate the event when the agent recovers. Note 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)
```

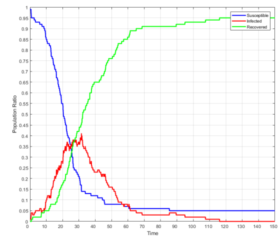
Running and stepping the simulation works now a bit differently. Yampa provides the function *embed* which allows us to run a SF for a given number of steps where in each step one provides the Δt and an optional input.

```
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 [] -- creates unique RandomGens for each agent
    sfs       = map (\ (g', a) -> sirAgent g' a) (zip rngs as)
```

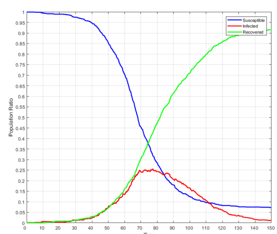
What we now need to implement is a closed feedback-loop. Fortunately, [27], [9] discusses implementing this in Yampa. The function *stepSimulation* is an implementation of such a closed feedback-loop. It takes the current signal functions and states of all agents, runs them all in parallel and returns 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 3. 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. In every step we generate a switching event using *switchingEvt* which receives the outputs of all the run signal functions. It always returns an *Event*



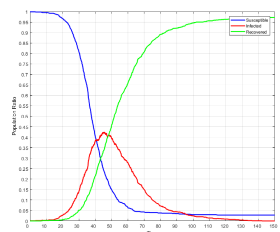
(a) 100 Agents, $\Delta t = 0.1$



(b) 100 Agents, $\Delta t = 0.01$



(c) 1,000 Agents, $\Delta t = 0.1$



(d) 1,000 Agents, $\Delta t = 0.01$

Fig. 4. FRP simulation of SIR using agent-based approach. Population size of 100 and 1,000 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 various Δt.

with all the outputs which causes to switch into the continuation which is *stepSimulation* itself . Note the use of *notYet* which is required because in Yampa switching occurs immediately at $t = 0$.

```
stepSimulation :: [SIRAgent] -> [SIRState] -> SF () [SIRState]
stepSimulation sfs as =
    dpSwitch
      (\_ sfs' -> (map (\sf -> (as, sf)) sfs'))
      sfs
      (switchingEvt >>> notYet)
      stepSimulation
  where
    switchingEvt :: SF ((), [SIRState]) (Event [SIRState])
    switchingEvt = arr (\ (_, newAs) -> Event newAs)
```

6.2.2 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 4.

Clearly by keeping the population size constant and just increasing the Δt results in a closer approximation to the SD dynamics. Although the dynamics of Figure 4d with 1000 agents and $\Delta t = 0.01$ look pretty close to SD, we are still not yet there. We would need to both decrease the sampling rate and increase the number of agents. Unfortunately at this point we are running into severe performance and memory problems because the whole system has to be sampled at an even finer Δt whereas we only need to sample *occasionally* with higher frequency. 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. An approach would be to introduce a new function to Yampa which allows to super-sample other signal functions.

```
superSampling :: Int -> SF a b -> SF a [b]
```

It evaluates *sf* for *n* times, each with $\Delta t = \frac{\Delta t}{n}$ and the same input argument *a* for all *n* evaluations. At time 0 no super-sampling is performed and just a single output of *sf* is calculated. A list of *b* is returned with length of *n* containing the result of the *n* evaluations of *sf*. If 0 or less super samples are requested exactly one is calculated. We could then just wrap the occasionally function which would then generate a list of events. We have investigated super-sampling more in-depth but have to leave this due to lack of space.

6.2.3 Discussion. 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 truly continuous system, resulting in a hybrid SD/ABS approach. Still we are not too happy about our approach as we feed back all agents states into every agent, something we want to omit in an agent-based simulation. We now move on to the next section in which we introduce a more general and much more controlled mechanism for feeding back agent-states.

6.3 Adding arbitrary data-flow

As already mentioned in the previous step, by revealing the state of every agent to all other agents makes the interactions implicit and deprives the agent of its control over which data which other agent sees. As a remedy we introduce data-flows which allow an agent to send arbitrary data to other agents which makes the communication and feedback amongst agents explicit. We cannot express this directly with Yampa combinators as data-flow is hard-wired at compile-time so we need to implement our own mechanism. The data will be collected from the sending agents and distributed to the receivers after each step, which means that we have a delay of one Δt and a round-trip takes $2\Delta t$ - which is exactly the feedback behaviour we had before. This change requires a different approach of how the agents interact with each other: a susceptible agent then sends to a random agent a data-flow indicating a contact. Only infected agents need to reply to such contact requests by revealing that they are infected. The susceptible agents then need to check for incoming replies which means they were in contact with an infected agent. The deeper motivation behind this data-flow mechanism is the need of an ABS to provide some means to allow agents to interact and communicate with each other. In object-oriented approaches this is rather trivial because agents can call methods of other agents directly and implicitly mutate the other agents state. When following a *pure* approach in Haskell this becomes much more difficult and the data-flow mechanism is our ad-hoc solution for this problem.

6.3.1 Implementation. First we need a way of addressing agents, which we do by introducing unique agent ids. Also we need a data-package which identifies the receiver and carries the data:

```
type AgentId      = Int
type DataFlow d = (AgentId, d)
```

Next we need more general input and output types of our agents signal functions. We introduce a new input type which holds both the agent id of the agent and the incoming data-flows from other agents:

```
data AgentIn d = AgentIn
  { aiId   :: AgentId
  , aiData :: [DataFlow d] }
```

We also introduce a new output type which holds both the outgoing data-flows to other agents and the observable state the agent wants to reveal to the outside world:

```
data AgentOut o d = AgentOut
  { aoData        :: [DataFlow d]
  , aoObservable  :: o }
```

Note that by making the observable state explicit in the types we give the agent further control of what it can reveal to the outside world which allows an even stronger separation between the agents internal state and what the agent wants the world to see.

Now we can then generalise the agents signal functions to the following type:

```
type Agent o d = SF (AgentIn d) (AgentOut o d)
```

For our SIR implementation we need concrete types, so we need to define what the type parameters o and d are. For d we use an ADT as contact-message. As type of the observable state we use the existing SIR state. Now we can define the type synonyms for our SIR implementation:

```
data SIRMsg      = Contact SIRState deriving Eq
type SIRAgentIn  = AgentIn SIRMsg
type SIRAgentOut = AgentOut SIRState SIRMsg
type SIRAgent    = Agent SIRState SIRMsg
```

Next we are going to re-implement the agent-behaviour:

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

The initial behaviour is the same as previously but it now takes a list of agent ids as additional parameter. With data-flow we need to know the ids of the agents we are communicating with - we need to know our neighbourhood, or seen differently: we need to have access to the environment we are situated in. In our case our environment is a fully connected read-only network in which all agents know all other agents. The easiest way of representing a fully connected network (complete graph) is simply using a list. The implementation of the recovered agent is still the same, its just a sink which ignores the environment and the random-number generator.

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

Note that instead of returning just a SIR state now the output of an agents signal function is of type *AgentOut*:

```
agentOut :: o -> AgentOut o d
agentOut o = AgentOut {
    aoData       = []
  , aoObservable = o }
```

The behaviour of the infected agent now explicitly ignores the environment which was not apparent previously on this level:

```
infectedAgent :: RandomGen g => g -> SIRAgent
infectedAgent g = switch infected (const recoveredAgent)
  where
    infected :: SF SIRAgentIn (SIRAgentOut, Event ())
    infected = proc ain -> do
      recEvt <- occasionally g illnessDuration () -< ()
      let a  = event Infected (const Recovered) recEvt -- event is equivalent to maybe
```

```
        ao = respondToContactWith Infected ain (agentOut a)
    returnA -< (ao, recEvt)
```

The implementation of the infected agent is essentially the same as previously but it now needs to reply to incoming contact data-flows with an "Infected" reply. This makes the difference to the previous step very explicit: in the data-flow approach agents now make explicit contact with each other which means that the susceptible agent sends out contact data-flows to which only infected agents need to reply. Note that at the moment of recovery the agent can still infect others because it will still reply with Infected. The response mechanism is implemented in *respondToContactWith*:

```
respondToContactWith :: SIRState -> SIRAgentIn -> SIRAgentOut -> SIRAgentOut
respondToContactWith state = onData respondToContactWithAux
  where
    respondToContactWithAux :: DataFlow SIRMsg -> SIRAgentOut -> SIRAgentOut
    respondToContactWithAux (senderId, Contact _) = dataFlow (senderId, Contact state)

onData :: (DataFlow d -> acc -> acc) -> AgentIn d -> acc -> acc
onData df ai a = foldr df a (aiData ai)

dataFlow :: DataFlow d -> AgentOut o d -> AgentOut o d
dataFlow df ao = ao { aoData = df : aoData ao }
```

Note that the order of data-packages in a data-flow is not specified and must not matter as it happens virtually at the same time, thus we always append at the front of the outgoing data-flow list.

Lets look at the susceptible agent behaviour. As already mentioned before, the feedback interaction between agents works now very explicit due to the data-flow but needs a different approach in our implementation:

```
susceptibleAgent :: RandomGen g => g -> [AgentId] -> SIRAgent
susceptibleAgent g ais = switch (susceptible g) (const (infectedAgent g))
  where
    susceptible :: RandomGen g => g -> SF SIRAgentIn (SIRAgentOut, Event ())
    susceptible g0 = proc ain -> do
      rec
        g <- iPre g0 -< g'
        let (infected, g') = runRand (gotInfected infectivity ain) g

      if infected
        then returnA -< (agentOut Infected, Event ())
        else (do
          makeContact <- occasionally g (1 / contactRate) () -< ()
          contactId   <- drawRandomElemSF g                  -< ais
          let ao = agentOut Susceptible
          if isEvent makeContact
            then returnA -< (dataFlow (contactId, Contact Susceptible) ao, NoEvent)
            else returnA -< (ao, NoEvent))

gotInfected :: RandomGen g => Double -> SIRAgentIn -> Rand g Bool
gotInfected p ain = onDataM gotInfectedAux ain False
  where
    gotInfectedAux :: RandomGen g => Bool -> DataFlow SIRMsg -> Rand g Bool
    gotInfectedAux False (_, Contact Infected) = randomBoolM p
    gotInfectedAux x _ = return x

onDataM :: Monad m => (acc -> DataFlow d -> m acc) -> AgentIn d -> acc -> m acc
onDataM df ai acc = foldM df acc (aiData ai)
```

Again the implementation is very similar to the previous step with the fundamental difference being how contacts are made and how infections occur. First the agent checks if it got infected. This happens if an infected agent replies to the susceptible agents contact *and* the susceptible agent got infected with the given probability. Note that *gotInfected* runs in the Random Monad which we run using *runRand* and the random-number generator. To update our random-number generator to the changed one, we use the *rec* keyword of the Arrow notation [30], which allows us to refer to a variable before it is defined. In combination with *iPre* we introduced a local state - the random-number generator - which changes in every step. The function *iPre* delays the input by one time-step, returns it in the next time-step and is initialized with an initial value. If the agent got infected, it simply returns an AgentOut with Infected as observable state and a switching event which indicates the switch to infected behaviour. If the agent is not infected it draws from *occasionally* to determine if it should make contact with a random agent. In case it should make contact it simply sends a data-package with the contact susceptible data to the receiver - note that only an infected agent will reply.

Stepping the simulation now works a little bit different as the input/output types have changed and we need to collect and distribute the data-flow amongst the agents:

```
stepSimulation :: [SIRAgent] -> [SIRAgentIn] -> SF () [SIRAgentOut]
stepSimulation sfs ains =
    dpSwitch
      (\_ sfs' -> (zip ains sfs'))
      sfs
      (switchingEvt >>> notYet)
      stepSimulation
  where
    switchingEvt :: SF ((), [SIRAgentOut]) (Event [SIRAgentIn])
    switchingEvt = proc (_, aos) -> do
      let ais      = map aiId ains
          aios     = zip ais aos
          nextAins = distributeData aios
      returnA -< Event nextAins
```

The distribution of the data-flows happens in the *distributeData* function of *switchingEvt* and is then passed on to the continuation-generation function as previously. Note that due to lack of space we can't give an implementation of *distributeData* but we provide the type.

```
distributeData :: [(AgentId, AgentOut o d)] -> [AgentIn d]
```

The difference is that it now creates a list of AgentIn for the next step instead of a list of all the agents SIR states of the previous step. Again the continuation-generation function recursively returns the *stepSimulation* signal function. The pairing function of *dpSwitch* is now slightly more straightforward as it just pairs up the AgentIn with its corresponding signal function.

6.3.2 Discussion. It seems that by introducing the data-flow mechanism we have increased complexity but we have gained a lot as well. Data-flows make the feedback between agents explicit and gives the agents full control over the data which is revealed to other agents. This also makes the fact even more explicit, that we cannot fix the connections between the agents already at compile time e.g. by connecting SFs which is done in many Yampa applications [27], [9], [28] because agents interact with each other randomly. One can look at the data-flow mechanism as a kind of messaging but there are fundamental differences. Messaging almost always comes up as an approach to managing concurrency and involves stateful message-boxes which can be checked an emptied by the receivers - this is not the case with the data-flow mechanism which behaves indeed as a flow where data is not stored in a message box but is only present in the current simulation-step and if ignored by the agent will be gone in the next step. Also by distinguishing between the internal and the observable state of the agent, we give the agent even more control of what is visible to the outside world. So far we have an acceptible implementation of an agent-based SIR approach. The next steps focus on introducing more concepts and generalising our implementation so far. 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 [33] which allows to do FRP but with a monadic context.

6.4 Generalising to Monadic Stream Functions

A part of the library Dunai is BearRiver, a wrapper which re-implements Yampa on top of Dunai, which should allow us to easily replace Yampa with MSFs. This will enable us to run arbitrary monadic computations in a signal function, which we will need in the next step when adding an environment.

6.4.1 Identity Monad. We start by making the transition to BearRiver by simply replacing Yampas signal function by BearRivers which is the same but takes an additional type parameter *m* indicating the monad. If we replace this type-parameter with the identity monad we should be able to keep the code exactly the same, except from a few type-declarations, because BearRiver re-implements all necessary functions we are using from Yampa [4]. We start by re-defining our general agent signal function, introducing the monad (stack) our SIR implementation runs in and the agents signal function:

```
type Agent m o d = SF m (AgentIn d) (AgentOut o d)
type SIRMonad    = Identity
type SIRAgent    = Agent SIRMonad SIRState SIRMsg
```

We also have to add the *SIRMonad* to the existing *stepSimulation* type-declarations and we are nearly done. The function *embed* for running the simulation is not provided by BearRiver but by Dunai which has important implications. Dunai does not know about time in MSFs, which is exactly what BearRiver builds on top of MSFs. It does so by adding a ReaderT Double which carries the Δt. This means that *embed* returns a computation in the ReaderT Double Monad which we need to run explicitly using *runReaderT*. This then results in an identity computation which we simply peel away using *runIdentity*. Here is the complete code of *runSimulation*:

```
runSimulation :: RandomGen g => g -> Time -> DTime -> [(AgentId, SIRState)] -> [[SIRState]]
runSimulation g t dt as = map (map aoObservable) aoss
  where
    steps         = floor (t / dt)
    dts           = replicate steps ()
    n             = length as
    (rngs, _)     = rngSplits g n []
    ais           = map fst as
    sfs           = map (\ (g', (_, s)) -> sirAgent g' ais s) (zip rngs as)
    ains          = map (\ (aid, _) -> agentIn aid) as
    aossReader    = embed (stepSimulation sfs ains) dts
    aossIdentity  = runReaderT aossReader dt
    aoss          = runIdentity aossIdentity
```

Note that embed does not take a list of Δt any more but simply a list of inputs for each step to the top level signal function.

[4]This was not quite true at the time of writing this paper, where *occasionally*, *noiseR* and *dpSwitch* were missing. We simply forked the project from GitHub and implemented these functions in our own branch.

6.4.2 Random Monad. Using the Identity Monad does not gain us anything but it was a first step towards a more general solution. Our next step is to replace the Identity Monad by the Random Monad which will allow us to get rid of the RandomGen arguments to our functions and run the whole simulation within the Random Monad *again* just as we started but now with the full features functional reactive programming. We start by re-defining the SIRMonad and SIRAgent:

```
type SIRMonad g = Rand g
type SIRAgent g = Agent (SIRMonad g) SIRState SIRMsg
```

The question is now how to access this Random Monad functionality within the MSF context. For the function *occasionally*, there exists a monadic pendant *occasionallyM* which requires a MonadRandom type-class. Because we are now running within a MonadRandom instance we simply replace *occasionally* with *occasionallyM*. Running *gotInfected* is now much easier. Using the function *arrM* of Dunai allows us to run a monadic action in the stack as an arrow. We then directly run *gotInfected* by lifting it into the Random Monad. This can be seen in the susceptible agent running in the random monad SF:

```
susceptibleAgent :: RandomGen g => [AgentId] -> SIRAgent g
susceptibleAgent ais = switch susceptible(const (infectedAgent))
  where
    susceptible :: RandomGen g => SF (SIRMonad g) SIRAgentIn (SIRAgentOut, Event ())
    susceptible = proc ain -> do
      infected <- arrM (lift . gotInfected infectivity) -< ain
      if infected
        then returnA -< (agentOut Infected, Event ())
        else (do
          makeContact <- occasionallyM (1 / contactRate) () -< ()
          contactId   <- drawRandomElemSF                  -< ais
          let ao = agentOut Susceptible
          if isEvent makeContact
            then returnA -< (dataFlow (contactId, Contact Susceptible) ao, NoEvent)
            else returnA -< (ao, NoEvent))
```

Note also that *drawRandomElemSF* doesn't take a random number generator as well as it has been reimplemented to make full use of the MonadRandom in the stack:

```
drawRandomElemS :: MonadRandom m => SF m [a] a
drawRandomElemS = proc as -> do
  r <- getRandomRS ((0, 1) :: (Double, Double)) -< ()
  let len = length as
  let idx = fromIntegral len * r
  let a =  as !! floor idx
  returnA -< a
```

Instead of *noiseR* which requires a RandomGen, it makes use of Dunai *getRandomRS* stream function which simply runs *getRandomR* in the MonadRandom.

Finally because our innermost monad is now the Random Monad instead of the Identity we run it by *evalRand*:

```
aossReader = embed (stepSimulation sfs ains) dts
  aossRand = runReaderT aossReader dt
      aoss = evalRand aossRand g
```

6.4.3 Discussion. By making the transition to MSFs we can now stack arbitrary number of monads. As an example we could add a StateT monad on the type of AgentOut which would allow to conveniently manipulate the AgentOut e.g. in case where one sends more than one message or the construction of the final AgentOut is spread across multiple functions which allows easy composition. When implementing this, one needs to replace the dpSwitch with an individual

(a) von Neumann (b) Moore

Fig. 5. Common neighbourhoods in discrete 2D environments of Agent-Based Simulation.

implementation in which one runs the state monad isolated for each agent. We could even add the IO monad if our agents require arbitrary IO e.g. reading/writing from files or communicating over TCP/IP. Although one could run in the IO Monad, one should not do so as one would loose all static guarantees about the reproducibility of the simulation. In ABS we need deterministic behaviour under all circumstances where repeated runs with the same initial conditions, including the random-number generator, should result in the same dynamics. If we allow the use of the IO Monad we loose the ability to guarantee the reproducibility at compile-time even if the agents never use IO facilities and just run in the IO for printing debug messages. So far making the transition to MSFs does not seem as compelling as making the move from the Random Monad to FRP in the beginning. Running in the Random Monad within FRP is convenient but we could achieve the same with passing RandomGen around as we already demonstrated. In the next step we introduce the concept of a read/write environment which we realise using a StateT monad. This will show the real benefit of the transition to MSFs as without it, implementing a general environment access would be quite cumbersome.

6.5 Adding an environment

In this step we will add an environment in which the agents exist and through which they interact with each other. This is a fundamental different approach to agent-agent interaction but is as valid as the interactions in the previous steps. In ABS agents are often situated within a discrete 2D environment [37], [15], [14] which is simply a finite NxM grid with either a Moore or von Neumann neighbourhood (see Figure 5). Agents are either static or can move freely around with cells allowing either single or multiple occupants.

We can directly map the SIR model to a discrete 2D environment by placing the agents on a corresponding 2D grid with an unrestricted neighbourhood. The behaviour of the agents is the same but they select their neighbours directly from the environment using the provided neighbourhood. Also instead of using data-flow to communicate, agents now communicate through the environment by revealing their current state to their neighbours by placing it on their cell. Agents can read the states of all their neighbours which tells them if a neighbour is infected or not. This allows us to implement the infection mechanism as in the beginning. For purposes of a more interesting and real approach which makes use of the features of agents within an environment, we restrict the neighbourhood to Moore (Figure 5b).

6.5.1 Implementation. We start by defining our discrete 2D environment for which we use an indexed two dimensional array. In each cell the agents will store their current state, thus we use the *SIRState* as type for our array data:

```
type Disc2dCoord = (Int, Int)
type SIREnv      = Array Disc2dCoord SIRState
```

Next we redefine our monad stack and agent signal function. We use a StateT transformer on top of our Random Monad from step 4 with the previously defined SIREnv as type for the state. Our agent signal function now has only unit input and output type as we removed the data-flow mechanism for reasons of clarity. This also indicates through the types that the actions of the agents are only visible in side-effects through the monad stack they are running in.

```
type SIRMonad g = StateT SIREnv (Rand g)
type SIRAgent g = SF (SIRMonad g) () ()
```

Instead of having a unique agent id an agent is now initialised through its coordinates in the environment and its initial state.

```
sirAgent :: RandomGen g => Disc2dCoord -> SIRState -> SIRAgent g
sirAgent c Susceptible = susceptibleAgent c
sirAgent c Infected    = infectedAgent c
sirAgent _ Recovered   = recoveredAgent
```

Again the recovered agent behaviour is the shortest one:

```
recoveredAgent :: RandomGen g => SIRAgent g
recoveredAgent = arr (const ())
```

The implementation of a susceptible agent is now a bit different and a mix between previous steps. Instead of using data-flows the agent directly queries the environment for its neighbours and randomly selects one of them. The remaining behaviour is similar:

```
susceptibleAgent :: RandomGen g => Disc2dCoord -> SIRAgent g
susceptibleAgent coord = switch susceptible (const (infectedAgent coord))
  where
    susceptible :: RandomGen g => SF (SIRMonad g) () ((), Event ())
    susceptible = proc _ -> do
      makeContact <- occasionallyM (1 / contactRate) () -< ()
      if not (isEvent makeContact)
        then returnA -< ((), NoEvent)
        else (do
          e <- arrM_ (lift get) -< ()
          let ns = neighbours e coord agentGridSize moore
          s <- drawRandomElemS -< ns
          case s of
            Infected -> do
              infected <- arrM_ (lift $ lift $ randomBoolM infectivity) -< ()
              if infected
                then (do
                  arrM (put . changeCell coord Infected) -< e
                  returnA -< ((), Event ()))
                else returnA -< ((), NoEvent)
            _        -> returnA -< ((), NoEvent))
```

Querying the neighbourhood is done using the *neighbours* function. It takes the environment, the coordinate for which to query the neighbours for, the dimensions of the 2D grid and the neighbourhood information and returns the data of all neighbours it could find. Note that on the edge of the environment, it could be the case that fewer neighbours than provided in the neighbourhood information will be found due to clipping.

```
neighbours :: SIREnv -> Disc2dCoord -> Disc2dCoord -> [Disc2dCoord] -> [SIRState]
neighbours e (x, y) (dx, dy) n = map (e !) nCoords'
  where
    nCoords  = map (\ (x', y') -> (x + x', y + y')) n
    nCoords' = filter (\ (x, y) -> x >= 0 && y >= 0 &&
                                  x <= (dx - 1) && y <= (dy - 1)) nCoords
```

```
neumann :: [Disc2dCoord]
neumann = [ top, left, right, bottom ]

moore :: [Disc2dCoord]
moore = [ topLeft,    top,    topRight,
          left,               right,
          bottomLeft, bottom, bottomRight ]

topLeft :: Disc2dCoord
topLeft  = (-1, -1)
top      = ( 0, -1)
topRight = ( 1, -1)
...
```

The behaviour of an infected agent is nearly the same with the difference that upon recovery the infected agent updates its state in the environment from Infected to Recovered.

```
infectedAgent :: RandomGen g => Disc2dCoord -> SIRAgent g
infectedAgent coord = switch infected (const recoveredAgent)
  where
    infected :: RandomGen g => SF (SIRMonad g) () ((), Event ())
    infected = proc _ -> do
      recovered <- occasionallyM illnessDuration () -< ()
      if isEvent recovered
        then (do
          e <- arrM (\_ -> lift get) -< ()
          arrM (\e -> put (changeCell coord Recovered e)) -< e
          returnA -< ((), Event ()))
        else returnA -< ((), NoEvent)
```

Running the simulation is now slightly different as we have an initial environment and also need to peel away the StateT transformer:

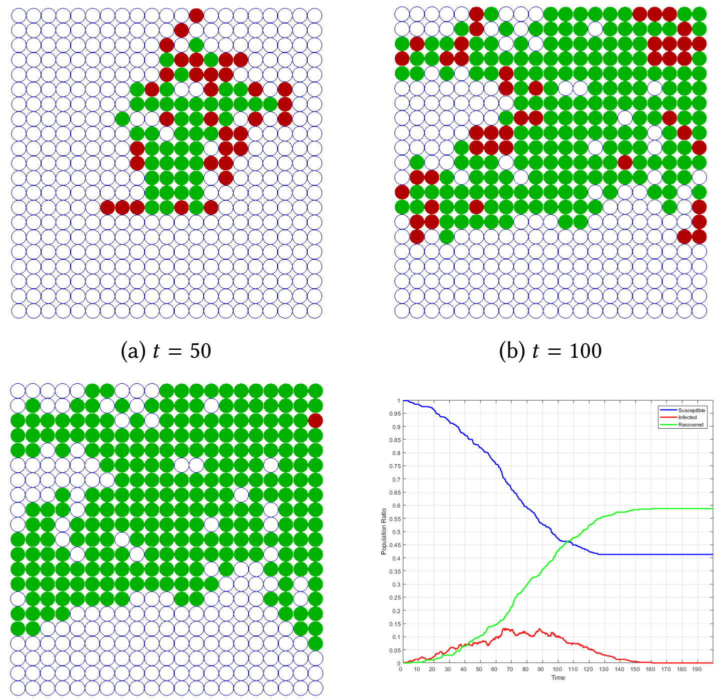
```
runSimulation :: RandomGen g => g -> Time -> DTime -> SIREnv -> [(Disc2dCoord, SIRState)] -> [SIREnv]
runSimulation g t dt e as = evalRand esRand g
  where
    steps    = floor (t / dt)
    dts      = replicate steps ()
    sfs      = map (uncurry sirAgent) as
    esReader = embed (stepSimulation sfs) dts
    esState  = runReaderT esReader dt
    esRand   = evalStateT esState e
```

As initial state we use the initial environment and instead of returning agent states we simply return a list of environments, one for each step. The agent states can then be extracted from each environment.

Due to the different approach of returning the SIREnv in every step, we implemented our own MSF:

```
stepSimulation :: RandomGen g => [SIRAgent g]-> SF (SIRMonad g) () SIREnv
stepSimulation sfs = MSF (\_ -> do
  res <- mapM (`unMSF` ()) sfs
  let sfs' = fmap snd res
  e <- get
  let ct = stepSimulation sfs'
  return (e, ct))
```

6.5.2 Results. We implemented rendering of the environments using the gloss library which allows us to cycle arbitrarily through the steps and inspect the spreading of the disease over time visually as in Figure 6.



(a) $t = 50$ (b) $t = 100$

(c) $t = 160$ (d) Dynamics over time

Fig. 6. Simulating the agent-based SIR model on a 21x21 2D grid with Moore neighbourhood and a single infected agent at the center and same SIR parameters. Simulation run until $t = 200$ with fixed $\Delta t = 0.1$. Last infected agent recovers shortly after $t = 160$. The susceptible agents are rendered as blue hollow circles for better contrast.

Note that the dynamics of the spatial SIR simulation which are seen in Figure 6d look quite different from the SD dynamics of Figure 2. This is due to a much more restricted neighbourhood which results in far fewer infected agents at a time and a lower number of recovered agents at the end of the epidemic, meaning that fewer agents got infected overall.

6.5.3 Discussion. At first the environment approach might seem a bit overcomplicated and one might ask what we have gained by using an unrestricted neighbourhood where all agents can contact all others. The real win is that we can introduce arbitrary restrictions on the neighbourhood as shown using the Moore neighbourhood. Of course the environment is not restricted to a discrete 2D grid and can be anything from a continuous N-dimensional space to a complex network - one only needs to change the type of the StateT monad and provide corresponding neighbourhood querying functions. The ability to place the heterogeneous agents in a generic environment is also the fundamental advantage of an agent-based over the SD approach and allows to simulate much more realistic scenarios. Note that for reasons of clarity we have removed the data-flow approach from this implementation which results in the unit-types of input and output. In a full blown agent-based simulation library we would combine both approaches.

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

(1) Passive read-only - implemented in the previous steps, where the environment never changes and is passed as static information, e.g. list of neighbours, to each agent.
(2) Passive read/write - implemented in this step. The environment itself is not modelled as an active process but just as shared data which can be accessed and manipulated by the agents.
(3) Active read-only - can be implemented by adding an environment agent which broadcasts changes in the environment to all agents using the data-flow mechanism.
(4) Active read/write - can be implemented as in this step plus adding an environment agent which reads/writes the environment e.g. regrowing some resources.

Attempting to introduce an active/passive read/write environment to the Yampa implementation would be quite cumbersome. A possible solution could be to add a type-parameter e which captures the type of the environment and then pass it in through the input and allow it to be returned in the output of an agent signal function. We would then end up with n copies of the environment - one for each agent - which we need to fold back into a single environment. Having an active environment complicates things even further. All these problems are not an issue when using MSFs with a StateT which is a compelling example for making the transition to the more general MSFs. The convenient thing is that although conceptually all agents act at the same time, technically by using *mapM* in stepSimulation they are run after another which also serialises the environment access which gives every agent exclusive read/write access while it is active.

6.6 Further Steps

6.6.1 Agent-Transactions. We have implemented synchronous interactions, which we termed agent-transactions in an additional step which we had to omit due to lack of space. Agent-transactions are necessary when an arbitrary number of interactions between two agents need to happen instantaneously without time-lag. The use-case for this are price negotiations between multiple agents where each pair of agents needs to come to an agreement in the same time-step. In object-oriented programming, the concept of synchronous communication between agents is implemented directly with method calls. We solved it pure functionally by running the signal functions of the transacting agent pair as often as their protocol requires but with $\Delta t = 0$, which indicates the instantaneous character of agent-transactions.

6.6.2 Dynamic Agent creation. In the SIR model, the agent population stays constant - agents don't die and no agents are created during simulation - but some simulations [15] require dynamic agent destruction and creation. We can easily add and remove agents signal functions in the recursive switch after each time-step. The only problem is that creating new agents requires unique agent ids but with the transition to MSFs we can add a monadic context which allows agents to draw the next unique agent id when they create a new agent.

7 CONCLUSIONS

Our approach is radically different from traditional approaches in the ABS community. First it builds on the already quite involved FRP paradigm. Second, due our hybrid approach, it forces one to think properly of time-semantics of the model, how to sample it, how small Δt should be and whether one needs super-sampling or not and if yes how many samples one should take. Third it requires to think about agent-interactions in a new way instead of being just method-calls. Also due to the underlying nature and motivation of FRP and Yampa, agents can be seen as signals which are generated and consumed by a signal function. If an agent does not change, the output signal should be constant and if the agent changes e.g. by sending a message or changing its state, the output signal should change as well. A dead agent then should have no signal at all. The agents in all but the very first step of our approach are completely time-dependent which means they will not act when time does not advance. Thus when running our simulation with $\Delta t = 0$ the dynamics stay constant and won't change.

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 (e.g. file access, networking, threading, random-number re-seeding). 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 to come from e.g. the current system time, which would require to run within the IO Monad. This means that after initialising the agents, which *could* run in the IO Monad, the simulation itself runs completely deterministicly. This is also ensured through 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 [34]. Also by using FRP we gain all the benefits from it and can use research on testing, debugging and exploring FRP systems [34], [31].

Also we have built a full blown library for implementing pure agent-based simulations which implements and combines all the presented techniques including agent-transactions. As examples we implemented a number of well known agent-based models with various complexity, including the seminal and highly complex Sugarscape model [15] and Schelling Segregation [37]. Compared to object-oriented implementations, the pure functional ones are quite concise and highly expressive. This shows that from an engineering point-of-view a pure functional approach to ABS is as well suited as object-oriented techniques [5].

Our approach is inherently time-driven where the system is sampled with fixed Δt. The other fundamental way to implement an ABS in general, is to follow an event-driven approach [26] which is based on the theory of discrete-event simulation [52]. In such an approach the system is not sampled in fixed Δt but advanced as events occur where the system stays constant in between events. Depending on the model, an event-driven approach may be more suitable and is more natural to express the requirements of the model. How to derive a purely functional approach to an event-driven approach to ABS we are leave for further research.

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 [24]. With the need to sample the system with high frequency, this issue gets worse. The reason for this is that we don't have in-place updates of data structures and make no use of references. This results in lots of copying which is simply not necessary in the imperative languages with implicit effects. Also it is much more difficult to reason about time and space in our approach.

Despite the strengths and benefits we get by leveraging on FRP, there are also weakness to it which show up in our approach as well. The authors [39] show that the type-system of Yampa does not prevent one from run-time errors and infinite loops, as FRP is sacrificing the safety of FP for sake of expressiveness. Amongst others they showed that well-formed feedback does depend on the programmer and cannot be guaranteed at compile time through the type system. Feedback is an inherent feature of ABS where agents update their state at time t+1 depending on time t.

[5]The code is freely available at https://github.com/thalerjonathan/chimera and we plan on releasing it on Hackage in the future.

This is only visible in Section 6.2 where we use feedback to update the random-number generator (rec/iPre/feedback makes the inherently feedback nature of ABS very explicit) but in more complex ABS models with a more complex state than in the SIR model, feedback is the core feature to keep and update the state of an agent. Also a chain of switches could result in an infinite loop - this cannot be checked at compile time and needs to be carefully designed by the programmer and results sometimes in popping up of operational details (e.g. the need to use $>>>$ $notYet$ in parallel switches for stepping the simulation). The authors of [39] proposed dependent types as a remedy for these shortcomings and discuss a dependently typed implementation of the core of Yampa in Agda.

Also having a two layer (arrows and pure functions) language in Yampa [22] and three a layer (arrows, monadic and pure functions) language in Dunai / BearRiver adds expressivity and power but can make things quite complex already in the simple SIR example. Fortunately with a more complex model the complexity in this context does not increase - in the end it is the price we need to pay for the high expressivity which functions like *occasionally* provide.

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. Due to the lack of mutable state and method calls this is naturally much more difficult to achieve in pure functional programming. Our current solution to the problem is the data-flow mechanism as implemented in section 6.3. We have to admit that it is rather ad-hoc and may need some refinement and generalisation, also it is unclear if there are not better/more flexible mechanisms of agent communication and interaction in pure functional programming. We 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 came 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. But our research was not in vain as we see it as an intermediary step towards using dependent types together with the pure functional approach. Moving to dependent types would pose a unique benefit over the object-oriented approach and would allow us to express and guarantee properties at compile time which is not possible with imperative approaches. We leave this for further research.

8 FURTHER RESEARCH

We see this paper as an intermediary and necessary step towards dependent types for which we first need to understand the potentials and limitations of a non-dependently typed pure functional approach in Haskell. Dependent types is extremely promising in functional programming as they allow us to express stronger guarantees about the correctness of programs and go as far as formulating programs and types as constructive proofs [48] which must be total by definition [46], [4], [3], [36]. 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 setting. 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 [7], [8] 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 dependent type way. This would allow the ABS community for the first time to reason about a proper formalisation of a model. We plan to implement a total and terminating implementation of our approach which would be a formal proof-by-construction that the agent-based approach of the SIR model terminates after a finite number of steps.

ACKNOWLEDGMENTS

The authors would like to thank I. Perez, H. Nilsson, H. Vollbrecht, J. Greensmith, S. Venkatesan and J. Hey for constructive comments and valuable discussions.

REFERENCES

[1] Helen Abbey. 1952. An Examination of the Reed-Frost Theory of Epidemics. *Human Biology* 24, 3 (Sept. 1952), 201. http://ezproxy.nottingham.ac.uk/login?url=https://search.proquest.com/docview/1301828990?accountid=8018

[2] Aslam Ahmed, Julie Greensmith, and Uwe Aickelin. 2013. Variance in System Dynamics and Agent Based Modelling Using the SIR Model of Infectious Disease. *arXiv:1307.2001 [cs]* (July 2013). http://arxiv.org/abs/1307.2001 arXiv: 1307.2001.

[3] Thorsten Altenkirch, Nils Anders Danielsson, Andres LÃűh, 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

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

[5] Nikolaos Bezirgiannis. 2013. *Improving Performance of Simulation Software Using HaskellâĂŹs Concurrency & Parallelism*. Ph.D. Dissertation. Utrecht University - Dept. of Information and Computing Sciences.

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

[7] 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

[8] Edwin Brady. 2017. *Type-Driven Development with Idris*. Manning Publications Company. Google-Books-ID: eWzEjwEACAAJ.

[9] 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

[10] D. Dawson, P. O. Siebers, and Tuong Manh Vu. 2014. Opening pandora's box: Some insight into the inner workings of an Agent-Based Simulation environment. In *2014 Federated Conference on Computer Science and Information Systems*. 1453–1460. https://doi.org/10.15439/2014F335

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

[12] Richard H. Enns. 2010. *It's a Nonlinear World* (1st ed.). Springer Publishing Company, Incorporated.

[13] Joshua M. Epstein. 2012. *Generative Social Science: Studies in Agent-Based Computational Modeling*. Princeton University Press. Google-Books-ID: 6jPiuMbKKJ4C.

[14] Joshua M. Epstein. 2014. *Agent_Zero: Toward Neurocognitive Foundations for Generative Social Science*. Princeton University Press. Google-Books-ID: VJEpAgAAQBAJ.

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

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

[17] 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. http://link.springer.com/chapter/10.1007/978-3-540-44833-4_6 DOI: 10.1007/978-3-540-44833-4_6.

[18] 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

[19] 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

[20] 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

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

[22] Alan Jeffrey. 2013. Causality for Free!: Parametricity Implies Causality for Functional Reactive Programs. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification (PLPV '13)*. ACM, New York, NY, USA, 57–68. https://doi.org/10.1145/2428116.2428127

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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

[28] Henrik Nilsson and Ivan Perez. 2014. Declarative Game Programming: Distilled Tutorial. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming (PPDP '14)*. ACM, New York, NY, USA, 159–160. https://doi.org/10.1145/2643135.2643160

[29] 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.

[30] 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

[31] 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

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

[33] Ivan Perez, Manuel BÃďrenz, 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

[34] 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

[35] 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

[36] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics.* https://homotopytypetheory.org/book, Institute for Advanced Study.

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

[38] Oliver Schneider, Christopher Dutchyn, and Nathaniel Osgood. 2012. Towards Frabjous: A Two-level System for Functional Reactive Agent-based Epidemic Simulation. In *Proceedings of the 2Nd ACM SIGHIT International Health Informatics Symposium (IHI '12)*. ACM, New York, NY, USA, 785–790. https://doi.org/10.1145/2110363.2110458

[39] 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

[40] 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.

[41] P. O. Siebers, C. M. Macal, J. Garnett, D. Buxton, and M. Pidd. 2010. Discrete-event simulation is dead, long live agent-based simulation! *Journal of Simulation* 4, 3 (Sept. 2010), 204–210. https://doi.org/10.1057/jos.2010.14

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

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

[44] 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

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

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

[47] 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. http://link.springer.com/chapter/10.1007/978-3-319-05579-4_47 DOI: 10.1007/978-3-319-05579-4_47.
[48] Philip Wadler. 2015. Propositions As Types. *Commun. ACM* 58, 12 (Nov. 2015), 75–84. https://doi.org/10.1145/2699407
[49] 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
[50] Gerhard Weiss. 2013. *Multiagent Systems*. MIT Press. Google-Books-ID: WY36AQAAQBAJ.
[51] Michael Wooldridge. 2009. *An Introduction to MultiAgent Systems* (2nd ed.). Wiley Publishing.
[52] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. 2000. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press. Google-Books-ID: REzmYOQmHuQC.

A EMULATING SYSTEM DYNAMICS

The introduction of data-flows in section 6.3 allows us to emulate the system dynamics (SD) approach because we can now express a system with parallel continuous-time flows between the stocks and flows. Each stock $S(t)$, $I(t)$, $R(t)$ and each flow $infectionRate$, $recoveryRate$ is implemented as an agent with a fixed agent id. The connections between them are implemented using the previously introduced data-flow mechanism. We start by refining the types for our SIR implementation:

```
type SDMsg      = Double
type SDAgentIn  = AgentIn SDMsg
type SDObs      = Maybe Double
type SDEntity   = Agent SDObs SDMsg
type SDEntityId = AgentId

totalPopulation :: Double
totalPopulation = 1000

infectedCount :: Double
infectedCount = 1
```

The message-data is now a plain Double and the observable data has been changed to a *Maybe* Double: instead of discrete agent-states we are dealing now with stocks and flows which are aggregates represented by continuous values. Note that we use a Maybe type as flows only connect stocks and transform their values but don't have any observable state themselves. Note also that the population size and number of infected is specified now as Double as we are dealing with continuous aggregates.

We give hard-coded agent ids to our stocks and flows. This allows then for setting up hard-coded connections between them at compile time.

```
susceptibleStockId :: SDEntityId
susceptibleStockId = 0

infectiousStockId :: SDEntityId
infectiousStockId = 1

recoveredStockId :: SDEntityId
recoveredStockId = 2

infectionRateFlowId :: SDEntityId
infectionRateFlowId = 3

recoveryRateFlowId :: SDEntityId
recoveryRateFlowId = 4
```

Next we give the implementation of the infectious stock (the implementations of the susceptible and recovered stock work in a similar way and are left as an easy exercise to the reader):

```
infectiousStock :: Double -> SDEntity
infectiousStock initValue = proc ain -> do
  let infectionRate = flowInFrom infectionRateFlowId ain
      recoveryRate  = flowInFrom recoveryRateFlowId ain

  stockValue <- (initValue+) ^<< integral -< (infectionRate - recoveryRate)

  let ao   = agentOut (Just stockValue)
      ao'  = dataFlow (infectionRateFlowId, stockValue) ao
      ao'' = dataFlow (recoveryRateFlowId, stockValue) ao'

  returnA -< ao''
```

The stock receives flows from both the infection-rate and recovery-rate flow using the function *flowInFrom* (see below). Then the current stock value is calculated using the *integral* function of Yampa with an initial value added which are the initially infected people. The integral primitive of Yampa integrates the fed in data over time using the rectangle rule which means it simply multiplies the input values by the current Δt and accumulates them. Note that we can directly express the SD equation using Yampas DSL for continuous-time systems. The current stock value is then set as the observable value of the stock and sent to the infection- and recovery-rate flows. For convenience we implemented an additional function *flowInFrom* which returns the first value sent from the corresponding agent id or 0.0 if none was sent.

```
flowInFrom :: SDEntityId -> SDAgentIn -> Double
flowInFrom senderId ain = firstValue dsFiltered
  where
    dsFiltered = filter ((==senderId) . fst) (aiData ain)

    firstValue :: [AgentData SDMsg] -> Double
    firstValue [] = 0.0
    firstValue ((_, v) : _) = v
```

The *infectionRate* flow is implemented as follows (the implementations of the recovery-rate flow works in a similar way and is left as an easy exercise to the reader):

```
infectionRateFlow :: SDEntity
infectionRateFlow = proc ain -> do
  let susceptible = flowInFrom susceptibleStockId ain
      infectious  = flowInFrom infectiousStockId ain

      flowValue   = (infectious * contactRate * susceptible * infectivity) / totalPopulation

      ao          = agentOut Nothing
      ao'         = dataFlow (susceptibleStockId, flowValue) ao
      ao''        = dataFlow (infectiousStockId, flowValue) ao'

  returnA -< ao''
```

Instead of integrating a value over time a stock just transforms incoming values from the connected stocks - in this case the susceptible and infectious stocks. Note again how directly we can express the formula for the infection rate.

When running the simulation one must make sure to use a small enough Δt as *integral* of Yampa is implemented using the rectangle rule which leads to considerable numerical errors with large Δt. Figure 2 was created with this SD emulation for which we used $\Delta t = 0.01$.

B AGENT TRANSACTIONS

Imagine two agents A and B want to engage in a bartering process where agent A, is the seller who wants to sell an asset to agent B who is the buyer. Agent A sends Agent B a sell offer depending on how much agent A values this asset. Agent B receives this sell offer, checks if the price satisfies its utility, if it has enough wealth to buy the asset and replies with either a refusal or its own price offer. Agent A then considers agent Bs offer and if it is happy it replies to agent B with an acceptance of the offer, removes the asset from its inventory and increases its wealth. Agent B receives this acceptance offer, puts the asset in its inventory and decreases its wealth (note that this process could involve a potentially arbitrary number of steps without loss of generality). We can see this behaviour as a kind of multi-step transactional behaviour because agents have to respect their budget constraints which means that they cannot spend more wealth or assets than they have. This implies that they have to 'lock' the asset and the amount of cash they are bartering about during the bartering process. If both come to an agreement they will swap the asset and the cash and if they refuse their offers they have to 'unlock' them. In classic OO implementations it is quite easy to implement this as normally only one agent is active at a time due to sequential (discrete event scheduling approach) scheduling of the simulation. This allows then agent A which is active, to directly interact with agent B through method calls. The sequential updating ensures that no other agent will touch the asset or cash and the direct method calls ensure a synchronous updating of the mutable state of both objects with no time passing between these updates.

B.1 Implementation

We start with the implementation of step 4 with the Random Monad and remove the data-flows from AgentIn and AgentOut. We then add a field in AgentOut which allows the agent to indicate that it wants to start a transaction with another agent with an initial data-package. Also we add a field in AgentIn which indicates an incoming transaction request from another agent with the given data-package. In addition we need another field in AgentOut which allows the agent to indicate that it accepts the incoming request:

```
data AgentIn d = AgentIn
  { aiId        :: !AgentId
  , aiRequestTx :: !(Event (AgentData d))}

data AgentOut m o d = AgentOut
  { aoObservable :: !o
  , aoRequestTx  :: !(Event (AgentData d, AgentTX m o d))
  , aoAcceptTx   :: !(Event (d, AgentTX m o d))}
```

We run the transactions in the specialised agent-transaction signal-functions *AgentTX* with different input and output types. This allows us to restrict the possible actions of an agent within a transaction:

```
type AgentTX m o d = SF m (AgentTXIn d) (AgentTXOut m o d)
```

The input *AgentTXIn* to an agent-transaction holds optional data and flags which indicate that the other agent has either committed or aborted the transaction.

```
data AgentTXIn d = AgentTXIn
  { aiTxData   :: Maybe d
  , aiTxCommit :: Bool
  , aiTxAbort  :: Bool
  }
```

The output *AgentTXOut* of an agent-transaction hold optional data a flag to abort the transaction and optional commit data which is Just in case the agent wants to commit. When committing the agent has to provide a potentially changed AgentOut and optionally a new agent behaviour signal-function. If the agent provides a signal-function when committing, the behaviour of the agent after the transaction will be this signal-function. If no signal-function is provided then the original one will be used.

```
data AgentTXOut m o d = AgentTXOut
  { aoTxData   :: Maybe d
  , aoTxCommit :: Maybe (AgentOut m o d, Maybe (Agent m o d))
  , aoTxAbort  :: Bool
  }
```

We also provide type aliases for our SIR implementation:

```
type SIRMonad g     = Rand g
data SIRMsg         = Contact SIRState deriving (Show, Eq)
type SIRAgentIn     = AgentIn SIRMsg
type SIRAgentOut g = AgentOut (SIRMonad g) SIRState SIRMsg
type SIRAgent g     = Agent (SIRMonad g) SIRState SIRMsg
type SIRAgentTX g  = AgentTX (SIRMonad g) SIRState SIRMsg
```

Stepping the simulation is now slightly more complex as in every step we need to run the transactions. Fortunately it is easy to provide customised implementations of MSFs in dunai, which is a bit more tricky in Yampa and requires to expose internals.

```
stepSimulation :: RandomGen g => [SIRAgent g] -> [SIRAgentIn] -> SF (SIRMonad g) () [SIRAgentOut g]
stepSimulation sfs ains = MSF (\_ -> do
  res <- mapM (\ (ai, sf) -> unMSF sf ai) (zip ains sfs)
  let aos  = fmap fst res
      sfs' = fmap snd res
      ais  = map aiId ains
      aios = zip ais aos

  -- this works only because runTransactions is stateless and runs the SFs with dt = 0
  ((aios', sfs''), _) <- unMSF runTransactions (aios, sfs')

  let aos'  = map snd aios'
      ains' = map agentIn ais
      ct    = stepSimulation sfs'' ains'

  return (aos', ct))
```

The implementation of *runTransactions* is quite involved and omitted here because it would require too much space, but we will give a short informal description. All agents are iterated in an unspecified sequence and if an agent requests a transaction the other agent is looked up and the transaction-pair is run. This is done recursively until there are no transaction requests any-more (note that through the AgentOut of a committed transaction, an agent can request a new transaction within the same time-step). Running a transaction-pair works as follows: The target agents signal-function is run again (resulting in a second, or third,... execution, depending on how many transactions have this agent as target) but now with a $\Delta t = 0$. The target agent can then accept the incoming transaction or simply ignore it. If it is ignored the transaction will never start. The fact that the target agent signal-function is run more than once within a simulation step but with a $\Delta t = 0$ requires agents to make their actions time-dependent *but* they must listen to incoming transactions independent of time. The implementation of the infected agent below will make this more clear. When the transaction is accepted the system switches to running the transaction signal-functions after another with passing the data forward and backward between the two agents. It is most important to note that again the signal functions are run with $\Delta t = 0$ because conceptionally transactions happen *instantaneously* without time advancing. This has important implications, and means that we cannot use any time-accumulating function e.g. integral or after within a transaction - simply because it makes no sense as no time passes. If *both* agents commit the transaction their new AgentOuts will replace the ones for the current simulation-step. If either one agent aborts the transaction the current AgentOuts of the current simulation-step will be used.

We provide a sequence diagram of data-flow in a multi-step negotiation as described in the introduction for a visual explanation of the complex protocol which is going on in a transaction.

Now it is time to look at the new agent implementations which use now the agent-transaction mechanism. The recovered agent is exactly the same but the susceptible and infected agent behaviour are very different now. Lets first look at the susceptible agent:

```
susceptibleAgent :: RandomGen g => [AgentId] -> SIRAgent g
susceptibleAgent ais = proc _ -> do
    makeContact <- occasionallyM (1 / contactRate) () -< ()

    if not (isEvent makeContact)
      then returnA -< agentOut Susceptible
      else (do
        contactId <- drawRandomElemS -< ais
        returnA -< requestTx
                    (contactId, Contact Susceptible)
                    susceptibleTx
                    (agentOut Susceptible))
  where
    susceptibleTx :: RandomGen g => SIRAgentTX g
    susceptibleTx = proc txIn ->
      -- should have always tx data
      if hasTxDataIn txIn
          then (do
            let (Contact s) = txDataIn txIn
            -- only infected agents reply, but make it explicit
            if Infected /= s
              -- don't commit with continuation, no change in behaviour
              then returnA -< commitTx (agentOut Susceptible) agentTXOut
              else (do
                infected <- arrM_ (lift (randomBoolM infectivity)) -< ()
                if infected
                  -- commit with continuation as we switch into infected behaviour
                  then returnA -< commitTxWithCont
                                    (agentOut Infected)
                                    infectedAgent
                                    agentTXOut
                  -- don't commit with continuation, no change in behaviour
                  else returnA -< commitTx (agentOut Susceptible) agentTXOut))
          else returnA -< abortTx agentTXOut
```

Instead of using a switch the susceptible agent behaves completely time-dependent and occasionally starts a new agent-transaction with a random agent. The function *susceptibleTx* handles the reply of the other agent. Note that we only commit with a continuation in case the agent becomes infected.

The infected agent is slightly less complex and still uses the switch mechanism:

```
infectedAgent :: RandomGen g => SIRAgent g
infectedAgent = switch infected (const recoveredAgent)
  where
    infected :: RandomGen g => SF (SIRMonad g) SIRAgentIn (SIRAgentOut g, Event ())
    infected = proc ain -> do
      recEvt <- occasionallyM illnessDuration () -< ()
      let a = event Infected (const Recovered) recEvt
```

```
    -- note that at the moment of recovery the agent can still infect others
    -- because it will still reply with Infected
    let ao = agentOut a

    if isRequestTx ain
      then (returnA -< (acceptTX
                          (Contact Infected)
                          (infectedTx ao)
                          ao, recEvt))
      else returnA -< (ao, recEvt)

  infectedTx :: RandomGen g => SIRAgentOut g -> SIRAgentTX g
  infectedTx ao = proc _ ->
    -- it is important not to commit with continuation as it
    -- would reset the time of the SF to 0. Still occasionally
    -- would work as it does not accumulate time but functions
    -- like after or integral would fail
    returnA -< commitTx ao agentTXOut
```

The agent acts time-dependent which in this case is the transition from infected to recovered - if occasionallyM is run with $\Delta t = 0$ then no Event can happen. The agent checks on every function call of infected for incoming transactions and accepts them all, independent of the state - only susceptible agents request transactions anyway. The agent simply replies with a Contact Infected and immediately commits the transaction in the transaction signal-function but does not switch into a new continuation.

B.2 Discussion

Note that the transactions run in the same monad as the normal agent behaviour signal-function which allows to add an environment as in step 5. In this case care must be taken when one has changed the environment but aborts the transaction as a roll back of the environment won't happen automatically. A different approach would allow to run the TX in a different monad and bring in e.g. the transactional state monad Control.Monad.Tx which supports rolling back of changes to the state.

The concept of agent-transactions is not explicitly known in the agent-based community and a novel development of this paper. The reason for this is that agent-transactions are already implicitly available in traditional OO implementations in which agents can call each others methods and change their state. By implementing this necessary and important concept in a pure functional approach we arrived at agent-transactions which make these synchronous, instantaneous, one-to-one interactions explicit.

Received March 2018