First contact: Agents meet Haskell
Simulating epidemics using Functional Reactive Programming

An Agent-Based Approach

Jonathan Thaler
School of Computer Science
University of Nottingham
jonathan.thaler@nottingham.ac.uk

Thorsten Altenkirch
School of Computer Science
University of Nottingham
thorsten.altenkirch@nottingham.ac.uk

Peer-Olaf Siebers
School of Computer Science
University of Nottingham
peer-olaf.siebers@nottingham.ac.uk

Abstract

TODO: cite my own 1st paper from SSC2017: add it to citations
TODO: refine it: start with simulating epidemics and then go into ABS

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, the Haskell community hasn't been much in contact with the community of ABS due to the latter's primary focus on the object-oriented programming paradigm. This paper tries to bridge the gap between those two communities by introducing the Haskell community to the concepts of ABS. We do this by deriving an agent-based implementation for the simple SIR model from epidemiology. In our approach we leverage the basic concepts of ABS with functional reactive programming from Yampa and Dunai which results in a surprisingly fresh, powerful and convenient EDSL for programming ABS in Haskell.

Index Terms

Functional Reactive Programming, Agent-Based Simulation

I. Introduction

In this paper we derive a pure functional approach for agent-based simulation in Haskell. We start from a very simple solution running in the Random Monad, then making the transition to Yampa

The aim of this paper is to show how ABS can be done in Haskell and what the benefits and drawbacks are. We do this by introducing the SIR model of epidemiology and derive an agent-based implementation for it based on Functional Reactive Programming. By doing this we give the reader a good understanding of what ABS is, what the challenges are when implementing it and how we solved these in our approach. We then discuss details which must be paid attention to in our approach and its benefits and drawbacks. The contribution is a novel approach to implementing ABS with powerful time-semantics and more emphasis on specification and possibilities to reason about the correctness of the simulation.

II. 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 [1]. We informally assume the following about our agents TODO: need some references here, we cannot claim this without citation here (cite Peers book):

- 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 [2] identifies ABS to be especially applicable for analysing *"spatially distributed systems of heterogeneous autonomous actors with bounded information and computing capacity"*. Thus in the line of the simulation types *Statistic* †, *Markov* ‡, *System Dynamics* §, *Discrete Event* $^{\mp}$, ABS is the most powerful one as it allows to model the following:

- Linearity & Non-Linearity $^{\dagger\ddagger\S\mp}$ - the dynamics of the simulation can exhibit both linear and non-linear behaviour.
- Time $^{\dagger\ddagger\S\mp}$ - agents act over time, time is also the source of pro-activity.
- States $^{\ddagger\S\mp}$ - agents encapsulate some state which can be accessed and changed during the simulation.

Fig. 1: Transitions in the SIR compartment model.

- Feedback-Loops §∓ - because agents act continuously and their actions influence each other and themselves, 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, making this a unique feature of ABS, not possible in the other simulation models.
- Spatiality & Networks - agents can be situated within e.g. a spatial (discrete 2d, continuous 3d,...) or network environment, making this also a unique feature of ABS, not possible in the other simulation models.

III. The SIR Model

To explain the concepts of ABS and of our functional reactive approach to it, we introduce the SIR model as a motivating example. It is a very well studied and understood compartment model from epidemiology [3] which allows to simulate the dynamics of an infectious disease like influenza, tuberculosis, chicken pox, rubella and measles [4] 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 with each other *on average* with a given rate β per time-unit and get 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.

The dynamics of this model over time can be formalized using the System Dynamics (SD) approach [5] which models a system through differential equations. For the SIR model we get the following equations:

$$\frac{\mathrm{d}S}{\mathrm{d}t} = -infectionRate \tag{1}$$
$$\frac{\mathrm{d}I}{\mathrm{d}t} = infectionRate - recoveryRate \tag{2}$$
$$\frac{\mathrm{d}R}{\mathrm{d}t} = recoveryRate \tag{3}$$

$$infectionRate = \frac{I\beta S\gamma}{N} \tag{4}$$
$$recoveryRate = \frac{I}{\delta} \tag{5}$$

Solving these equations is then 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*. The $1+$ in $I(t)$ amounts to the initially infected agent - if there wouldn't be a single infected one, the system would immediately reach equilibrium.

$$S(t) = N + \int_0^t -infectionRate \, \mathrm{d}t \tag{6}$$
$$I(t) = 1 + \int_0^t infectionRate - recoveryRate \, \mathrm{d}t \tag{7}$$
$$R(t) = \int_0^t recoveryRate \, \mathrm{d}t \tag{8}$$

There exist a huge number of software packages which allow to conveniently express SD models using a visual approach like in Figure 2.

Running the SD simulation over time results in the dynamics as shown in Figure 3 with the given variables.

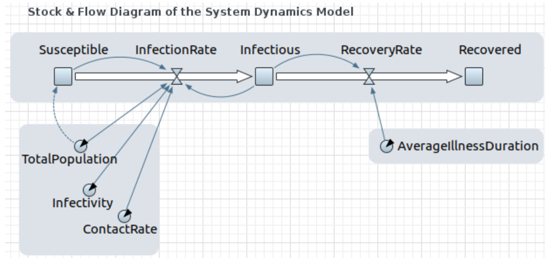

Fig. 2: A visual representation of the SD stocks and flows of the SIR compartment model. Picture taken using AnyLogic Personal Learning Edition 8.1.0.

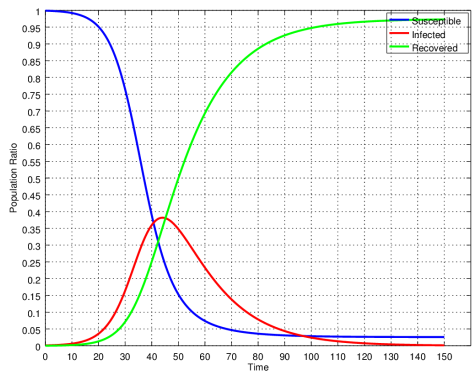

Fig. 3: Dynamics of the SIR compartment model using the System Dynamics approach. Population Size N = 1,000, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time-steps.

An Agent-Based approach

The SD approach is inherently top-down because the emergent property of the system is formalized in differential equations. The question is if such a top-down behaviour can be emulated using ABS, which is inherently bottom-up. Also the question is if there are fundamental drawbacks and benefits when doing so using ABS. Such questions were asked before and modelling the SIR model using an agent-based approach is indeed possible. It is important to note that SD can be seen as operating on averages thus treating the population completely continuous which results in non-discrete values of stocks e.g. 3.1415 infected persons. Thus the fundamental approach to map the SIR model to an ABS is to discretize the population and model each person in the population as an individual agent. The transition 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.

TODO: this is already a too technical explanation which fixes the implementation details already on messaging / data-flow - this is too early and in deriving our approach we will implement 4 different approaches (feedback of all agent-states, data-flow, environment, transactions) TODO: the main point is that we are implementing a state-chart with the transitions are the main thing to consider

- 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. Note that we need to sample from an exponential CDF because the rate is proportional to the size of the population as [6] pointed out.
- 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*. Obviously the already mentioned messaging which allows agents to interact is perfectly suited to do this.
  - *Susceptibles*: These agents make contact with other random agents (excluding themselves) with a "Susceptible" message. They can be seen to be the drivers of the dynamics.
  - *Infected*: These agents only reply to incoming "Susceptible" messages with an "Infected" message to the sender. Note that they themselves do *not* make contact pro-actively but only react to incoming one.
  - *Recovered*: These agents do not need to send messages because contacting it or being contacted by it has no influence on the state.

- Transition of susceptible to infected state - a susceptible agent needs to have made contact with an infected agent which happens when it receives an "Infected" message. If this happens an infection occurs with a probability of γ. The infection can be calculated by drawing p from a uniform random-distribution between 0 and 1 - infection occurs in case of $\gamma >= p$. Note that this needs to be done for *every* received "Infected" message.
- Transition of infected to recovered - a person recovers *on average* after δ time unites. This is implemented by drawing the duration from an exponential distribution [6] with $\lambda = \frac{1}{\delta}$ and making the transition after this duration.

For a more in-depth introduction of how to approximate an SD model by ABS see [7] 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 [8].

IV. Deriving a functional approach

In this section we will derive a functional approach for implementing an agent-based simulation of the SIR model. We will start out with a very naive approach and show its limitations which can be overcome by bringing in FRP. Then in three steps we will add more concepts and generalisations, ending up at the final approach which utilises monadic stream functions (MSF) [9], a generalisation of FRP. Although 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, we haven't talked about technical implementation details on how to actually implement such a state-machine. In these steps we will ultimately present four different approaches on how to implement these states and transitions. Although all result *on average* in the same dynamics, not all of them are equally expressive and testable.

A. Step I: Naive beginnings

In our first step we start with modelling the states of the agents for which we simply use an Algebraic Data Type (ADT):

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

Also agents are ill for some duration meaning we need to keep track when a potentially infected agent recovers. Also as previously mentioned, 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 simply in a list and define helper functions:

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

is :: SIRState -> SIRAgent -> Bool
is s (s',_) = s == s'

susceptible :: SIRAgent
susceptible = (Susceptible, 0)

infected :: Time -> SIRAgent
infected t = (Infected, t)

recovered :: SIRAgent
recovered = (Recovered, 0)
```

Next we need to think about how to actually step our simulation. For this we define a function which simply steps 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. TODO: need a much better explanation and maybe split up into more steps? 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 Rand Monad at this point as threading a generator through the simulation and the agents is very cumbersome. Thus our simulation stepping runs in the Rand Monad:

```
runSimulation :: RandomGen g
              => Time
              -> TimeDelta
              -> Agents
              -> Rand g [Agents]
runSimulation tEnd dt as = runSimulationAux 0 dt as []
  where
    runSimulationAux :: RandomGen g
                     => Time
                     -> TimeDelta
                     -> Agents
                     -> [Agents]
```

```
                    -> Rand g [Agents]
    runSimulationAux t dt as acc
      | t >= tEnd = return $ reverse (as : acc)
      | otherwise = do
        as' <- stepSimulation dt as
        runSimulationAux (t + dt) 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 all the agents states so it can draw random contacts. 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 that it can be neglected.

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 - its state 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 <- doTimes (floor rc) (makeContact as)
    if elem True cs
      then infect
      else return susceptible

  where
    makeContact :: RandomGen g => Agents -> Rand g Bool
    makeContact as = do
      randContact <- randomElem as
      if (is Infected randContact)
        then randomBoolM infectivity
        else return False

    infect :: RandomGen g => Rand g SIRAgent
    infect = do
      randIllDur  <- randomExpM (1 / illnessDuration)
      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
    | otherwise = infected t'
  where
    t' = t - dt
```

1) Results: We run the simulation for $t = 150$ time-units with a fixed $\Delta t = 1.0$. With increasing number of agents the dynamics approach the one of the SD simulation. TODO: add pictures

Reflecting on our first naive approach we can conclude that it already introduced the most 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 $t + 1$.
- Environment - as environment we implicitly assume a fully-connected network where every agent 'knows' every other agents, including itself and thus can make contact with every other agent (including itself).
- Stochasticity - its an inherently stochastic simulation, which is indicated by the Rand Monadic type.

- 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 Rand monad and NOT in the IO Monad. This guarantees that no external, uncontrollable sources of randomness can interfere with the simulation.
- Dynamics - it works as expected: with increasing number of agents our solution approaches the SD dynamics

Nonentheless our approach has also weaknesses and dangers:

1) Δt is passed explicitly as argument to the agent and needs to be dealt with explicitly. It seems to be 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?
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, 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 SIRStates 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 step in which we will address points 1 and 2, point 3 will be solved in step 3.

B. Step II: Adding FRP

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 functional. We can solve both these weaknesses by switching to the functional reactive programming (FRP) paradigm, because it allows to express systems with discrete and continuous time-semantics. TODO: need more introduction. In this step we are focusing on arrowized FRP using the library Yampa. In it, time is handled implicit and cannot be messed with and the whole system is built on the concept of signal-functions (SF). A signal-function is basically a continuation which allows then 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.

We start by defining our agents now as a signal-function which receives the states of all agents as input and outputs the state of the agent:

```
type SIRAgent = SF [SIRState] SIRState
```

Now we can re-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 of step 1 where it acts as a sink which returns constantly the same state:

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

The implementation of a susceptible agent in FRP is a bit more involved but much more expressive and elegant:

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

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

```
                then returnA -< (Infected, Event ())
                else returnA -< (Susceptible, NoEvent))
            else returnA -< (Susceptible, NoEvent))
        else returnA -< (Susceptible, NoEvent)
```

The implementation works as follows: the agent behaves as susceptible until it becomes infected, then it behaves as an infected agent by switching into the *infectedAgent* SF. Instead of randomly drawing the number of contacts to make we now follow a fundamentally different approach by using the occasionally function. It generates on average an event after the given time - the important difference is that 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. When occasionally generates an event we then draw a random element from the other agents state and if we picked an infected agent we need to draw a random boolean which results in True with a uniform probability of 0.05 given in *infectivity*. If the boolean is True this means the agent got infected thus returning the Infected state and an Event with unit which signals that the agent will switch now into the infected behaviour - the state and behaviour is now implicit and much more expressive.

We deal with randomness different now and implement signal-functions built on the noiseR function provided by Yampa. This function takes a range of values and the random-number generator as input and returns the *next* value in the range. This is another example of the 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 function works similar but takes the 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)
```

Implementing the infected agent in FRP is also a bit more involved but much more expressive too:

```
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)
```

The infected agent behaves as infected until it recovers on average after the illness duration after which it behaves then as a recovered agent by switching into *recoveredAgent*. As in 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.

Running and stepping the simulation works now a bit different:

```
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

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

Yampa provides the function *embed* which allows to run a signal-function for a given number of steps where in each step one provides the Δt and an optional input. We run the signal-function *stepSimulation*:

```
stepSimulation :: [SIRAgent] -> [SIRState] -> SF () [SIRState]
stepSimulation sfs as =
    dpSwitch
      (\_ sfs' -> (map (\sf -> (as, sf)) sfs'))
      sfs
      -- if we switch immediately we end up in endless switching, so always wait for 'next'
      (switchingEvt >>> notYet)
      cont
```

```haskell
  where
    switchingEvt :: SF ((), [SIRState]) (Event [SIRState])
    switchingEvt = arr (\ (_, newAs) -> Event newAs)

    cont :: [SIRAgent] -> [SIRState] -> SF () [SIRState]
    cont sfs' newAs = stepSimulation sfs' newAs
```

This function takes all the signal-functions and current states of all agents and returns a signal-function which has the unit-type as input and returns a list of agent-states. What we need to do is to run all agents signal-functions in parallel where all the agent-states are passed as inputs and collect the output of all signal-functions into a list. Fortunately Yampa provides the function *dpSwitch* for this task TODO: explain the d in p, and why we use it. Its first argument is the pairing-function which pairs up the input to the signal-functions, the second argument is the collection of signal-functions, the third argument is a signal-function generating the switching event and the last argument is a function which generates the continuation after the switching event has occurred. pSwitch then 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 continuation-generation function gets passed the signal-functions after they were run in parallel and the data of the switching event which in combination allows us to recursively switch back into the stepSimulation function. In every step we generate a switching event which passes the final agent-states to the continuation-generation which in turn simply returns stepSimulation recursively but now with the new signal-functions and the new agent-states. Note that we delay the switching event always by one step because the continuations are evaluated always upon switching which would result in an infinite switching loop if not delayed using *notYet*.

1) Results: As already mentioned, because we are now using the occasionally function we need a different approach of sampling the system. In the infected agent occasionally determines the time when an agent recovers so its a discrete transition but in a susceptible agent it determines the average number of contacts per time unit: it is a rate with an exponential distribution. This requires us to sample the system with small enough Δt to arrive at the correct solution - if we choose a too large Δt, we loose events which will result in dynamics which do not approach the SD dynamics. We investigated the behaviour of occasionally under varying Δt, which we added as Appendix TODO add appendix. From this it becomes apparent that we need to step our simulation with a $\Delta t <= 0.1$ to arrive at a sufficiently close approximation of the SD dynamics. TODO: show results

TODO: ALARM ALARM !!!! our dynamics do not approach the ones of SD yet, Step 1 solution is much better, probably because we need supersampling! when reducing time-deltas to 0.001 we arrive at good solutions. TODO: compare the results to random-monad solution 100 agents. with smaller and smaller time-delta we need only 1 infected agent and arrive already at a good approximation. this is not the case in random-monad where because of 1.0 time-delta it results in too non fine-grained resolution

By moving on to FRP using Yampa we made a huge improvement in clarity, expressivity and robustness of our implementation. Also by using explicit time-semantics with *occasionally* we can achieve extremely fine grained stochastics: as opposed to draw random number of events we create only a single event or not. This requires then to sample the system with a much smaller Δt: we are treating it as a truly continuous system. 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 step 3 in which we introduce a more general and much more controlled mechanism for feeding back agent-states.

C. Step III: Adding data-flow

In this step we will introduce a data-flow mechanism between agents which makes the feedback explicit. 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 other agent sees its data. As a remedy we introduce data-flows which allow an agent to send arbitrary data to other agents. 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 behaviour we had before: feedback. This change requires then 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.

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:

```haskell
type AgentId     = Int
type AgentData 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:

```haskell
data AgentIn d = AgentIn
  {
    aiId    :: !AgentId
```

```
  , aiData  :: ![AgentData 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        :: ![AgentData 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 / data 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 simply define an ADT which defines a contact-message, and for *o* which defines the 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 (Show, Eq)
type SIRAgentIn  = AgentIn SIRMsg
type SIRAgentOut = AgentOut SIRState SIRMsg
type SIRAgent    = Agent SIRState SIRMsg
```

Obviously the existing implementation from step 2 needs to be adjusted. Lets look at the initial agent-behaviour:

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

It still takes a random-number generator, the initial sir-state and returns the corresponding signal-function depending on the state now in addition it now takes a list of agent ids. When using 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 is simply using a list. Again when we look at the functions which are returned we see that recovered agent is still the same: it is a sink which ignores the environment and the random-number generator.

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

The implementation is nearly the same as in step 2 but instead of returning only the 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 in step 2 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
      let ao = respondToContactWith Infected ain (agentOut a)
      returnA -< (ao, recEvt)
```

The implementation of the infected agent now basically works the same as in step 2 but it additionally needs to reply to incoming contact data-flows with an "Infected" reply. This makes the difference to step 2 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 ain ao = onData respondToContactWithAux ain ao
```

```
  where
    respondToContactWithAux :: AgentData SIRMsg -> SIRAgentOut -> SIRAgentOut
    respondToContactWithAux (senderId, Contact _) ao = dataFlow (senderId, Contact state) ao

onData :: (AgentData d -> acc -> acc) -> AgentIn d -> acc -> acc
onData dHdl ai a = foldr (\msg acc'-> dHdl msg acc') a (aiData ai)

dataFlow :: AgentData 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 TODO DOLLAR 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

          if isEvent makeContact
            then returnA -< (dataFlow (contactId, Contact Susceptible) TODO DOLLAR agentOut Susceptible, NoEvent)
            else returnA -< (agentOut Susceptible, NoEvent))

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

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

Again the implementation is very similar to step 2 with the fundamental difference 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 Arrows, which allows us to refer to a variable after it is defined. In combination with iPre we introduced a local state - the random-number generator - which changes in every step. 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 - 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)
      cont
```

```
  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

    cont :: [SIRAgent] -> [SIRAgentIn] -> SF () [SIRAgentOut]
    cont sfs nextAins = stepSimulation sfs nextAins
```

The distribution of the data-flows happens in the *switchingEvt* signal-function and is then passed on to the continuation-generation function as in step 2. The difference is that it creates now 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 pSwitch is now slightly more straightforward as it just pairs up the AgentIn with its corresponding signal-function.

1) Reflection: It seems that by introducing the data-flow mechanism we have complicated things but this is not so. 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 (TODO: cite Henrik papers) 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 messagebox 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 by 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 a pretty decent implementation of an agent-based SIR approach. The next three steps focus - as this 3rd one - on introducing more concepts and generalising our implementation so far. What we are lacking at the moment is a general treatment of the environment and of synchronised transactional behaviour between agents. To be able to conveniently introduce both 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 [9]. The authors of Dunai implement BearRiver which is a re-implementation of Yampa on top of MSF which should allow us to easily replace Yampa with MSFs in our implementation of Step 3.

D. Step IV: Generalising to Monadic Stream Functions

TODO: write a bit introductory words for this subsection

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* which indicates 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 [1]. We start by re-defining our general agent signal-function, introducing the monad (stack) our SIR implementation runs in and the sir-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 and care 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 (\aos -> map aoObservable aos) aoss
  where
    steps = floor TODO DOLLAR t / dt
    dts = replicate steps ()
```

[1]This was not quite true at the time we wrote this paper, where *occasionally*, *noiseR* and *dpSwitch* were missing. We simply implemented these functions and created a pull request using git.

```haskell
    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.

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 RandomMonad *again* just as we started but now with the full features functional reactive programming! We start by re-defining the SIRMonad and SIRAgent:

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

Note that we parametrise the Random Monad with a RandomGen g thus this requires to add the RandomGen type-class to all functions where it was not yet added. We also simply remove all RandomGen arguments to all functions except *runSimulation*. 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:

```haskell
susceptibleAgent :: RandomGen g => [AgentId] -> SIRAgent g
susceptibleAgent ais =
    switch
      susceptible
      (const TODO DOLLAR infectedAgent)
  where
    susceptible :: RandomGen g
                => SF (SIRMonad g) SIRAgentIn (SIRAgentOut, Event ())
    susceptible = proc ain -> do
      infected <- arrM (\ain -> lift TODO DOLLAR gotInfected infectivity ain) -< ain

      if infected
        then returnA -< (agentOut Infected, Event ())
        else (do
          makeContact <- occasionallyM (1 / contactRate) () -< ()
          contactId   <- drawRandomElemSF                   -< ais

          if isEvent makeContact
            then returnA -< (dataFlow (contactId, Contact Susceptible) TODO DOLLAR agentOut Susceptible, NoEvent)
            else returnA -< (agentOut Susceptible, 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:

```haskell
drawRandomElemS :: MonadRandom m => SF m [a] a
drawRandomElemS = proc as -> do
  r <- getRandomRS ((0, 1) :: (Double, Double)) -< ()
  let len = length as
  let idx = (fromIntegral TODO DOLLAR 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 in *runSimulation* we need to replace *runIdentity* by *evalRand*:

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

3) Reflections: 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 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 we would loose all guarantees about the reproducibility of our 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 IO 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 RandomMonad in step 1 to FRP in step 2. Running in the RandomMonad within FRP is convenient but we could achieve the same with passing RandomGen around as we showed in Step 3. 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.

E. Step V: Adding an environment

we already have one variant of possible environment scenarios: the read-only one. Now we will introduce a different approach of communication between the agents by introducing a read/write environment. adding Environment: when in in/out its cumbersome, end up with n copies, pro-active Environment needs to be hacked in, have additional complexities

F. Step VI: Adding agent transactions

TX: running agents and with 0 dt is easy with bearriver.

Acknowledgments

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

References

[1] M. Wooldridge, *An Introduction to MultiAgent Systems*, 2nd ed. Wiley Publishing, 2009.

[2] J. M. Epstein, *Generative Social Science: Studies in Agent-Based Computational Modeling*. Princeton University Press, Jan. 2012, google-Books-ID: 6jPiuMbKKJ4C.

[3] W. O. Kermack and A. G. McKendrick, "A Contribution to the Mathematical Theory of Epidemics," *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 115, no. 772, pp. 700–721, Aug. 1927. [Online]. Available: http://rspa.royalsocietypublishing.org/content/115/772/700

[4] R. H. Enns, *It's a Nonlinear World*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[5] D. E. Porter, "Industrial Dynamics. Jay Forrester. M.I.T. Press, Cambridge, Mass.; Wiley, New York, 1961. xv + 464 pp. Illus. $18," *Science*, vol. 135, no. 3502, pp. 426–427, Feb. 1962. [Online]. Available: http://science.sciencemag.org/content/135/3502/426.2

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

[7] C. M. Macal, "To Agent-based Simulation from System Dynamics," in *Proceedings of the Winter Simulation Conference*, ser. WSC '10. Baltimore, Maryland: Winter Simulation Conference, 2010, pp. 371–382. [Online]. Available: http://dl.acm.org/citation.cfm?id=2433508.2433551

[8] A. Ahmed, J. Greensmith, and U. Aickelin, "Variance in System Dynamics and Agent Based Modelling Using the SIR Model of Infectious Disease," *arXiv:1307.2001 [cs]*, Jul. 2013, arXiv: 1307.2001. [Online]. Available: http://arxiv.org/abs/1307.2001

[9] I. Perez, M. Brenz, and H. Nilsson, "Functional Reactive Programming, Refactored," in *Proceedings of the 9th International Symposium on Haskell*, ser. Haskell 2016. New York, NY, USA: ACM, 2016, pp. 33–44. [Online]. Available: http://doi.acm.org/10.1145/2976002.2976010