Pure Functional Epidemics

An Agent-Based Approach

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

ABSTRACT

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

So far mainly object-oriented techniques and languages have been used in ABS. Using the SIR model of epidemiology, which simulates the spreading of an infectious disease through a population, we demonstrate how to use pure Functional Reactive Programming to implement ABS. With our approach we can guarantee the reproducibility of the simulation at compile time and rule out specific classes of run-time bugs, something that is not possible with traditional object-oriented languages. Also, we found that the representation in a purely functional format is conceptually quite elegant and opens the way to formally reason about ABS.

KEYWORDS

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

ACM Reference Format:
Jonathan Thaler, Thorsten Altenkirch, and Peer-Olaf Siebers. 2019. Pure Functional Epidemics: An Agent-Based Approach. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The traditional approach to Agent-Based Simulation (ABS) has so far always been object-oriented techniques, due to the influence of the seminal work of Epstein et al [17] in which the authors claim "[..] object-oriented programming to be a particularly natural development environment for Sugarscape specifically and artificial societies generally [..]" (p. 179). This work established the metaphor in the ABS community, that *agents map naturally to objects* [33] which still holds up today.

In this paper we challenge this metaphor and explore ways of approaching ABS in a pure (lack of implicit side-effects) functional way using Haskell. By doing this we expect to leverage the benefits of pure functional programming [23]: higher expressivity through declarative code, being polymorph and explicit about side-effects through monads, more robust and less susceptible to bugs due to explicit data flow and lack of implicit side-effects.

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

Over the course of three steps, we derive all necessary concepts required for a full agent-based implementation. We start with a Functional Reactive Programming (FRP) [49] solution using Yampa [22] to introduce most of the general concepts and then make the transition to Monadic Stream Functions (MSF) [37] which allow us to add more advanced concepts of ABS to pure functional programming.

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

The contributions of this paper are:

- We present an approach to ABS using *declarative* analysis with FRP in which we systematically introduce the concepts of ABS to *pure* functional programming in a step-by-step approach. Also this work presents a new field of application to FRP as to the best of our knowledge the application of FRP to ABS (on a technical level) has not been addressed before. The result of using FRP allows expressing continuous time-semantics in a very clear, compositional and declarative way, abstracting away the low-level details of time-stepping and progress of time within an agent.
- Our approach can guarantee reproducibility already at compile time, which means that repeated runs of the simulation with the same initial conditions will always result in the same dynamics, something highly desirable in simulation in general. This can only be achieved through purity, which guarantees the absence of implicit side-effects, which allows to rule out non-deterministic influences at compile time through the strong static type system, something not possible with traditional object-oriented approaches. Further, through purity and the strong static type system, we can rule out important classes of run-time bugs e.g. related to dynamic typing, and the lack of implicit data-dependencies

IFL'18, August 2019, Lowell, MA, USA
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

which are common in traditional imperative object-oriented approaches.
- Using pure functional programming, we can guarantee the correct semantics of agent execution through the types, something not possible using traditional object-oriented approaches. We demonstrate that we can have sequential monadic behaviour but where agents act *conceptually* at the same time in lock-step, guaranteed through the type system.

In Section 2 we define Agent-Based Simulation, introduce Functional Reactive Programming, Arrowized programming and Monadic Stream Functions, because our approach builds heavily on these concepts. In Section 3 we introduce the SIR model of epidemiology as an example model to explain the concepts of ABS. The heart of the paper is Section 4 in which we derive the concepts of a pure functional approach to ABS in three steps, using the SIR model. Section 5 discusses related work. Finally, we draw conclusions and discuss issues in Section 6 and point to further research in Section 7.

2 BACKGROUND

2.1 Agent-Based Simulation

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

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

We informally assume the following about our agents [28, 42, 50]:

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

Epstein [16] identifies ABS to be especially applicable for analysing *"spatially distributed systems of heterogeneous autonomous actors with bounded information and computing capacity"*. They exhibit the following properties:

- Linearity & Non-Linearity - actions of agents can lead to non-linear behaviour of the system.
- Time - agents act over time which is also the source of their 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.

2.2 Functional Reactive Programming

Functional Reactive Programming is a way to implement systems with continuous and discrete time-semantics in pure functional languages. There are many different approaches and implementations but in our approach we use *Arrowized* FRP [24, 25] as implemented in the library Yampa [11, 22, 31].

The central concept in Arrowized FRP is the Signal Function (SF) which can be understood as a *process over time* which maps an input- to an output-signal. A signal can be understood as a value which varies over time. Thus, signal functions have an awareness of the passing of time by having access to Δt which are positive time-steps with which the system is sampled.

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

Yampa provides a number of combinators for expressing time-semantics, events and state-changes of the system. They allow to change system behaviour in case of events, run signal functions and generate stochastic events and random-number streams. We shortly discuss the relevant combinators and concepts we use throughout the paper. For a more in-depth discussion we refer to [11, 22, 31].

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

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

Randomness. In ABS, often one needs to generate stochastic events which occur based on e.g. an exponential distribution. Yampa provides the combinator *occasionally :: RandomGen g ⇒ g → Time → b → SF a (Event b)* for this. It takes a random-number generator, a rate and a value the stochastic event will carry. It generates events on average with the given rate. Note that at most one event will be generated and no 'backlog' is kept. This means that when this function is not sampled with a sufficiently high frequency, depending on the rate, it will lose 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* $\longrightarrow$ *(a, [(DTime, Maybe a)])* $\longrightarrow$ *[b]* which allows to run an SF for a given number of steps where in each step one provides the Δt and an input *a*. The function then returns the output of the signal function for each step. Note that the input is optional, indicated by *Maybe*. In the first step at $t = 0$, the initial *a* is applied and whenever the input is *Nothing* in subsequent steps, the last *a* which was not *Nothing* is re-used.

2.3 Arrowized programming

Yampa's signal functions are arrows, requiring us to program with arrows. Arrows are a generalisation of monads which, in addition to the already familiar parameterisation over the output type, allow parameterisation over their input type as well [24, 25].

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

There exists a number of arrow combinators which allow arrowized programing in a point-free style but due to lack of space we will not discuss them here. Instead we make use of Paterson's do-notation for arrows [34] which makes code more readable as it allows us to program with points.

To show how arrowized programming works, we implement a simple signal function, which calculates the acceleration of a falling mass on its vertical axis as an example [38].

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

To create an arrow, the *proc* keyword is used, which binds a variable after which the *do* of Patersons do-notation [34] follows. Using the signal function *integral :: SF v v* of Yampa which integrates the input value over time using the rectangle rule, we calculate the current velocity and the position based on the initial position *p0* and velocity *v0*. The <<< is one of the arrow combinators which composes two arrow computations and *arr* simply lifts a pure function into an arrow. To pass an input to an arrow, -< is used and <- to bind the result of an arrow computation to a variable. Finally to return a value from an arrow, *returnA* is used.

2.4 Monadic Stream Functions

Monadic Stream Functions (MSF) are a generalisation of Yampa's signal functions with additional combinators to control and stack side effects. An 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 [36, 37]:

```haskell
newtype MSF m a b =
  MSF { unMSF :: 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. MSFs are implemented in Dunai, which is available on Hackage. Dunai allows us to apply monadic transformations to every sample by means of combinators like *arrM :: Monad m* $\Rightarrow$ *(a* $\longrightarrow$ *m b)* $\longrightarrow$ *MSF m a b* and *arrM_ :: Monad m* $\Rightarrow$ *m b* $\longrightarrow$ *MSF m a b*. A part of the library Dunai is BearRiver, a wrapper which re-implements Yampa on top of Dunai, which enables one to run arbitrary monadic computations in a signal function. BearRiver simply adds a monadic parameter *m* to each SF which indicates the monadic context this signal function runs in.

To show how arrowized programming with MSFs works we extend the falling mass example from above to incorporate monads. In this example we assume that in each step we want to accelerate our velocity *v* not by the gravity constant anymore but by a random number in the range of 0 to 9.81. Further we want to count the number of steps it takes us to hit the floor, that is when position *p* is less than 0. Also when hitting the floor we want to print a debug message to the console with the velocity by which the mass has hit the floor and how many steps it took.

We define a corresponding monad stack with *IO* as the innermost Monad, followed by a *RandT* transformer for drawing random-numbers and finally a *StateT* transformer to count the number of steps we compute. We can access the monadic functions using *arrM* in case we need to pass an argument and *_arrM* in case no argument to the monadic function is needed:

```haskell
type FallingMassStack g = StateT Int (RandT g IO)
type FallingMassMSF g   = SF (FallingMassStack g) () Double

fallingMassMSF :: RandomGen g => Double -> Double -> FallingMassMSF g
fallingMassMSF v0 p0 = proc _ -> do
  -- drawing random number for our gravity range
  r <- arrM_ (lift $ lift $ getRandomR (0, 9.81)) -< ()
  v <- arr (+v0) <<< integral -< (-r)
  p <- arr (+p0) <<< integral -< v
  -- count steps
  arrM_ (lift (modify (+1))) -< ()
  if p > 0
    then returnA -< p
    -- we have hit the floor
    else do
      -- get number of steps
      s <- arrM_ (lift get) -< ()
      -- write to console
      arrM (liftIO . putStrLn) -< "hit floor with v " ++ show v ++
                                  " after " ++ show s ++ " steps"
      returnA -< p
```

To run the *fallingMassMSF* function until it hits the floor we proceed as follows:

```haskell
runMSF :: RandomGen g => g -> Int -> FallingMassMSF g -> IO ()
runMSF g s msf = do
  let msfReaderT = unMSF msf ()
      msfStateT  = runReaderT msfReaderT 0.1
      msfRand    = runStateT msfStateT s
      msfIO      = runRandT msfRand g
  (((p, msf'), s'), g') <- msfIO
  when (p > 0) (runMSF g' s' msf')
```

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 is the reason why we need one extra lift for accessing *StateT* and *RandT*. Thus *unMSF* returns a computation in the *ReaderT Double* Monad which we need to peel away using *runReaderT*. This then results in a *StateT Int* computation which we evaluate by using *runStateT* and the current number of steps as state. This then results in another monadic computation of *RandT* Monad which we evaluate using *runRandT*. This finally returns an *IO* computation which we simply evaluate to arrive at the final result.

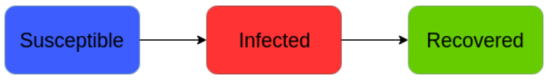


Figure 1: States and transitions in the SIR compartment model.

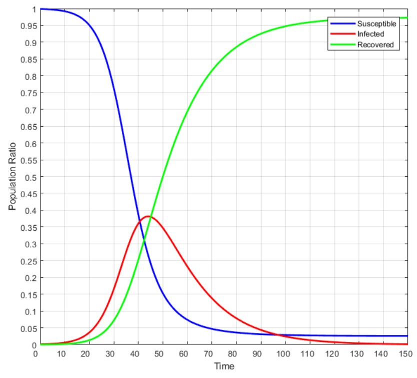


Figure 2: Dynamics of the SIR compartment model using the System Dynamics approach. Population Size $N = 1{,}000$, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time-steps.

3 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 [27] which allows to simulate the dynamics of an infectious disease like influenza, tuberculosis, chicken pox, rubella and measles spreading through a population [15].

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 of β per time-unit and become infected with a given probability γ when interacting with an infected person. When infected, a person recovers *on average* after δ time-units and is then immune to further infections. An interaction between infected persons does not lead to re-infection, thus these interactions are ignored in this model. This definition gives rise to three compartments with the transitions seen in Figure 1.

This model was also formalized using System Dynamics (SD) [39]. In SD one models a system through differential equations, allowing to conveniently express continuous systems which change over time, solving them by numerically integrating over time which gives then rise to the dynamics. We won't go into detail here and provide the dynamics of such a solution for reference purposes, shown in Figure 2.

An Agent-Based approach

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 happening due to discrete events caused both by interactions amongst the agents and time-outs. The major advantage of ABS is that it allows to incorporate spatiality as shown in Section 4.3 and simulate heterogenity of population e.g. different sex, age. This is not possible with other simulation methods e.g. SD or Discrete Event Simulation [51].

According to the model, every agent makes *on average* contact with β random other agents per time unit. In ABS we can only contact discrete agents thus we model this by generating a random event on average every $\frac{1}{\beta}$ time units. We need to sample from an exponential distribution because the rate is proportional to the size of the population [5]. 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.

4 DERIVING A PURE FUNCTIONAL APPROACH

In [45] two fundamental problems of implementing an ABS from a programming-language agnostic point of view is discussed. The first problem is how agents can be pro-active and the second how interactions and communication between agents can happen. For agents to be pro-active, they must be able to perceive the passing of time, which means there must be a concept of an agent-process which executes over time. Interactions between agents can be reduced to the problem of how an agent can expose information about its internal state which can be perceived by other agents.

Both problems are strongly related to the semantics of a model and the authors show that it is of fundamental importance to match the update-strategy with the semantics of the model - the order in which agents are updated and actions of agents are visible can make a big difference and need to match the model semantics. The authors identify four different update-strategies, of which the *parallel* update-strategy matches the semantics of the agent-based SIR model due to the underlying roots in the System Dynamics approach. In the parallel update-strategy, the agents act *conceptually* all at the same time in lock-step. This implies that they observe the same environment state during in a time-step and actions of an agent are only visible in the next time-step - they are isolated from each other, see Figure 3.

Also, the authors [45] have shown the influence of different deterministic and non-deterministic elements in ABS on the dynamics and how the influence of non-determinism can completely break them down or result in different dynamics despite same initial conditions. This means that we want to rule out any potential source of non-determinism which we achieve by keeping our implementation pure. This rules out the use of the IO Monad and thus any potential source of non-determinism under all circumstances because we would lose all compile time guarantees about reproducibility. Still we will make use of the Random Monad which indeed allows

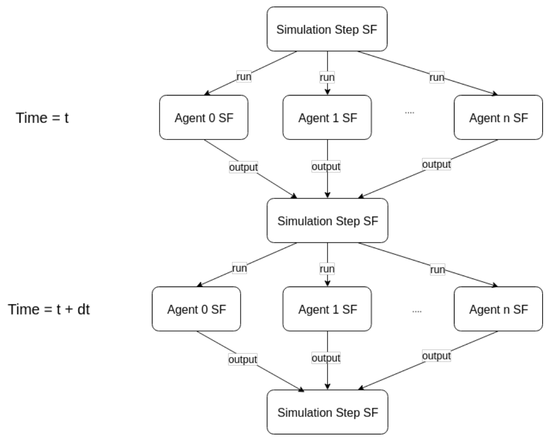


Figure 3: Parallel, lock-step execution of the agents.

side-effects but the crucial point here is that we restrict side-effects only to these types in a controlled way without allowing general unrestricted effects like in traditional object-oriented approaches in the field.

In the following, we derive a pure functional approach for an ABS of the SIR model in which we pose solutions to the previously mentioned problems. We start out with a first approach in Yampa and show its limitations. Then we generalise it to a more powerful approach, which utilises Monadic Stream Functions, a generalisation of FRP. Finally we add an explicit environment, making the example more interesting and shows the real strength of ABS over other simulation methodologies like System Dynamics and Discrete Event Simulation [1].

4.1 Functional Reactive Programming

As described in the Section 2.2, Arrowized FRP [24] is a way to implement systems with continuous and discrete time-semantics where the central concept is the signal function, which can be understood as a process over time, mapping an input- to an output-signal. Technically speaking, a signal function is a continuation which allows to capture state using closures and hides away the Δt, which means that it is never exposed explicitly to the programmer, meaning it cannot be messed with.

As already pointed out, agents need to perceive time, which means that the concept of processes over time is an ideal match for our agents and our system as a whole, thus we will implement them and the whole system as signal functions.

4.1.1 Implementation. We start by defining the SIR states as ADT and our agents as signal function (SF) which receives the SIR states of all agents as input and outputs the current SIR state of the agent:

```
data SIRState = Susceptible | Infected | Recovered

type SIRAgent = SF [SIRState] SIRState
```

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

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

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

Depending on the initial state we return the corresponding behaviour. Note that we are 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 because a recovered agent does nothing, stays immune forever and can not get infected again in this model. Thus a recovered agent is a consuming state from which there is no escape, it simply acts as a sink which returns constantly *Recovered*:

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

Lets look how we can implement the behaviour of a susceptible agent. It 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. To make contact, it 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.

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. When an infection event occurs we change the behaviour of an agent using the Yampa combinator *switch*, which is quite elegant and expressive as it makes the change of behaviour at the occurrence of an event explicit. Note that to make contact *on average*, we use Yampas *occasionally* function which requires us to carefully select the right Δt for sampling the system as will be shown in results.

Note the use of *iPre* $:: a \rightarrow SF\ a\ a$ which delays the input signal by one sample, taking an initial value for the output at time zero. The reason for it is, that we need to delay the transition from susceptible to infected by one step due to the semantics of the *switch* combinator: whenever the switching event occurs, the signal function into which is switched will be run at the time of the event. This means that a susceptible agent could make a transition to recovered within one time-step, which we want to prevent because the semantics, should be that only one state-transition can happen per time-step.

```
susceptibleAgent :: RandomGen g => g -> SIRAgent
susceptibleAgent g
    = switch
      -- delay switching by 1 step to prevent against transition
      -- from Susceptible to Recovered within one time-step
      (susceptible g >>> iPre (Susceptible, NoEvent))
      (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
          -- draw random element from the list
          a <- drawRandomElemSF g -< as
          case a of
            Infected -> do
              -- returns True with given probability
              i <- randomBoolSF g infectivity -< ()
              if i
                then returnA -< (Infected, Event ())
                else returnA -< (Susceptible, NoEvent)
             _       -> returnA -< (Susceptible, NoEvent))
        else returnA -< (Susceptible, NoEvent)
```

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

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

An infected agent recovers *on average* after δ time units. This is implemented by drawing the duration from an exponential distribution [5] with $\lambda = \frac{1}{\delta}$ and making the transition to the *Recovered* state after this duration. Thus 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.

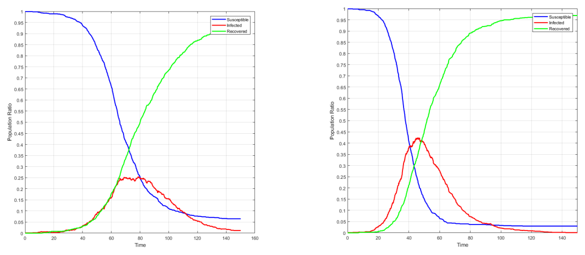
```
infectedAgent :: RandomGen g => g -> SIRAgent
infectedAgent g
    = switch
      -- delay switching by 1 step
      (infected >>> iPre (Infected, NoEvent))
      (const recoveredAgent)
  where
    infected :: SF [SIRState] (SIRState, Event ())
    infected = proc _ -> do
      recEvt <- occasionally g illnessDuration () -< ()
      let a = event Infected (const Recovered) recEvt
      returnA -< (a, recEvt)
```

For running the simulation we use Yampas function *embed*:

```
runSimulation :: RandomGen g => g -> Time -> DTime
              -> [SIRState] -> [[SIRState]]
runSimulation g t dt as
    = embed (stepSimulation sfs as) ((), dts)
  where
    steps     = floor (t / dt)
    dts       = replicate steps (dt, Nothing)
    n         = length as
    (rngs, _) = rngSplits g n [] -- unique rngs for each agent
    sfs       = zipWith sirAgent rngs as
```

What we need to implement next is a closed feedback-loop - the heart of every agent-based simulation. Fortunately, [11, 31] 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 this step's new agent states. Note the use of *notYet* which is required because in Yampa switching occurs immediately at $t = 0$. If we don't delay the switching at $t = 0$ until the next step, we would enter an infinite switching loop - *notYet* simply delays the first switching until the next time-step.

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



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

Figure 4: FRP simulation of agent-based SIR showing the influence of different Δt. Population size of 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 respective Δt.

Yampa provides the *dpSwitch* combinator for running signal functions in parallel, which has the following type-signature:

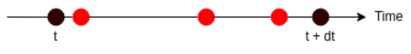
```
dpSwitch :: Functor col
         -- routing function
         => (forall sf. a -> col sf -> col (b, sf))
         -- SF collection
         -> col (SF b c)
         -- SF generating switching event
         -> SF (a, col c) (Event d)
         -- continuation to invoke upon event
         -> (col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

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

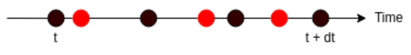
Conceptually, *dpSwitch* allows us to recursively switch back into the *stepSimulation* with the continuations and new states of all the agents after they were run in parallel.

4.1.2 Results. The dynamics generated by this step can be seen in Figure 4.

By following the FRP approach we assume a continuous flow of time, which means that we need to select a *correct* Δt otherwise



(a) Under-sampling



(b) Super-sampling

Figure 5: A visual explanation of under-sampling and super-sampling. The black dots represent the time-steps of the simulation. The red dots represent virtual events which occur at specific points in continuous time. In the case of under-sampling, 3 events occur in between the two time steps but *occasionally* only captures the first one. By increasing the sampling frequency either through a smaller Δt or super-sampling all 3 events can be captured.

we would end up with wrong dynamics. The selection of a correct Δt depends in our case on *occasionally* in the *susceptible* behaviour, which randomly generates an event on average with *contact 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 frequency of events generated by *contact rate*. If we choose a too large Δt, we loose events, which will result in wrong dynamics as can be seen in Figure 4a. This issue is known as under-sampling and is described in Figure 5.

For tackling this issue we have two options. The first one is to use a smaller Δt as can be seen 4b, which results in the whole system being sampled more often, thus reducing performance. The other option is to implement super-sampling and apply it to *occasionally*, 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 to super-sampling would be to introduce a new combinator to Yampa which allows us to super-sample other signal functions.

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

It evaluates the *SF* argument 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 the *SF* argument is calculated. A list of b is returned with length of n containing the result of the n evaluations of the *SF* argument. If 0 or less super samples are requested exactly one is calculated. We could then wrap the occasionally function which would then generate a list of events. We have investigated super-sampling more in-depth but have to omit this due to lack of space.

4.1.3 Discussion. We can conclude that our first step already introduced most of the fundamental concepts of ABS:

- Time - the simulation occurs over virtual time which is modelled explicitly, divided into *fixed* Δt, where at each step all agents are executed.
- Agents - we implement each agent as an individual, with the behaviour depending on its state.
- Feedback - the output state of the agent in the current time-step t is the input state for the next time-step $t + \Delta t$.
- Environment - as environment we implicitly assume a fully-connected network (complete graph) where every agent 'knows' every other agent, including itself and thus can make contact with all of them.
- Stochasticity - it is an inherently stochastic simulation, which is indicated by the random-number generator and the usage of *occasionally*, *randomBoolSF* and *drawRandomElemSF*.
- 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 *not* in the IO Monad. This guarantees that no external, uncontrollable sources of non-determinism can interfere with the simulation.
- Parallel, lock-step semantics - TODO

Using FRP in the instance of Yampa results in a clear, expressive and robust implementation. State is implicitly encoded, depending on which signal function is active. By using explicit time-semantics with *occasionally* we can achieve extremely fine grained stochastics by sampling the system with small Δt: we are treating it as a truly continuous time-driven agent-based system.

A very severe problem, hard to find with testing but detectable with in-depth validation analysis, is the fact that in the *susceptible* agent the same random-number generator is used in *occasionally*, *drawRandomElemSF* and *randomBoolSF*. This means that all three stochastic functions, which should be independent from each other, are inherently correlated. This is something one wants to prevent under all circumstances in a simulation, as it can invalidate the dynamics on a very subtle level, and indeed we have tested the influence of the correlation in this example and it has an impact. We left this severe bug in for explanatory reasons, as it shows an example where functional programming actually encourages very subtle bugs if one is not careful. A possible but not very elegant solution would be to simply split the initial random-number generator in *sirAgent* three times (using one of the splited generators for the next split) and pass three random-number generators to *susceptible*. A much more elegant solution would be to use the Random Monad which is not possible because Yampa is not monadic.

So far we have an acceptable implementation of an agent-based SIR approach. What we are lacking at the moment is a general treatment of an environment and an elegant solution to the correlation of the random-number generators. In the next step we make the transition to Monadic Stream Functions as introduced in Dunai [37], which allows FRP within a monadic context and gives us a way for an elegant solution to the random-number correlation.

4.2 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, solving our problem of correlated random-numbers through the use of the Random Monad.

4.2.1 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 monadic context. If we replace this type-parameter with the Identity Monad, we should be able to keep the code exactly the same, because BearRiver re-implements all necessary functions we are using from Yampa. We simply re-define our agent signal function, introducing the monad stack our SIR implementation runs in:

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

4.2.2 Random Monad. Using the Identity Monad does not gain us anything but it is 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 run the whole simulation within the Random Monad with the full features of FRP, finally solving the problem of correlated random-numbers in an elegant way. We start by re-defining the SIRMonad and SIRAgent:

```
type SIRMonad g = Rand g
type SIRAgent g = SF (SIRMonad g) [SIRState] SIRState
```

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*.

```
occasionallyM :: MonadRandom m => Time -> b -> SF m a (Event b)
-- can be used through the use of arrM and lift
randomBoolM :: RandomGen g => Double -> Rand g Bool
-- this can be used directly as a SF with the arrow notation
drawRandomElemSF :: MonadRandom m => SF m [a] a
```

4.2.3 Discussion. Running in the Random Monad solved the problem of correlated random-numbers and elegantly guarantees us that we won't have correlated stochastics as discussed in the previous section. In the next step we introduce the concept of an explicit discrete 2D environment.

4.3 Adding an environment

In ABS agents are often situated within a discrete 2D environment [17] which is simply a finite *NxM* grid with either a Moore or von Neumann neighbourhood (Figure 6). 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 interactions directly from the shared read-only environment, which will be passed to the agents as input. This allows agents to read the states of all their neighbours which tells them if a neighbour is infected or not. For purposes of a more interesting approach, we restrict the neighbourhood to Moore (Figure 6b).

We also implemented this spatial approach in Java using the well known ABS library RePast [32], to have a comparison with a state of the art approach and came to the same results as shown in Figure 7. This supports that our pure functional approach can produce such results as well and compares positively to the state of the art in the ABS field.

4.3.1 Implementation. We start by defining our discrete 2D environment for which we use an indexed two dimensional array.

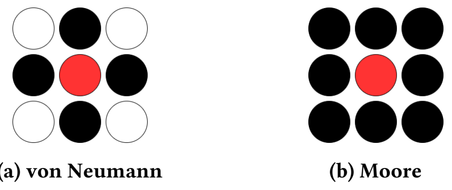


Figure 6: Common neighbourhoods in discrete 2D environments of Agent-Based Simulation.

Each cell stores the agents state of the current time-step, 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 the previous step with *SIREnv* as type for the state. Our agent signal function now has unit input and output type, which indicates that the actions of the agents are only visible through side-effects in the monad stack they are running in.

```
type SIRAgent g = SF (Rand g) SIREnv SIRState
```

The implementation of a susceptible agent is now a bit different. The agent directly queries the environment for its neighbours and randomly selects one of them. The remaining behaviour is similar: The behaviour of an infected agent is similar to in the previous step, with the difference that upon recovery the infected agent updates its state in the environment from Infected to Recovered.

For running the simulation with MSFs we use the function *embed* which is not provided by BearRiver but by Dunai which has important implications. As already explained in the background Section 2.4, Dunai does not know about time in MSFs, which is what BearRiver builds on top of MSFs. Thus, when running our simulation using *embed* we get the *ReaderT* in addition to the other Monad Transformers, which we need to run using *runReaderT*. Note that 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.

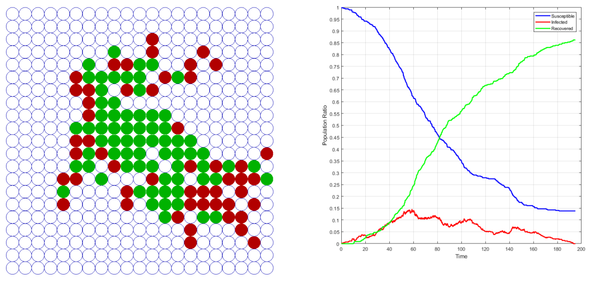
Due to the different approach of returning the SIREnv in every step, we implemented our own MSF: – run all agents sequentially but keep the environment – read-only: it is shared as input with all agents – and thus cannot be changed by the agents themselves

```
simulationStep :: RandomGen g => [(SIRAgent g, Disc2dCoord)]
               -> SIREnv -> SF (Rand g) () SIREnv
simulationStep sfsCoords env = MSF (\_ -> do
  let (sfs, coords) = unzip sfsCoords
  -- run agents sequentially but with shared, read-only environment
  ret <- mapM (`unMSF` env) sfs
  -- construct new environment from all agent outputs for next step
  let (as, sfs') = unzip ret
      env' = foldr (\ (a, coord) envAcc -> updateCell coord a envAcc)
               env (zip as coords)

      sfsCoords' = zip sfs' coords
      cont       = simulationStep sfsCoords' env'
  return (env', cont))

updateCell :: Disc2dCoord -> SIRState -> SIREnv -> SIREnv
```

4.3.2 Results. We implemented rendering of the environments using the gloss library which allows us to cycle arbitrarily through



(a) Environment at $t = 50$ **(b) Dynamics over time**

Figure 7: Simulating the agent-based SIR model on a 21x21 2D grid with Moore neighbourhood (Figure 6b), a single infected agent at the center and same SIR parameters as in Figure 2. Simulation run until $t = 200$ with fixed $\Delta t = 0.01$. Last infected agent recovers around $t = 194$. The susceptible agents are rendered as blue hollow circles for better contrast.

the steps and inspect the spreading of the disease over time visually as seen in Figure 7.

Note that the dynamics of the spatial SIR simulation which are seen in Figure 7b look quite different from the reference 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.

4.3.3 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 advantage is that we can introduce arbitrary restrictions on the neighbourhood as shown with the Moore neighbourhood.

Of course an environment is not restricted to be 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 other simulation approaches and allows us to simulate much more realistic scenarios.

4.4 Additional Steps

ABS involves a few more advanced concepts which we don't fully explore in this paper due to lack of space. Instead we give a short overview and discuss them without presenting code or going into technical details.

4.4.1 Synchronous Agent Interactions. Synchronous agent interactions are necessary when an arbitrary number of interactions between two agents need to happen instantaneously within the same time-step. 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 [17]. In object-oriented programming, the concept of synchronous communication between agents is implemented directly with method calls.

We have implemented synchronous interactions in an additional step. 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 these interactions.

4.4.2 Event Scheduling. 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 [30], which is based on the theory of Discrete Event Simulation [51]. In such an approach the system is not sampled in fixed Δt but advanced as events occur where the system stays constant in between. Depending on the model, in an event-driven approach it may be more natural to express the requirements of the model.

In an additional step we have implemented a rudimentary event-driven approach, which allows the scheduling of events but had to omit it due to lack of space. Using the flexibility of MSFs we added a State transformer to the monad stack, which allows queuing of events into a priority queue. The simulation is advanced by processing the next event at the top of the queue, which means running the MSF of the agent which receives the event. The simulation terminates if there are either no more events in the queue or after a given number of events, or if the simulation time has advanced to some limit. Having made the transition to MSFs, implementing this feature was quite straight forward, which shows the power and strength of the generalised approach to FRP using MSFs.

4.4.3 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 [17] require dynamic agent creation and destruction. 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.

4.4.4 Conflicts in Environment. The semantics of the agent-based SIR model allowed a straight-forward implementation of the parallel update-strategy. This is not easily possible when there could be conflicts in the environment e.g. moving agents where only a single one can occupy a cell. Most models in ABS [17] solve this by implementing a sequential update-strategy [45], where agents are run after another but can already observe the changes by agents run before them in the same time-step. To prevent the introduction of artefacts due to a specific ordering, these models shuffle the agents before running them in each step to average the probability for a specific agent to be run at a fixed position.

TODO Functional programming could offer a different approach due to the way side-effects are handled and immutability of data. One approach could be to randomly select a winner and re-run other conflicting agents Signal Functions as long as the underlying monadic context is robust to re-runs - in case of the Random Monad this would be no problem.

5 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 related to the field of Multi Agent Systems and look into how agents can be specified using the belief-desire-intention paradigm [13, 26, 44].

The author of [4] investigated in his master thesis Haskells parallel and concurrency features to implement (amongst others) *HLogo*, a Haskell clone of the ABS simulation package NetLogo, where agents run within the IO Monad and make use of Software Transactional Memory for a limited form of agent-interactions.

A library for DES and SD in Haskell called *Aivika 3* is described in the technical report [43]. It is not pure, as it uses the IO Monad under the hood and comes only with very basic features for event-driven ABS, which allows to specify simple state-based agents with timed transitions.

Using functional programming for DES was discussed in [26] where the authors explicitly mention the paradigm of FRP to be very suitable to DES.

A domain-specific language for developing functional reactive agent-based simulations was presented in [48]. 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.

Object-oriented programming and simulation have a long history together as the former one emereged out of Simula 67 [12] which was created for simulation purposes. Simula 67 already supported Discrete Event Simulation and was highly influential for today's object-oriented languages. Although the language was important and influential, in our research we look into different approaches, orthogonal to the existing object-oriented concepts.

Lustre is a formally defined, declarative and synchronous dataflow programming language for programming reactive systems [19]. While it has solved some issues related to implementing ABS in Haskell it still lacks a few important features necessary for ABS. We don't see any way of implementing an environment in Lustre as we do in our approach in Section 4.3. Also the language seems not to come with stochastic functions, which are but the very building blocks of ABS. Finally, Lustre does only support static networks, which is clearly a drawback in ABS in general where agents can be created and terminated dynamically during simulation.

There exists some research [14, 41, 47] of using the functional programming language Erlang [3] to implement ABS. The language is inspired by the actor model [1] and was created in 1986 by Joe Armstrong for Eriksson for developing distributed high reliability software in telecommunications. The actor model can be seen as quite influential to the development of the concept of agents in ABS which borrowed it from Multi Agent Systems [50]. It emphasises message-passing concurrency with share-nothing semantics, which maps nicely to functional programming concepts. The mentioned papers investigate how the actor model can be used to close the conceptual gap between agent-specifications, which focus on message-passing and their implementation. Further they also showed that using this kind of concurrency allows to overcome some problems of low level concurrent programming as well. Despite the natural mapping of ABS concepts to such an actor language it leads to simulations which despite same initial starting conditions might lead to different results due to concurrency.

6 CONCLUSIONS

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

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

Also we can statically guarantee the reproducibility of the simulation, which means that repeated runs with the same initial conditions are guaranteed to result in the same dynamics. Although we allow side-effects within agents, we restrict them to only the Random Monad in a controlled, deterministic way and never use the IO Monad which guarantees the absence of non-deterministic side effects within the agents and other parts of the simulation.

Determinism is also ensured by fixing the Δt and not making it dependent on the performance of e.g. a rendering-loop or other system-dependent sources of non-determinism as described by [38]. Also by using FRP we gain all the benefits from it and can use research on testing, debugging and exploring FRP systems [35, 38].

Also we showed how to implement the *parallel* update-strategy [45] in a way that the correct semantics are enforced and guaranteed already at compile time through the types. This is not possible in traditional imperative implementations and poses another unique benefit over the use of functional programming in ABS.

Issues

Currently, the performance of the system is not comparable to imperative implementations. We compared the performance of our pure functional approach as presented in Section 4.3 to an implementation in Java using the ABS library RePast [32]. We ran the simulation until $t = 100$ on a 51x51 (2,601 agents) with $\Delta t = 0.1$ (unknown in RePast) and averaged 8 runs. The performance results make the lack of speed of our approach quite clear: the pure functional approach needs around 72.5 seconds whereas the Java RePast version just 10.8 seconds on our machine to arrive at $t = 100$. We have already started investigating speeding up performance through the use of Software Transactional Memory [20, 21] which is quite straight forward when using MSFs. It shows very good results but we have to leave the investigation and optimization of the performance aspect of our approach for further research as it is beyond the scope of this paper.

Despite the strengths and benefits we get by leveraging on FRP, there are errors that are not raised at compile time, e.g. we can still have infinite loops and run-time errors. This was for example investigated in [40] where the authors use dependent types to avoid some run-time errors in FRP. We suggest that one could go further and develop a domain specific type system for FRP that makes the FRP based ABS more predictable and that would support further mathematical analysis of its properties. Furthermore, moving to dependent types would pose a unique benefit over the traditional object-oriented approach and should allow us to express and guarantee even more properties at compile time. We leave this for further research.

In our pure functional approach, agent identity is not as clear as in traditional object-oriented programming, where there is a quite clear concept of object-identity through the encapsulation of data and methods. Signal Functions don't offer this strong identity and one needs to build additional identity mechanisms on top e.g. sending messages to specific agents.

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

7 FURTHER RESEARCH

We see this paper as an intermediary and necessary step towards dependent types for which we first needed to understand the potential and limitations of a non-dependently typed pure functional approach in Haskell. Dependent types are extremely promising in functional programming as they allow us to express stronger guarantees about the correctness of programs and go as far as allowing to formulate programs and types as constructive proofs which must be total by definition [2, 29, 46].

So far no research using dependent types in agent-based simulation exists at all. 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 [6] 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.

We hypothesize that dependent types could help ruling out even more classes of bugs at compile time and encode invariants and model specifications on the type level, which implies that we don't need to test them using e.g. property-testing with QuickCheck. This would allow the ABS community to reason about a model directly in code. We think that a promising approach is to follow the work of [7–10, 18] in which the authors utilize GADTs to implement an indexed monad which allows to implementation correct-by-construction software.

- Accessing the environment in section 4.3 involves indexed array access which is always potentially dangerous as the indices have to be checked at run-time.
  Using dependent types it should be possible to encode the environment dimensions into the types. In combination with suitable data types for coordinates one should be able to ensure already at compile time that access happens only within the bounds of the environment.
- In the SIR implementation one could make wrong state-transitions e.g. when an infected agent should recover, nothing prevents one from making the transition back to susceptible.
  Using dependent types it might be possible to encode invariants and state-machines on the type level which can prevent such invalid transitions already at compile time. This would be a huge benefit for ABS because many agent-based models define their agents in terms of state-machines.
- An infected agent recovers after a given time - the transition of infected to recovered is a timed transition. Nothing prevents us from *never* doing the transition at all.
  With dependent types we might be able to encode the passing of time in the types and guarantee on a type level that an infected agent has to recover after a finite number of time steps.
- In more sophisticated models agents interact in more complex ways with each other e.g. through message exchange using agent IDs to identify target agents. The existence of an agent is not guaranteed and depends on the simulation time because agents can be created or terminated at any point during simulation.
  Dependent types could be used to implement agent IDs as a proof that an agent with the given id exists *at the current time-step*. This also implies that such a proof cannot be used in the future, which is prevented by the type system as it is not safe to assume that the agent will still exist in the next step.
- In our implementation, we terminate the SIR model always after a fixed number of time-steps. We can informally reason that restricting the simulation to a fixed number of time-steps is not necessary because the SIR model *has to* reach a steady state after a finite number of steps. This means that at that point the dynamics won't change any more, thus one can safely terminate the simulation. Informally speaking, the reason for that is that eventually the system will run out of infected agents, which are the drivers of the dynamic. We know that all infected agents will recover after a finite number of time-steps *and* that there is only a finite source for infected agents which is monotonously decreasing.
  Using dependent types it might be possible to encode this in the types, resulting in a total simulation, creating a correspondence between the equilibrium of a simulation and the totality of its implementation. Of course this is only possible for models in which we know about their equilibria a priori or in which we can reason somehow that an equilibrium exists.

ACKNOWLEDGMENTS

The authors would like to thank I. Perez, H. Nilsson, J. Greensmith, M. Baerenz, H. Vollbrecht, S. Venkatesan and J. Hey for constructive feedback, comments and valuable discussions. TODO: add thanks to the anonymous referees from IFL 2018?

REFERENCES

[1] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

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

[3] Joe Armstrong. 2010. Erlang. *Commun. ACM* 53, 9 (Sept. 2010), 68–75. https://doi.org/10.1145/1810891.1810910

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

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

[6] 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

[7] Edwin Brady. 2013. Programming and Reasoning with Algebraic Effects and Dependent Types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13).* ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2500365.2500581

[8] Edwin Brady. 2016. *State Machines All The Way Down - An Architecture for Dependently Typed Applications.* Technical Report. https://www.idris-lang.org/drafts/sms.pdf

[9] Edwin Brady and Kevin Hammond. 2010. Correct-by-Construction Concurrency: Using Dependent Types to Verify Implementations of Effectful Resource Usage Protocols. *Fundam. Inf.* 102, 2 (April 2010), 145–176. http://dl.acm.org/citation.cfm?id=1883634.1883636

[10] Edwin C. Brady. 2011. Idris âĂŤ systems programming meets full dependent types. In *In Proc. 5th ACM workshop on Programming languages meets program verification, PLPV âĂŽ11.* ACM, 43–54.

[11] 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

[12] Ole-johan Dahl. 2002. The birth of object orientation: the simula languages. In *Software Pioneers: Contributions to Software Engineering, Programming, Software Engineering and Operating Systems Series.* Springer, 79–90.

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

[14] Antonella Di Stefano and Corrado Santoro. 2005. Using the Erlang Language for Multi-Agent Systems Implementation. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT '05).* IEEE Computer Society, Washington, DC, USA, 679–685. https://doi.org/10.1109/IAT.2005.141

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

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

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

[18] Simon Fowler and Edwin Brady. 2014. Dependent Types for Safe and Secure Web Programming. In *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages (IFL '13).* ACM, New York, NY, USA, 49:49–49:60. https://doi.org/10.1145/2620678.2620683

[19] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (Sept. 1991), 1305–1320. https://doi.org/10.1109/5.97300

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

[21] Tim Harris and Simon Peyton Jones. 2006. Transactional memory with data invariants. https://www.microsoft.com/en-us/research/publication/transactional-memory-data-invariants/

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

[23] 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

[24] 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

[25] 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

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

[27] 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

[28] 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

[29] James McKinna. 2006. Why Dependent Types Matter. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06).* ACM, New York, NY, USA, 1–1. https://doi.org/10.1145/1111037.1111038

[30] 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

[31] 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

[32] Michael J. North, Nicholson T. Collier, Jonathan Ozik, Eric R. Tatara, Charles M. Macal, Mark Bragen, and Pam Sydelko. 2013. Complex adaptive systems modeling with Repast Simphony. *Complex Adaptive Systems Modeling* 1, 1 (March 2013), 3. https://doi.org/10.1186/2194-3206-1-3

[33] 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.

[34] 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

[35] 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

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

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

[38] 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

[39] 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

[40] 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

[41] Gene I. Sher. 2013. *Agent-Based Modeling Using Erlang Eliminating The Conceptual Gap Between The Programming Language & ABM.*

[42] 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.

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

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

[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] Carlos Varela, Carlos Abalde, Laura Castro, and Jose GulÃŋas. 2004. On Modelling Agent Systems with Erlang. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Erlang (ERLANG '04).* ACM, New York, NY, USA, 65–70. https://doi.org/10.1145/1022471.1022481

[48] Ivan Vendrov, Christopher Dutchyn, and Nathaniel D. Osgood. 2014. Frabjous: A Declarative Domain-Specific Language for Agent-Based Modeling. In *Social*

Computing, Behavioral-Cultural Modeling and Prediction, William G. Kennedy, Nitin Agarwal, and Shanchieh Jay Yang (Eds.). Number 8393 in Lecture Notes in Computer Science. Springer International Publishing, 385–392. https://doi.org/10.1007/978-3-319-05579-4_47

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

[51] 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.

Received May 2018