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

———————— MY CONCLUSIONS ———————— Go into a hardcore crunch for 3 weeks and make the following changes:

1. Instead of implementing the model in 3 different ways (Random Monad, Classic Yampa, MSFs) I will only leave in the Classic Yampa version: this frees up space for other stuff (below), focuses on the central concept (it seemed that the step to MSFs with the environment was not clear enough) and I don't need to explain MSFs. Also Henrik mentioned that the Random Monad approach is a detour, so it feels right to throw it out. 2. Put the MSF implementation with the environment into a small functional pearl which shows step-by-step how to implement an agent-based SIR model within a Discrete 2D environment in a pure functional way using Dunai. I guess in such a functional pearl I can discuss more implementation details, develop it slower and am not under the pressure of a unique, strong scientific contribution. The code, lots of text and figures are already there, so I think it is really feasible. 3. Add a section on verification in which first I will use QuickCheck to test properties and verify the correctness (also for selecting the right delta t) through property testing (which I have already implemented, so I 'just' need to bring it in shape and write it up). Second I want to try to 'somehow' (in)formally reason about the correctness in the code - which I have never done so far and I don't know how to do it. I mean, there are some examples in Grahams Book and papers, but I don't know how to formally do this with code which is based on Yampa, maybe informally would work in combination with QuickCheck. This last bit is the most unpredictable one, I might get it done or I might fail completely. 4. Incorporate the referees feedback as far as applicable in the new version. From Review 1: => obviously the scientific contribution of the "purity" aspect, conceptually cleaner and reproducibility is not clear enough => the referee seems not to get the point of reproducibility: yes it is true that "as long as you dont use non-deterministic concurrency...", but in our approach we can actually guarantee that statically at compile time by being pure - this is DEFINITELY NOT possible in traditional imperative OOP. Any non-determinism could break reproducibility e.g. reading from file, user-input,... which we also can exclude at compile time with our approach. Obviously this seems to be not clear enough yet, we need to make this much much clearer and also make the point clear that an ABS as we implement it is a pure computation in the end. => being the first one seems to be not a contriubtion? hmm this is probably true, i guess i had to learn this lesson => functional approach conceptually cleaner: how can we back this up? => "All functional implementations suffer from having to choose the right time step deltat you definitely do not have the illusion of continuous time.": this is just wrong by the referee: actually quite the opposite is true: our implementation allows the illusion of continuous time in the way it is implemented and time-semantic functions are used (occasionally). Selecting the right dT is ALWAYS an issue in EVERY simulation, be it OOP or functional, but the way we represent our continous time-system is definitely not as elegant possible in OOP. Still I cannot really substantiate this claim as i havent looked into FRP in OOP. => may indicate that we still want to do too much in the paper, get rid of Random Monad and Environment => true, maybe we should omit these claims of superiority over OOP alltogether - except the reproducibility guarantees at compile time through purity. => have a look at the rabbits foxes simulation of the Objects First book by Barnes Koelling => "but the concept of identity of an agent is a bit lost." this is definitely true in our functional approach, especially in our SIR implementation because the SIR agent has no explicit internal state appart from the current continuation which determines the Susceptible/Infected/Recovered state: but I claim that this alone is enough for the agents identity. When we look at e.g. the sugarscape model, agent-identity becomes much more explicit e.g. when we add agent-ids and messaging and much more complex internal state. => Yampa was built on arrowized programming to prevent depending on the past, thus this shouldnt be an issue. True OO implementations are much less likely to depend on the past but again the way how data

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

is handled is completely different and if one uses OO FRP then probably one might run into similar problems of depending on the past => dependent types is work in progress, will be next paper - Some entries in the references lack details for finding them without a search engine, e.g. 26, 27, 28. Is reference 2 still in preparation, since 2005? - In an English text citations should not be nouns (the authors of [x]), they are parenthesised additions. Removing them should leave a readable text. Also citations such as [6], [9], [12] should be combined into [6, 9, 12].

From Review 2: => True, we should make this performance problem clear in the very beginning => Maybe add a short section on Cloud Haskell which 'theoretically' would allow us to solve this problem and also maps nicely to ABS but leads in a completely different direction than what we want to do in our research: explicit message passing which is what we do in the end in abs, not pure anymore thus not guaranteed repeatability at compile time, time stepping needs to be synchronised across all agents which becomes a bottle neck, could we implement it as a distributed event-driven approach? generally: it would allow massive parallel simulations but results in a completely different approach, vastly different to ours. although Performance is important, our primary concern is about ensuring/verifying/proofing general correctness of the implementation of a model => ok I really might have missed some approaches, add simula 67 and lustre => the referee is wrong here and may have misunderstood something, the reproducibility at compile time follows directly from purity! this is not possible in traditional OO languages! There is no need for evidence, it is clear. BUT again, we need to present this much more explicit as it seems to confuse people (or am I confused here) => True, maybe we should omit the claim that it is conceptually cleaner than oop or can we substantiate it some how? => true, maybe we should add a formal reasoning section which proves that our implementation is a match of the SD formulas => yes the agents are imperative which in our opinion lies in the nature of ABS in general. And if we are going down to the technical level: we are still being functional although we are "emulating" imperativeness in a monad / arrows! => i understand why the referee is so upset but sorry, the Yampa implementation is still much more robust and clearer than the Random Monad one even if there are things like notyet. Get rid of the Random Monad step => the deltat thing is a problem in all simulations, not unique to our approach => Most worrisome, there seems to be a serious error in Yampa's susceptibleAgent implementation: it looks like (occasionally g), (drawRandomElemSF g) and (randomBoolSF g) receive all the same g. That is, the three streams are highly correlated! The random boolean produced by randomBoolSF g will be correlated to the random agent picked by (drawRandomElemSF g). That is a very serious and a very subtle error. One can also see this as the illustration that functional approach *facilitates* errors. : that is absolutely true, thanks for this sharp and deep insight. I have to test how much influence this has on the dynamics. I followed the approach of Yampa papers (e.g. space invaders) where the same RNG is passed to different random functions - in a game this shouldnt be too much a problem but in a simulation this could be a severe issue. I suggest implementing running replications and exporting the maximum number of infected and last infected recovered and look at the distributions (as suggested by the statistical lady of the FH) => i will incorporate this as an example that functional approach facilitates errors

From Review 3: => ok this is probably the most valuable comment: the structure of the paper is not good, what we try to get accross is not clear and WHY we are doing it is also unclear. These are the main things also mentioned by the guys above. At least we are not lacking scientific contribution! => ok, i will put the related work section to the end of the paper as i have originally intended to do so, but was urged by my supervisors to move it to the front. this is my paper, i will do it my way. => true, maybe we should directly incorporate the background stuff it into the implementation section instead of presenting it disconnected at this point => I'd strongly advise to restructure the paper so that already the introduction has a concrete example of what agent-based modelling is and perhaps even explain the SIR model in somewhat more detail. I'd then move Section 2 to the end of the paper, and explain the background on FRP at a point where it is actually needed (in / before Section 6.2). It makes little to no sense to talk about a function like dpSwitch without having a concrete example close by. There is a use on page 8, so you might as well explain it at that point. Similarly for arrow notation, which is introduced on page 3 but then not used again until page 7. => true! => maybe consider removing Random Monad approach => his may all be true, but it is difficult to see. Except for the abstraction of time itself, you do not make it very concrete in which way your code is "cleaner", "clearer", or "compositional" more than any other. From my admittedly subjective viewpoint, for a Haskell program, the code presented does not strike me as particularly elegant, and it does not look particularly modular or reusable, given that most aspects seem entirely specific to the implementation of the SIR model. => hmm, what should we do with this? => ok this guy hasnt probably understood the SIR model at all, also not the environment concept. I might conder removing this bit - You often say "The authors of [XY] discuss/present ...". This is bad for two reasons. First of all, numeric references should not be used as a part of the sentence, because the "[XY]" form typographically is an annotation, and also because there's no info conveyed by a number and flipping back and forth is not very nice. Furthermore, it is semantically not what I think you mean. The authors of a given paper may have done a lot of things in their lives, but I think you actually mean that the paper you refer to discusses/presents these things. If you simply name the authors and say "A and B discuss/present ... [XY]", then you avoid both problems.

- Does Section 4 really need to be its own section? => true, probably not

- The formulas on page 4 look really ugly. More importantly, you don't explain $\beta, \gamma and \delta except in the description of Figure 2. These are important parameters, they should be explained in the main text! ok agree about the formulas but i actually explain the parameters...its$ [illegible]

- In Section 6.1, despite the "Naive beginnings" header, I see no reason to model SIRAgents in the way you do. First, the Time associated with the agent is stated to be the "potential recovery time" and in the end turns out to be a duration. One paragraph earlier, you have introduced two type synonyms, one for absolute points in time (Time), and one for durations (TimeDelta), yet you incorrectly use Time and not TimeDelta for the recovery duration. => wrong, the referee misunderstood my code: recoverytime is actually a duration not an absolute point in time, maybe i shouldnt use recovery time but illness duration

More importantly though, the recovery time is only relevant for the "Infected" state. So why model it like this? The correct way is > data SIRState > = Susceptible > | Infected TimeDelta > | Recovered As a result, the code in this variant would already become much cleaner. => thank you! i dont know why this hasnt crossed my mind!

For some reason, while runAgent is parameterised by a TimeDelta, you choose not to pass that parameter to the susceptibleAgent function, and do not take it into account when computing how many contacts the agent has in line 572. This is almost certainly wrong, and I think you're saved only by the fact that in Figure 3 you indicate you've only been running this program with dt values of 1.0. => true, in Random Monad we implicitly step with a dt = 1.0 and it is not possible to change this

- The code

> forM [1..floor rc] (const (makeContact as))

is more idiomatically written as

> replicateM (floor rc) (makeContact as)

- The code

> elem True cs

is more idiomatically written as

> or cs

- In Section 6.1.1 you talk about undersampling the contact-rate. You should highlight here what exactly the problem is, and why it cannot be fixed without switching to a different implementation. => ok add more details on here

- In Section 6.2, I feel quite uncomfortable with the statements about "occasionally" and the dependence on a particular sampling rate. What happens if the sampling rate is chosen incorrectly? Is there no other way to do this that is less sensitive? You say the primary advantage of using FRP is that one does not have to worry about time, but is that true if one has to be careful about sampling rates not just for precision but even for correctness? => maybe i explained it wrong: in FRP one DOES have to worry about time and sampling but it is handled implicitly behind the scenes, there is no explicit delta-t draged thorugh all functions which it is conceptually much cleaner

- The discussion of Dependent Types in Section 8 comes somewhat surprisingly. Sure, you have problems at the end of Section 7 that are not covered by the type system, but they're not a prominent part of the paper. From the presentation in the paper, I cannot immediately see where you're missing more precise types. The efficiency problems would seem like a greater concern. => true but still... i have to make clearer that correctness is our primary concern here and that this approach although a nice try in the right direction is not as far as we can go, which is why we need to bring in dependent types.

[ ] Performance: look into cloud Haskell [ ] cloud Haskell: explicit message passing which is what we do in the end in abs, not pure anymore thus not guaranteed repeatability at compile time, time stepping needs to be synchronised across all agents which becomes a bottle neck, could we implement it as a distributed event-driven approach? generally: it would allow massive parallel simulations but results in a completely different approach, vastly different to ours. although Performance is important, our primary concern is about ensuring/verifying/proofing general correctness of the implementation of a model [ ] Performance: pure functional haskell allows running parallel replicatiins for free without worrying of intereference [ ] look into lustre [ ] look into simula 68 [ ] consider scraping 6.1 naive beginnings to free up space for more theoretical work [ ] make purity aspect much clearer and its implications for compile time: it is really not possible to guarantee that in traditional oo languages [ ] conceptually cleaner: can we substantiate it? no dt, more declarative? compare to lustre and simula 68 [ ] formally reason about: probably not really possible? can i formally proof the correctness of the implementation? [ ] idea: scrape section on Environment as well and try a formal proof of correctness of the yampa only approach: can we formally boil it down to the SD formulas? [ ] consider informal property testing with quickcheck, encodes the sd formula im it, also allows to find the dt [ ] rework paper structure: instead of splitting too much up into separate sections explain it as introduced and used. related work to the very end (as i have wanted it)! implement agent-based sir (no Environment) with yampa, then verify it with quickcheck and then formally reasoning

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

So far mainly object-oriented techniques and languages have been used in ABS. Using the SIR model of epidemiology, which allows to simulate the spreading of an infectious disease through a population, we show how to use Functional Reactive Programming to implement ABS. With our

approach we can guarantee the reproducibility of the simulation already at compile time, which is not possible with traditional object-oriented languages. Also, we claim that this representation is conceptually cleaner 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. 2018. Pure functional epidemics: An Agent-Based Approach. In *Proceedings of Haskell Symposium (HS18).* ACM, New York, NY, USA, **??** 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 [**?**] 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* [**?**] which still holds up today.

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

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

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

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

The contributions of this paper are:

- To the best of our knowledge, we are the first to *systematically* introduce the concepts of ABS to the *pure* functional programming paradigm in a step-by-step approach. It is also the first paper to show how to apply Arrowized FRP to ABS on a technical level, presenting a new field of application to FRP.
- Our approach shows how robustness can be achieved through purity which guarantees reproducibility at compile time, something not possible with traditional object-oriented approaches.
- The result of using Arrowized FRP is a conceptually much cleaner approach to ABS than traditional imperative object-oriented approaches. It allows expressing continuous time-semantics in a much clearer, compositional and declarative way, without having to deal with low-level details related to the progress of time.

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

2 Background

2.1 Functional Reactive Programming

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

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 [**?**], [**?**], [**?**].

Event An event in FRP is an occurrence at a specific point in time which has no duration e.g. the the recovery of an

infected agent. Yampa represents events through the *Event* type which is programmatically equivalent to the *Maybe* type.

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

Sometimes one needs to run a collection of signal functions in parallel and collect all of their outputs in a list. Yampa provides the combinator *dpSwitch* for it. It is quite involved and has the following type-signature:

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

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

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

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

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

2.2 Arrowized programming

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

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 [**?**] 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 [**?**].

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

To create an arrow, the *proc* keyword is used, which binds a variable after which then the *do* of Patersons do-notation [**?**] 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.

2.3 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 [?], [?]:

```
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 => (a -> m b) -> MSF m a b* and *arrM_ :: Monad m => m b -> MSF m a b*.

3 Defining Agent-Based Simulation

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

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

We informally assume the following about our agents [?], [?], [?]:

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

4 The SIR Model

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

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

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

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

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

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

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

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

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

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

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

An Agent-Based approach

The SD approach is inherently top-down because the behaviour of the system is formalized in differential equations. This requires that the macro behaviour of the system is known a priori which may not always be the case. In the case of the SIR model we already have a top-down description of the system in the form of the differential equations from SD. We want now to derive an agent-based approach which exhibits the same dynamics as shown in Figure **??**.

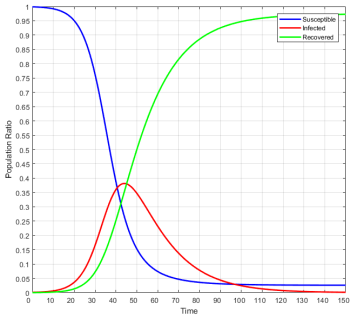


Figure 2. Dynamics of the SIR compartment model using the System Dynamics approach. Population Size N = 1,000, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time-steps.

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

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

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

This results in the following agent behaviour:

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

5 Deriving a pure functional approach

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

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

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

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

In this step we are focusing on Arrowized FRP [**?**] using the library Yampa [**?**]. In it, time is handled implicit, meaning it cannot be messed with, which is achieved by building the whole system on the concept of signal functions

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

(SF). An SF can be understood as a process over time and is technically a continuation which allows to capture state using closures. Both these fundamental features allow us to tackle the weaknesses of our first step and push our approach further towards a truly elegant functional approach.

5.0.1 Implementation

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

We start by modelling the states of the agents:

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

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

```
type Time      = Double
type TimeDelta = Double

type SIRAgent = SF [SIRState] SIRState
```

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

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

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

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

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

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

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

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

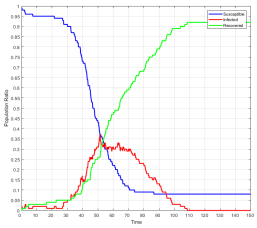
The infected agent behaves as infected until it recovers on average after the illness duration after which it behaves as a recovered agent by switching into *recoveredAgent*. As in the case of the susceptible agent, we use the *occasionally* function to generate the event when the agent recovers. Note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

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

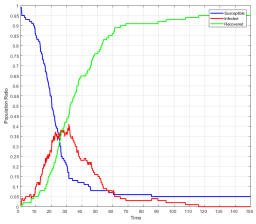
Running and stepping the simulation works now a bit differently, using Yampas function *embed*:

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

What we need to implement next is a closed feedback-loop. Fortunately, [**?**], [**?**] discusses implementing this in Yampa. The function *stepSimulation* is an implementation of such a closed feedback-loop. It takes the current signal



(a) $\Delta t = 0.1$



(b) $\Delta t = 0.01$

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

functions and states of all agents, runs them all in parallel and returns the new agent states of this step. Yampa provides the *dpSwitch* combinator for running signal functions in parallel, which is quite involved and discussed more in-depth in section **??**. It allows us to recursively switch back into the *stepSimulation* with the continuations and new states of all the agents after they were run in parallel. Note the use of *notYet* which is required because in Yampa switching occurs immediately at $t = 0$.

```
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)
```

5.0.2 Results

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

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

5.0.3 Discussion

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

- Time - the simulation occurs over virtual time which is modelled explicitly divided into *fixed* Δt where at each step all agents are executed.
- Agents - we implement each agent as an individual, with the behaviour depending on its state.
- Feedback - the output state of the agent in the current time-step t is the input state for the next time-step $t + \Delta t$.
- Environment - as environment we implicitly assume a fully-connected network (complete graph) where every agent 'knows' every other agent, including itself and thus can make contact with all of them.
- Stochasticity - it is an inherently stochastic simulation, which is indicated by the Random Monad type and the usage of *randomBoolM* and *randomExpM*.
- Deterministic - repeated runs with the same initial random-number generator result in same dynamics. This may not come as a surprise but in Haskell we can guarantee that property statically already at compile time because our simulation runs in the Random Monad and *not* in the IO Monad. This guarantees that no external, uncontrollable sources of randomness can interfere with the simulation.
- Dynamics - with increasing number of agents the dynamics smooth out [**?**].

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

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

6 Related Work

The amount of research on using pure functional programming with Haskell in the field of ABS has been moderate so far. Most of the papers are more related to the field of Multi Agent Systems and look into how agents can be specified using the belief-desire-intention paradigm [**?**], [**?**], [**?**].

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

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

The authors of [**?**] present a domain-specific language for developing functional reactive agent-based simulations. This language called FRABJOUS is human readable and easily understandable by domain-experts. It is not directly implemented in FRP/Haskell but is compiled to Yampa code which they claim is also readable. This supports that FRP is a suitable approach to implement ABS in Haskell. Unfortunately, the authors do not discuss their mapping of ABS to FRP on a technical level, which would be of most interest to functional programmers.

7 Conclusions

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

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

Also we can statically guarantee the reproducibility of the simulation. Within the agents there are no side effects possible which could result in differences between same runs. Every agent has access to its own random-number generator or the Random Monad, allowing randomness to occur in the simulation but the random-generator seed is fixed in the beginning and can never be changed within an agent. This means that after initialising the agents, which *could* run in the IO Monad, the simulation itself runs completely deterministic.

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

Issues

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

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

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

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

8 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 [**?**], [**?**], [**?**].

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

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

Acknowledgments

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

Received March 2018