2nd Year Report

Pure Functional Agent-Based Simulation

Jonathan Thaler (4276122)
jonathan.thaler@nottingham.ac.uk

supervised by
Dr. Peer-Olaf Siebers
Dr. Thorsten Altenkirch

April 23, 2018

Abstract

TODO: update and adopt to 2nd year

This research investigates how the pure functional programming paradigm can be applied to Agent-Based Simulations (ABS). So far there existed no research of how it can be done in this paradigm and it was unclear how to use the mechanisms of the pure functional programming paradigm to ones benefit.

In this 1st year we first conducted fundamental research on the general nature of update-strategies in ABS which is of direct relevance on how to implement ABS in a functional way. Building on these insights, we implemented a highly promising approach to pure functional ABS by building on the Functional Reactive paradigm using the library Yampa. By this we could show that ABS is indeed very possible in Haskell with important benefits. The result is a surprisingly fresh approach to ABS as it allows to incorporate discrete time-semantics similar to Discrete Event Simulation and continuous time-flows like System Dynamics. Also, due to the declarative nature of functional programming the implementations look very much like a model-specification.

In the remaining two years of the PhD the research will be centred on comparing this new pure functional approach to the existing state-of-the-art, which follows the object-oriented paradigm. We will identify benefits and drawbacks and look for unique features which may be only possible using the pure functional programming paradigm.

Contents

Chapter 1

Introduction

TODO

1.1 Motivation

The observations of the problems presented in the previous section leads us to posing fundamental directions and questions which are the basis of the motivation of our thesis.

1. **Alternative approach to object-oriented ABMS** - Is there an alternative view to the established object-oriented view to ABMS which does treat agents *not* like objects and does not mix the concept of agent and object? Is there an alternative to the established object-oriented implementation approach to ABMS which offers composability, explicit data-flow?

2. **Verification & Validation of ABMS** - Is there a way for formal specification and verification which is still readable and does not fall back to pure mathematics? What exactly is the meaning of validation in ABMS and is there a way to do formal validation in ABMS?

Chapter 2

Literature Review

In this chapter we present additional literature fundamental for the aims & objectives of the 2nd year.

TODO: literature on dependent types TODO: literature on verification & validation in Simulation and ABS

Chapter 3

Aims and Objectives

TODO: refine and adapt to 2nd year

This chapter gives a compact and concise overview of the aims and objectives. Chapter 5 gives a more in-depth plan and details in how we will approach the aim in general and the objectives in particular.

3.1 Aim

The aim of this Ph.D. is to investigate how the pure functional programming paradigm can be used to increase the robustness and gain insights into dynamics of Agent-Based Simulations.

3.2 Objectives

1. Develop a library for general-purpose Agent-Based Simulation in Haskell to have a tool in the pure functional programming paradigm to be used for conducting the research.

2. Compare the general approach of pure functional programming in the instance of Haskell and object-oriented programming in the instance of Java to implement ABS. Look into differences and similarities and identify benefits and drawbacks.

3. Explore how pure functional programming can increase testability, verification and correctness of Agent-Based Simulations.

4. Investigate to which extent one can use reasoning-techniques of the pure functional paradigm to reason about dynamics in ABS.

Chapter 4

Work To Date

Here we give a concise overview over the activities performed in the 2nd year.

4.1 Social Simulation Conference 2017

4.2 Paper Published

TODO: Art of Iteration was published in SSC2017 proceedings

4.3 Papers Submitted

4.3.1 Pure Functional Epidemics

This paper, which is attached in Appendix A

4.4 Reports

4.4.1 Haskell Communities and Activities Report (HCAR) May 2017

We wrote a new entry for the HCAR May 2017, which tries to compile and publish novel and on-going ideas in the Haskell community. It is freely available under `https://www.haskell.org/communities/05-2017/html/report.html`. We hope that our idea and the work of our PhD gets a bit more attention and may start some discussions with people interested in this work.

4.4.2 2nd Year Report

This document.

4.5 Talks

So far only two talks were given. The first one was a presentation of the ideas underlying the update-strategies paper at the IMA - seminar day. The second was presenting my ideas about functional reactive ABS to the FP-Lab Group at the FPLunch.

Chapter 5

Future Work Plan

TODO: update and adopt to 2nd year Papers route: end of march 2018: PFE paper, dependent types in ABS: march 2019 for ICFP 2019, towards pure functional abs paper as concept paper to the ABS Community until april 2019 (before starting thesis writing)

In this chapter we discuss in more detail how we plan to approach the aim and objectives stated in Chapter 3. Further we give a short overview of planned papers and present a Gantt-Chart **??** reflecting the most important activities and relevant milestones.

5.1 Approaching the Objectives

5.1.1 Tools and foundations

In the first year a prototype of the library should be implemented with a few examples to early gain a good understanding of the topic and to be able to judge whether it is a dead-end or not. This has already successfully happened: the library prototype exists with a number of non-trivial examples including a full implementation of the Sugarscape model. The library is termed FrABS [1] and is described in Appendix **??**.

To be able to compare our approach to the existing state-of-the-art in the field of ABS we will use the existing library Repast Java. This will allow us to compare the dominant object-oriented paradigm in the field to our approach, discuss similarities and differences and identify draw benefits and drawbacks.

Clarification on the fundamental challenges in implementing ABS from a programming-paradigm agnostic perspective must be done. This was already conducted in the first paper on Update-Strategies in ABS, which is now accepted at SSC2017 and can be found in Appendix **??**

[1] Note that - despite it took quite some effort - this library in itself is no the contribution of this Ph.D. but just a means to an end which allows to research and investigate the objectives and research questions and substantiate the hypotheses.

5.1.2 Pure functional and object-oriented paradigms

The next objective is then to investigate the fundamental differences between the pure functional and object-oriented paradigms in implementing ABS with their respective incarnations Haskell and Java. This research will be conducted in the form of a report because it imposes no limits on content-size, is easier to be incorporated in the thesis than a paper and if required a paper can be condensed out of it. This work should pick up the work of the Update-Strategies paper and broaden and deepen its approach: show in more depth and in more detail which problems need to be solved when implementing an ABS and how both paradigms can be used in their way to solve them and identify potential benefits and drawbacks. There exists already code in Haskell and Java without any additional libraries (except JDK and Haskells standard library called Prelude) produced for the Update-Strategies paper which can be used for this work. In the next step we need to go up the abstraction ladder and investigate how FrABS and Repast Java approach these problems and compare them from a programing paradigm perspective and identify potential benefits and drawbacks. As examples the SIR compartment model in epidemiology should be implemented both as a System-Dynamics and ABS 2D-spatial/network model. Also a full implementation of Sugarscape should be attempted in Repast Java [2]. The questions we want to answer in this objective are

- What are the approaches of pure functional and object-oriented programming to ABS?
- What are the benefits/drawbacks of Haskell and what are the benefits/-drawbacks of Java in implementing ABS?
- Are the benefits/drawbacks orthogonal to each other e.g. are the weaknesses of one language the other languages strength?
- Are there things which are unique when doing Haskell in ABS and cannot be done in a Java approach and vice versa?

At least half of the second year will be allocated to this very important research as it lays the very foundations. We conjecture that the conclusion of this research will be that both approaches - paradigms and libraries - though solving specific problems different, are basically equally powerful in implementing the given examples and ABS in general with no fundamental impossibilities on either side. The main noteable (but not fundamental) differences will be performance and memory-consuption, lines-of-code, expressiveness, usability, learning curve and they way to think about ABS. The question is then: why go through all the pain? Is there a difference? Can't we build somehow on the promised benefits of pure functional programming as in Haskell [3]?

[2] There exists already an implementation of Chapter II in the Repast Java Demo library upon which we can build.

[3] Explicit about side-effects, strong static type-system, algebraic reasoning, declarative style, lambda calculus

5.1.3 Robustness and correctness

The answer to the previous question is attempted in a second study, which will look into how pure functional programming can increase robustness, testability, verification and correctness of Agent-Based Simulations. Again the approach will be in writing a report so not to restrict the size of the content, make it easier to incorporate it in the final thesis and condense a paper out of it if required. Again Repast Java and FrABS Haskell will be used as tools to investigate the research question by investigating the examples implemented in the previous report. The following investigations should be made:

- Compare the potential for bugs at runtime e.g. which are not possible to detect at compile time.
- Reproducibility robustness: can unpredictable side-effects affect the systems behaviour? Can such side-effects be guaranteed to not show up?
- Testing and testability of an implementation: how well can we test it using automated tests e.g. unit-tests, property-tests?
- Verification of an implementation: can we verify the correctness of a model or at least of parts of the specification?
- Can we establish a measure of correctness?

The second half of the 2nd year is allocated to this work.

We conjecture that this research will come to the conclusion that pure functional programming dramatically increases the robustness and correctness of an ABS and that it allows for easier and clearer approach to verification using specification testing in QuickCheck and automated testing using Unit- and Property-Testing.

5.1.4 Reasoning about dynamics

This objective investigates how far we can get using the pure functional paradigms' ability to reason about a program when applying it to reason about the dynamics of an ABS. As use-cases the idea is to look into reasoning about the equivalence of SD- and ABS-implementation of the SIR compartment model in epidemiology and about the equilibrium dynamics of the bilateral decentralized bartering in Sugarscape.

Both the System-Dynamics and Agent-Based implementation of the SIR compartment model in epidemiology lead to the same dynamics or put different: the Agent-Based implementation shows the same dynamics of the SD implementation when using replications. This is shown by plotting the dynamics as graphs.

In the Sugarscape model where agents engage in bilateral decentralized bartering equilibrium is only reached when neo-classical agents are used which

don't die of natural age. The equilibrium is not reached when more realistic assumptions are made. This is shown by plotting the prices over time.

In this part of the Ph.D. we want to answer the following questions:

- What and to which extent can we reason about an Agent-Based Simulation in pure functional programming in general and using FrABS in particular?
- Can we show that the SD and ABS implementations are equivalent through reasoning about the code?
- Can we show that the equilibrium in the decentralized bilateral bartering of Sugarscape is reached / not reached when using neo-classical agents / realistic agents through reasoning about the code?

We are very well aware that this topic alone could justify a Ph.D. on its own and it is unsure if we would could make it this far (if the previous research takes longer than anticipated). Still we want to explicitly have clearly stated these objectives and outlined how to approach them - the worst which can happen is that they merely serve as a very in-depth outlook on future research. Thus the outcome of this research is highly uncertain but the conjecture is that we come to the following conclusion: it highly depends on the complexity and semantics of the model. In the case of the SIR compartment model we expect a success. In the case of the bilateral decentralized bartering in Sugarscape we probably will gain some insights but that at some point due to the interlocking of so many mechanisms it would get too complicated and a full and formal treatment is out of the scope of this Ph.D. We allocate the first half of the 3rd year to this work.

5.2 Planned Papers

For the remainder of the PhD we planned for two more papers. The first one will be focused on the pure functional approach to ABS, and targeted as a conference paper. Because we are doing interdisciplinary research which is a cross-over between the fields of ABMS and functional programming in the theoretical computer sciences, we plan to write two papers on this same topic but for different audiences: one for the ABMS community and one for the field of functional programming. This would allow us to present our new concepts to two different groups and explain the same thing from two different perspectives. Also it would introduce the two groups to the ideas of the other group and may result in a cross-fertilization. The second paper we target as a journal paper and will focus on testing, verification and correctness in the functional approach to ABS. This paper will be written for the ABMS community and released in an ABMS journal.

Chapter 6

Conclusions

6.1 Being Realistic

It is of most importance to stress that we don't condemn the current state-of-the-art approach of object-oriented specification and implementation to ABS. The strength of object-oriented programming is surely that it can be seen as *programming as modelling* and thus will be always an attractive approach to ABS. Also we are realists and know that there are more points to consider when selecting a set of methods for developing software for an ABS than robustness, verification and validation. Almost always the popularity of an existing language and which languages the implementer knows is the driving force behind which methods and languages to choose. This means that ABS will continue to be implemented in object-oriented programming languages and many perfectly well functioning models will be created by it in the future. Although they all suffer from the same issues mentioned in the introduction this doesn't matter as they are not of central importance to most of them. Nonetheless we think our work is still essential and necessary as it may start a slow paradigm-shift and opens up the minds of the ABS community to a more functional and formal way of approaching and implementing agent-based models and simulations and recognizing the benefits one gets automatically from it by doing so.

6.2 What we are not doing

Because of this highly interdisciplinary topic we explicitly mention what we do not want to undertake in this PhD. First we don't want to develop another language for formal agent-specification which needs to be compiled or used in some fancy tool - we want to put it directly into Haskell, building on the existing facilities. Second, we are not developing a new economic theory about decentralized bilateral bartering, we take the existing theory and existing agent-based models and apply our methods to them. Third, we don't want to use fancy statistics and number juggling for comparing validating and verifying models: we want

structural comparison (category-theory).

References

[1] Graham, P. *Hackers & Painters: Big Ideas from the Computer Age*. "O'Reilly Media, Inc.", 2004. Google-Books-ID: shycAgAAQBAJ.

Appendices

Appendix A

Pure Functional Epidemics

Pure functional epidemics

An Agent-Based Approach

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

Abstract

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

So far mainly object-oriented techniques and languages have been used in ABS. Using the SIR model of epidemiology, which allows to simulate the spreading of an infectious disease through a population, we show how to use Functional Reactive Programming to implement ABS. With our approach we can guarantee the reproducibility of the simulation already at compile time, which is not possible with traditional object-oriented languages. Also, we claim that this representation is conceptually 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, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The traditional approach to Agent-Based Simulation (ABS) has so far always been object-oriented techniques, due to the influence of the seminal work of Epstein et al [7] in which the authors claim "[..] object-oriented programming to be a particularly natural development environment for Sugarscape specifically and artificial societies generally [..]" (p. 179). This work established the metaphor in the ABS community, that *agents map naturally to objects* [17] which still holds up today.

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

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

Over the course of four steps, we derive all necessary concepts required for a full agent-based implementation. We start from a very simple solution running in the Random Monad which has all general concepts already there and then refine it in various ways, making the transition to Functional Reactive Programming (FRP) [31] and to Monadic Stream Functions (MSF) [21].

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.

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

Section 2 discusses related work. In section 3 we introduce functional reactive programming, arrowized programming and monadic stream functions, because our approach builds heavily on these concepts. Section 4 defines agent-based simulation. In section 5 we introduce the SIR model of epidemiology as an example model to explain the concepts of ABS. The heart of the paper is section 6 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 7 and point to further research in section 8.

2 Related Work

The amount of research on using pure functional programming with Haskell in the field of ABS has been moderate so far. Most of the papers are more related to the field of Multi Agent Systems and look into how agents can be specified using the belief-desire-intention paradigm [6], [27], [12].

A library for Discrete Event Simulation (DES) and System Dynamics (SD) in Haskell called *Aivika 3* is described in the technical report [26]. It is not pure, as it uses the IO Monad under the hood and comes only with very basic features for event-driven ABS, which allows to specify simple state-based agents with timed transitions.

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

The authors of [30] present a domain-specific language for developing functional reactive agent-based simulations. This language called FRABJOUS is human readable and easily understandable by domain-experts. It is not directly implemented in FRP/Haskell but is compiled to Yampa code which they claim is also readable. This supports that FRP is a suitable approach to implement ABS in Haskell. Unfortunately, the authors do not discuss their mapping of ABS to FRP on a technical level, which would be of most interest to functional programmers.

3 Background

3.1 Functional Reactive Programming

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

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

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

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

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

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

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.

3.2 Arrowized programming

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

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

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

To show how arrowized programming works, we implement a simple signal function, which calculates the acceleration of a falling mass on its vertical axis as an example [22].

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

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

3.3 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 [21], [20]:

```
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*.

4 Defining Agent-Based Simulation

Agent-Based Simulation (ABS) is a methodology to model and simulate a system where the global behaviour may be unknown but the behaviour and interactions of the parts making up the system is 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 [25], [32], [15]:

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

5 The SIR Model

To explain the concepts of ABS and of our pure functional approach to it, we introduce the SIR model as a motivating example and use-case for our implementation. It is a very

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

well studied and understood compartment model from epidemiology [13] which allows to simulate the dynamics of an infectious disease like influenza, tuberculosis, chicken pox, rubella and measles spreading through a population.

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

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) [23]. 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:

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

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

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

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

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

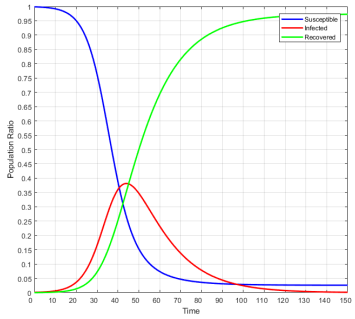


Figure 2. Dynamics of the SIR compartment model using the System Dynamics approach. Population Size N = 1,000, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time-steps.

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

An Agent-Based approach

The SD approach is inherently top-down because the behaviour of the system is formalized in differential equations. This requires that the macro behaviour of the system is known a priori which may not always be the case. In the case of the SIR model we already have a top-down description of the system in the form of the differential equations from SD. We want now to derive an agent-based approach which exhibits the same dynamics as shown in Figure 2.

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

The fundamental difference is that SD is operating on averages, treating the population completely continuous which results in non-discrete values of stocks e.g. 3.1415 infected persons. The approach of mapping the SIR model to an ABS is to discretize the population and model each person in the population as an individual agent. The transitions between the states are no longer happening according to continuous differential equations but due to discrete events caused both by interactions amongst the agents and time-outs. Besides the already mentioned differences, the true advantage of ABS becomes now apparent: with it we can incorporate spatiality as shown in section 6.4 and simulate heterogenity of population e.g. different sex, age,... Note that the latter is theoretically possible in SD as well but with increasing number of population properties, it quickly becomes intractable.

According to the model, every agent makes *on average* contact with β random other agents per time unit. In ABS

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

This results in the following agent behaviour:

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

6 Deriving a pure functional approach

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

The authors of [28] 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].

6.1 Naive beginnings

We start by modelling the states of the agents:

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

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

```
type Time      = Double
type TimeDelta = Double
```

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

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

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

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

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

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

```
runAgent :: RandomGen g
  => TimeDelta -> Agents -> SIRAgent -> Rand g SIRAgent
runAgent _  as   (Susceptible, _) = susceptibleAgent as
```

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

```
runAgent dt _  a@(Infected    , _) = return (infectedAgent dt a)
runAgent _  _  a@(Recovered   , _) = return a
```

An agent gets fed the states of all agents in the system from the previous time-step so it can draw random contacts - this is one, very naive way of implementing the interactions between agents.

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

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

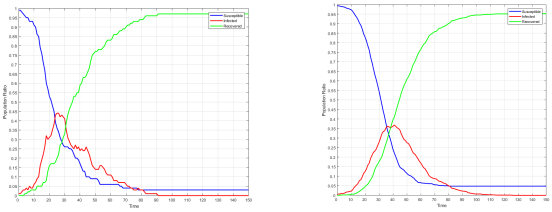
```
susceptibleAgent :: RandomGen g => Agents -> Rand g SIRAgent
susceptibleAgent as = do
    -- draws from exponential distribution
    rc <- randomExpM (1 / contactRate)
    cs <- forM [1..floor rc] (const (makeContact as))
    if elem True cs
      then infect
      else return (Susceptible, 0)
 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

    infect :: RandomGen g => Rand g SIRAgent
    infect = randomExpM (1 / illnessDuration)
               >>= \rd -> return (Infected, rd)
```

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

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

6.1.1 Results

When running our naive implementation with increasing population sizes we get the dynamics as seen in Figure 3. With increasing number of agents [14] our solution becomes increasingly smoother and approaches the SD dynamics from Figure 2 but doesn't quite match them because we are undersampling the contact-rate. We will will address this problem in the next section.



(a) 100 Agents **(b)** 1,000 Agents

Figure 3. Naive simulation of SIR using the agent-based approach. Varying population size, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time-steps with fixed $\Delta t = 1.0$.

6.1.2 Discussion

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

- Time - the simulation occurs over virtual time which is modelled explicitly divided into *fixed* Δt where at each step all agents are executed.
- Agents - we implement each agent as an individual, with the behaviour depending on its state.
- Feedback - the output state of the agent in the current time-step t is the input state for the next time-step $t + \Delta t$.
- Environment - as environment we implicitly assume a fully-connected network (complete graph) where every agent 'knows' every other agent, including itself and thus can make contact with all of them.
- Stochasticity - it is an inherently stochastic simulation, which is indicated by the Random Monad type and the usage of *randomBoolM* and *randomExpM*.
- Deterministic - repeated runs with the same initial random-number generator result in same dynamics. This may not come as a surprise but in Haskell we can guarantee that property statically already at compile time because our simulation runs in the Random Monad and *not* in the IO Monad. This guarantees that no external, uncontrollable sources of randomness can interfere with the simulation.
- Dynamics - with increasing number of agents the dynamics smooth out [14].

Nonetheless our approach has also weaknesses and dangers:

1. Δt is passed explicitly as argument to the agent and needs to be dealt with explicitly. This is not very elegant and a potential source of errors - can we do better and find a more elegant solution?

2. The way our agents are represented is not very modular. The state of the agent is explicitly encoded in an ADT and when processing the agent, the behavioural function always needs to distinguish between the states. Can we express it in a more modular way e.g. continuations?

We now move on to the next section in which we will address these points and the under-sampling issue.

6.2 Adding Functional Reactive Programming

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

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

6.2.1 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:

```
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*:

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

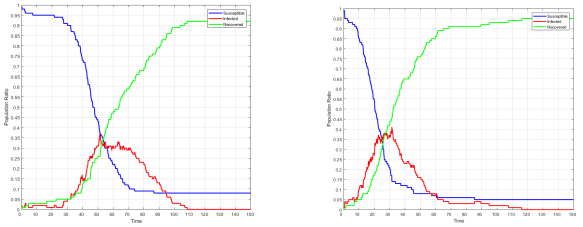
What we need to implement next is a closed feedback-loop. Fortunately, [16], [5] discusses implementing this in Yampa. The function *stepSimulation* is an implementation of such a closed feedback-loop. It takes the current signal functions and states of all agents, runs them all in parallel and returns the new agent states of this step. Yampa provides the *dpSwitch* combinator for running signal functions in parallel, which is quite involved and discussed more in-depth in section 3. It allows us to recursively switch back into the *stepSimulation* with the continuations and new states of all the agents after they were run in parallel. Note the use of *notYet* which is required because in Yampa switching occurs immediately at $t = 0$.

```haskell
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)
```

6.2.2 Results

The function which drives the dynamics of our simulation is *occasionally*, which randomly generates an event on average with a given rate following the exponential distribution. To arrive at the correct dynamics, this requires us to sample *occasionally*, and thus the whole system, with small enough Δt which matches the rate. If we choose a too large Δt, we loose events which will result in dynamics which do not approach the SD dynamics sufficiently enough, see Figure 4.

Clearly by keeping the population size constant and just increasing the Δt results in a closer approximation to the SD dynamics. 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.



(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 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.

6.2.3 Discussion

By moving on to FRP using Yampa we made a huge improvement in clarity, expressivity and robustness of our implementation. State is now implicitly encoded, depending on which signal function is active. Also by using explicit time-semantics with *occasionally* we can achieve extremely fine grained stochastics. Compared to drawing a random number of events we create only a single event or none at all. This requires to sample the system with a much smaller Δt: we are treating it as a 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 [21] which allows FRP within a monadic context.

6.3 Generalising to Monadic Stream Functions

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

6.3.1 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, except from a few type-declarations, 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
```

6.3.2 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 get rid of the RandomGen arguments to our functions and run the whole simulation within the Random Monad *again* just as we started but now with the full features functional reactive programming. We start by re-defining the SIRMonad and SIRAgent:

```
type SIRMonad g = Rand g
type SIRAgent g = 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*.

6.3.3 Discussion

So far making the transition to MSFs does not seem as compelling as making the move from the Random Monad to FRP in the beginning. Running in the Random Monad within FRP is convenient but we could achieve the same with passing RandomGen around as we already demonstrated. In the next step we introduce the concept of a read/write environment which we realise using a StateT monad. This will show the real benefit of the transition to MSFs as without it, implementing a general environment access would be quite cumbersome.

6.4 Adding an environment

In this step we will add an environment in which the agents exist and through which they interact with each other. This is a fundamental different approach to agent interaction but is as valid as the approach in the previous steps.

In ABS agents are often situated within a discrete 2D environment [7] which is simply a finite NxM grid with either a Moore or von Neumann neighbourhood (Figure 5). Agents are either static or can move freely around with cells allowing either single or multiple occupants.

We can directly map the SIR model to a discrete 2D environment by placing the agents on a corresponding 2D grid with an unrestricted neighbourhood. The behaviour of the agents is the same but they select their interactions directly from the environment. Also instead of feeding back the states of all agents as inputs, agents now communicate through the environment by revealing their current state to their neighbours by placing it on their cell. Agents can read the states of all their neighbours which tells them if a neighbour

(a) von Neumann **(b)** Moore

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

is infected or not. This allows us to implement the infection mechanism as in the beginning. For purposes of a more interesting approach, we restrict the neighbourhood to Moore (Figure 5b).

6.4.1 Implementation

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

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

Next we redefine our monad stack and agent signal function. We use a StateT transformer on top of our Random Monad from 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 SIRMonad g = StateT SIREnv (Rand g)
type SIRAgent g = SF (SIRMonad g) () ()
```

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

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

```
                returnA -< ((), Event ()))
            else returnA -< ((), NoEvent)
        _       -> returnA -< ((), NoEvent))
```

Querying the neighbourhood is done using the *neighbours :: SIREnv -> Disc2dCoord -> Disc2dCoord -> [Disc2dCoord] -> [SIRState]* function. It takes the environment, the coordinate for which to query the neighbours for, the dimensions of the 2D grid and the neighbourhood information and returns the data of all neighbours it could find. Note that on the edge of the environment, it could be the case that fewer neighbours than provided in the neighbourhood information will be found due to clipping.

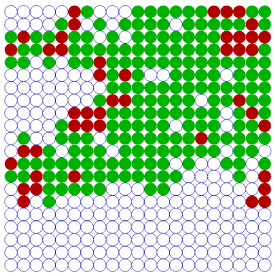
The behaviour of an infected agent is nearly the same as in the previous step, with the difference that upon recovery the infected agent updates its state in the environment from Infected to Recovered.

Running the simulation with MSFs works slightly different. The function *embed* we used before is not provided by BearRiver but by Dunai which has important implications. Dunai does not know about time in MSFs, which is exactly what BearRiver builds on top of MSFs. It does so by adding a ReaderT Double which carries the Δt. This is the reason why we need lifts e.g. in case of getting the environment. Thus *embed* returns a computation in the ReaderT Double Monad which we need to peel away using *runReaderT*. This then results in a StateT computation which we evaluate by using *evalStateT* and an initial environment as initial state. This then results in another monadic computation of the Random Monad type which we evaluate using *evalRand* which delivers the final result. 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.

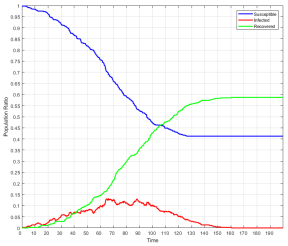
```
runSimulation :: RandomGen g => g -> Time -> DTime
  -> SIREnv -> [(Disc2dCoord, SIRState)] -> [SIREnv]
runSimulation g t dt env as = evalRand esRand g
  where
    steps    = floor (t / dt)
    dts      = replicate steps ()
    -- initial SFs of all agents
    sfs      = map (uncurry sirAgent) as
    -- running the simulation
    esReader = embed (stepSimulation sfs) dts
    esState  = runReaderT esReader dt
    esRand   = evalStateT esState env
```

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

```
stepSimulation :: RandomGen g
  => [SIRAgent g] -> SF (SIRMonad g) () SIREnv
stepSimulation sfs = MSF (\_ -> do
  -- running all SFs with unit input
  res <- mapM (`unMSF` ()) sfs
  -- extracting continuations, ignore output
  let sfs' = fmap snd res
  -- getting environment of current step
  env <- get
  -- recursive continuation
```



(a) $t = 100$



(b) Dynamics over time

Figure 6. Simulating the agent-based SIR model on a 21x21 2D grid with Moore neighbourhood (Figure 5b), a single infected agent at the center and same SIR parameters as in Figure 2. Simulation run until $t = 200$ with fixed $\Delta t = 0.1$. Last infected agent recovers shortly after $t = 160$. The susceptible agents are rendered as blue hollow circles for better contrast.

```
  let ct = stepSimulation sfs'
  return (env, ct))
```

6.4.2 Results

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

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

6.4.3 Discussion

At first the environment approach might seem a bit overcomplicated and one might ask what we have gained by using an unrestricted neighbourhood where all agents can contact all others. The real win is that we can introduce arbitrary restrictions on the neighbourhood as shown 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 the SD approach and allows to simulate much more realistic scenarios.

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 [22]. Also by using FRP we gain all the benefits from it and can use research on testing, debugging and exploring FRP systems [22], [19].

Issues

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

Despite the strengths and benefits we get by leveraging on FRP, there are errors that are not raised at compile-time, e.g. we can still have infinite loops and run-time errors. This was for example investigated by [24] 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 [29], [2], [1].

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

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

Acknowledgments

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

References

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

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

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

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

[5] Antony Courtney, Henrik Nilsson, and John Peterson. 2003. The Yampa Arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell (Haskell '03)*. ACM, New York, NY, USA, 7–18. https://doi.org/10.1145/871895.871897

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

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

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

[9] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*. ACM, New York, NY, USA, 12–1–12–55. https://doi.org/10.1145/1238844.1238856

[10] John Hughes. 2000. Generalising Monads to Arrows. *Sci. Comput. Program.* 37, 1-3 (May 2000), 67–111. https://doi.org/10.1016/S0167-6423(99)00023-4

[11] John Hughes. 2005. Programming with Arrows. In *Proceedings of the 5th International Conference on Advanced Functional Programming (AFP'04)*. Springer-Verlag, Berlin, Heidelberg, 73–129. https://doi.org/10.1007/11546382_2

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

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

[14] Charles M. Macal. 2010. To Agent-based Simulation from System Dynamics. In *Proceedings of the Winter Simulation Conference (WSC '10)*. Winter Simulation Conference, Baltimore, Maryland, 371–382. http://dl.acm.org/citation.cfm?id=2433508.2433551

[15] C. M. Macal. 2016. Everything you need to know about agent-based modelling and simulation. *Journal of Simulation* 10, 2 (May 2016), 144–156. https://doi.org/10.1057/jos.2016.7

[16] Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 51–64. https://doi.org/10.1145/581690.581695

[17] Michael J. North and Charles M. Macal. 2007. *Managing Business Complexity: Discovering Strategic Solutions with Agent-Based Modeling and Simulation*. Oxford University Press, USA. Google-Books-ID: gRATDAAAQBAJ.

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

[19] Ivan Perez. 2017. Back to the Future: Time Travel in FRP. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. ACM, New York, NY, USA, 105–116. https://doi.org/10.1145/3122955.3122957

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

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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.

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

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

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

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

[30] Ivan Vendrov, Christopher Dutchyn, and Nathaniel D. Osgood. 2014. Frabjous: A Declarative Domain-Specific Language for Agent-Based Modeling. In *Social Computing, Behavioral-Cultural Modeling and Prediction*, William G. Kennedy, Nitin Agarwal, and Shanchieh Jay Yang (Eds.). Number 8393 in Lecture Notes in Computer Science. Springer International Publishing, 385–392. http://link.springer.com/chapter/10.1007/978-3-319-05579-4_47 DOI: 10.1007/978-3-319-05579-4_47.

[31] 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

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

Received March 2018

Appendix B

Questions & Answers

TODO: update and adopt to 2nd year

In this chapter I give answers to anticipated questions and objections about my research direction and vision of doing pure functional ABS [1].

Doesn't NetLogo provide all this? Fair point, NetLogo can be seen as a functional language. Problem: side-effects, global state, dynamic, very difficult to verify. Main benefits: quick and easy to learn, for beginners.

Why another formalism, don't we have already enough of them? It is only partly another formalism - the important fact is, that it is directly built into Haskell, thus leveraging on a pure functional programming language directly than having some formal language which is then translated to machine code.

Pure functional programming? You must be kidding! Objects are so close to Agents! Exactly this is the problem: objects are in fact very close to agents but they have also very important differences which are quickly cast aside when implementing ABS in e.g. Java leading to problems like shared state.

What about parallelism and concurrency? Although very important, they are not of primary interest in this PhD. It is very well known that pure functional programming is exceptionally well suited in implementing parallel programs (no side-effects) and also concurrent ones (through STM, allowing composition of concurrency) and we have looked into this a bit in the paper on programming paradigms in ABS, so all of this can be applied to our research as well.

[1] They are not always posed in a dead-serious way but as it is a quite controversial topic - ABS should be done object-orientated after all huh? - I think it is appropriate. Also some objections were raised in exactly this way.

You mentioned performance and space-leaks as weaknesses? Yes this is probably the largest issue to date as even problems with small number of agents (10,000) are getting very slow. Although performance is not of interest here we are well aware that slow software is simply not used, thus we are looking for ways in improving the speed.

You mentioned the difficulty of debugging in weaknesses of pure functional programming, how can you be sure your program is correct? Good point but having the ability to step through a program alone does not guarantee the correctness of a program. In Haskell the static type system enforces already very much and prevents bugs which are normal in e.g. Java, also the lack of implicit side-effects makes programs much less prone to errors. In the end there is also reasoning techniques and QuickCheck. Thus we have much more power at hand in detecting bugs and ensuring correctness than classic oop languages like Java or C++.

So you say ABS should be done in Haskell using your library and using your techniques and object-orientation sucks? You're mixing things up but: it depends. If you are familiar with Haskell and like functional programming, go for it. If you are an oop-disciple, who can code the most complex ABS just don't. If you need correctness and verification in your simulation then definitely go for it.

Nobody has done it before? Probably for a good reason, object-oriented is the way to go for ABS! At least to my knowledge, backed-up by a thorough literature-review, no one has done a proper treatment of ABS in a pure functional way. My original motivation was just one thing: curiosity! No one has done it and I want to find out how it can be done, what its benefits are, what its weaknesses are. Just to claim oo is the *right* way is not a serious claim unless you haven't tried other ways and compared them. I have done both ways in-depth, have you?

What are the benefits? Simulations are more likely to be correct, better support for verification, no gap between specification and code, easier to replicate.

What are the weaknesses? Performance, steep learning curve if you don't know functional programming.

Why not using Erlang? Its functional, it has the actor model built in? Yes, Erlang would be nearly perfect but the non-determinism and asynchronous nature of the messaging is the main problem which can hardly be overcome. If you don't need determinism (as in every run may have different dynamics despite using the same RNG) in your model, then Erlang (or Scala) is definitely an interesting option.

But why functional? Is this unique to functional? All this can be done in principle in other languages! In principle you can implement an ABS in assembly, machine code, lambda calculus or even a Turing machine. The point is that the languages in which we program shape and define how we think of a problem and pure functional programming makes you think in a much more high level and abstract way which is much more suitable to validation and verification - after all in functional programming you think about *what* something is instead of *how* to compute it. Also it allows to draw parallels and map concepts from category theory (yes yes of course you could do that in principle in all the above cited languages. Also in principle I could drive in a wheelchair from Chile to Alaska).

I am a very experienced programmer in language *Blob*, Haskell doesn't have feature X of *Blub* at all! Also you could do anything in any Turing-complete language, a good programmer would not make these serious kind of mistakes, a good programmer does not use this bad language-feature, you could also program functional *Blub*, you could do that somehow also in *Blub*! Just let me quote Paul Graham from his book *Hackers & Painters* [1]: "Programmers get very attached to their favorite languages, and I don't want to hurt anyone's feelings, so to explain this point I'm going to use a hypothetical language called Blub. Blub falls right in the middle of the abstractness continuum. It is not the most powerful language, but it is more powerful than Cobol or machine language. As long as our hypothetical Blub programmer is looking down the power continuum, he knows he's looking down. Languages less powerful than Blub are obviously less powerful, because they're missing some feature he's used to. But when our hypothetical Blub programmer looks in the other direction, up the power continuum, he doesn't realize he's looking up. What he sees are merely weird languages. He probably considers them about equivalent in power to Blub, but with all this other hairy stuff thrown in as well. Blub is good enough for him, because he thinks in Blub. When we switch to the point of view of a programmer using any of the languages higher up the power continuum, however, we find that he in turn looks down upon Blub. How can you get anything done in Blub? It doesn't even have y."

But aren't all these Languages you're talking about (Java, Haskell,...) Turing Complete? So aren't they equally powerful? I've heard this line of argumentation quite often - for me it resembles half-knowledge about programming languages and the origins, concepts and limitations of computation. But ok... you really asked the question, so let me give you an answer: in theory they are equally powerful yes but in practise absolutely not. The Turing Machine (from which the term Turing Completeness comes) AND the Lambda Calculus were abstract concepts conceived around the same time, to describe the principles of computation and theoretically explore their limitations. Of course it is possible to "program" in both a Turing Machine and the Lambda

Calculus - have you ever done it? I really doubt that, otherwise you would not ask such a question. Also I doubt you were ever confronted with real-world industry-challenging programming tasks. So... I've "programmed" a small toy example in a Turing Machine and I've programmed a few more examples in the untyped Lambda Calculus. It's just not feasible, the level is too low and complexity takes over quickly as one wants to solve more complicated problems than very simple arithmetic on natural numbers. Also do you know what turing completeness means? It only talks about the fundamental computational limitations of a formal language, nothing more. To say all languages are equal in principle because they are turing complete would amount to the same as saying my simple hand calculator is equal in principle to a supercomputer because both are turing complete. In very theory yes but in practice this claim is ridiculous. So... don't lets go there, lets try again with a different question.

So you're implying is that different Languages have different power? Yes. Let me give you an example: I guess you would agree that Machine Language is not as powerful as Assembly and this in turn is not as powerful as C. With powerful i mean: how easily the language let me express abstractions and solve problems *in a specific domain*.

But wait... Java supports now lambda expressions, so you can do functional programming in Java now! Well... no. Just adding lambdas to a language does not make it functional. The main criterion to classify a language as functional is whether it is based on the lambda calculus or not - this is clearly not the case in Java as it still remains an imperative language with features for object-oriented programming. Of course, by adding lambdas, programming in a functional *style* is now much easier but you have to very clearly distinguish between a functional *style* and actual functional programming. After all, it is possible to program in an object-oriented style in both C and Haskell but that does not make those language object-oriented! So coming back to your question: Java added a few features in Version 8 which support programming in the functional style: Lambdas, Method References and Aggregate Operation. All come with quite lots of ceremony - as is usual for Java - but are indeed an important step towards functional style programming. So having these mechanisms at hand allows one to program functionally in Java? Yes, I

looked at quite a number of Blogs, Tutorials, Java Documentation, JDK, [2]

My conclusion is the following: the Aggregate Operations in combination with the Lambdas are a real nice and powerful way to concisely operate on collections, which was always tedious in Java (and OOP in general). Apart from that, emulating a (pure) functional style in Java becomes very messy and ugly and needs still lots of ceremony (funny enough as one expects to reduce ceremony by the use of lambdas) - for obvious reasons: the language is still imperative OOP. So finally: use the Aggregate Operations to manipulate Collections and stick to established OO techniques otherwise or completely switch to Haskell if you really want to go functional.

[2]

- Introduction to functional programmin in Java 8 Part 0 - 4: `https://flyingbytes.github.io/programming/java8/functional/part0/2017/01/16/Java8-Part0.html`
- https://dzone.com/articles/functional-programming-java-8
- Oracle Tutorial on Lambda Expressions: `https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`
- Oracle Tutorial on Method Reference: `https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html`
- Oracle Tutorial on Aggregate Operations: `https://docs.oracle.com/javase/tutorial/collections/streams/index.html`