Pure Functional Programming in Agent-Based Simulation

by Jonathan Thaler

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy

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

July 10, 2019

Contents

Abstract

This thesis systematically investigates the use of the *pure* functional programming paradigm for implementing Agent-Based Simulations (ABS) and the benefits and drawbacks when doing so. As language of choice, Haskell is used due to its modern, *pure* nature and increasing use in real-world applications. First, the thesis explore *how* to implement ABS pure functionally, discussing both a time- and event-driven approach. In each case arrowized Functional Reactive Programming plays a fundamental role to derive fundamental abstractions and concepts. As use cases the well known explanatory agent-based SIR and the exploratory Sugarscape model are used. Then the thesis explores *why* it is of benefit to implement ABS pure functionally, where it focuses on parallelism and concurrency and property-based testing. In the parallelism part, the main focus is on how to speedup the simulation but keeping it still pure, whereas in the concurrency part, Software Transactional Memory is used at the price of purity. Property-based testing is used to show how to encode full agent specifications, model invariants and do verification of the explanatory model directly in code. Further, hypothesis testing for the exploratory Sugarscape model is shown.

TODO first version for others to read: until 18th July (before going to SummerSim conference)

1. PRINT OUT full proof reading by me but this time back-to-front, focus on:
   - types / function names / monad names texttt
   - instead of italics use texttt use it for types, functions and emphasising certain word, it looks better
   - when citing multiple references, make sure they are in ascending order of their reference number
   - be consistent about hyphens and rather avoid them
   - move links from footnote into references and put full implementations of existing models from phd repo in separate gitrepso. e.g. sugarscape, SIR event and time as and reconcile all code into thesis code folder?
   - for references add hyperlinks using howpublished in case it is a blog or website (e.g. stephen diel what i wish...)
2. write a GOOD and STRONG abstract, the current one sucks
3. introduce abbreviations once (ABS, MAS, STM, FRP)?
4. get rid of all e.g.
5. epigraphs for all chapters?
6. what about: time-flow, pro-active, multi-processing,

TODO "cleaning up / polishing" milestone until end of August 2019

1. reading by my supervisors
2. reading by james hey
3. reading by thomas schwarz for content
4. proof-reading by professional lector (Karly), focus on
   - s vs 's vs. s' e.g. Agents vs. Agents's vs. Agents'
5. check for any violation of the guidelines https://www.nottingham.ac.uk/physics/currentstudents/pg-submissionguide.aspx

To my parents Irmentraud and Wolfgang.
For their unconditional love and support throughout all my life.

Acknowledgements

Thanks go to my first supervisor Peer-Olaf Siebers, who always very patiently reminded me that a Ph.D. is not about to change the world but learning how to do research on my own. He was a strong guidance throughout my three years in Nottingham and I could not have hoped for a better and more dedicated first supervisor.

I am also thankful for my second supervisor Thorsten Altenkirch, who gave strong and sometimes brutal feedback about the technical details of my approaches. Due to the fact that his main interest is a rather theoretical spin on functional programming and computing, I am deeply grateful for his strong support of my rather practical approach to functional programming.

I am in debt to the whole Functional Programming Lab at UoN, for welcoming me in their midst despite my lack of specific theoretical background. I owe them many open and deep discussions which resulted in new insights. Further, presenting at their *FP Lunch* was always a challenging but highly rewarding activity, always resulting in valuable feedback.

I am especially in debt to Ivan Perez for always having an open ear for questions and valuable discussions about his research, without this Ph.D. would have probably developed a very different spin.

Many thanks go to Martin Handley, James Hey and Thomas Schwarz for many discussions, feedback and proof reading of my papers and my thesis.

Thanks go also to Julie Greensmith for valuable discussions and pointing me into right directions at important stages of the Ph.D.

Part I:

Preliminaries

Chapter 1

Introduction

The traditional approach to Agent-Based Simulation (ABS) has so far always been object-oriented, due to the influence of the seminal Sugarscape model [47], 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* [113], which still holds up today.

This thesis challenges this metaphor and explores ways of approaching ABS through the *pure* functional programming paradigm using the language Haskell. To the best of the authors knowledge, it is the first one to do so on a *systematical* level, developing a foundation by presenting fundamental concepts and advanced features to show how to leverage the paradigms benefits [76] to make them available when implementing ABS functionally. By doing this, the thesis shows *how* to implement ABS purely functional and *why* it is of benefit of doing so, what the drawbacks are and also when a pure functional approach should *not* be used. This aim leads to the thesis' general research question of *how to implement ABS purely functional and what the benefits and drawbacks are when doing so.*

Struggles with established ABS approaches are described in [9], where Axelrod reports the vulnerability of ABS to misunderstanding. Due to informal specifications of models and change requests among members of a research team, bugs are very likely to be introduced. He also reported how difficult it was to reproduce the work of [7], which took the team four months due to inconsistencies between the original code and the published paper. The consequence is that counter-intuitive simulation results can lead to weeks of checking whether the code matches the model and is bug-free as reported in [8].

The same problem was reported in [84], which tried to reproduce the work of Gintis [59]. In his work, Gintis claimed to have found a mechanism in bilateral decentralized exchange, which resulted in Walrasian General Equilibrium without the neo-classical approach of a tatonement process through a central auctioneer. This was a major breakthrough for economics as the theory of Wal-

rasian General Equilibrium is non-constructive as it only postulates the properties and existence of the equilibrium [32] but does not explain the process and dynamics through which this equilibrium can be reached or constructed - Gintis seemed to have found this very process.

The authors [84] failed to reproduce the results and were only able to solve the problem by directly contacting Gintis which provided the code, the definitive formal reference [1]. It was found that there was a bug in the code leading to the unexpected results, which were seriously damaged through this error. They also reported ambiguity between the informal model description in Gintis paper and the actual implementation. This lead to a research in a functional framework for agent-based models of exchange as described in [21], which tried to give a very formal functional specification of the model coming very close to an implementation in Haskell. The failure of Gintis was investigated more in depth in the thesis by [48] who got access to Gintis code of [59]. They found that the code didn't follow good object-oriented design principles (all was public, code duplication) and - in accordance with [84] - discovered a number of bugs serious enough to invalidate the results.

Thus it is clear that due to the fact that ABS is almost always used for scientific research, producing often break-through scientific results, besides on converging both on standards for testing the robustness of implementations and on its tools, ABS need to be *free of bugs*, *verified against their specification*, *validated against hypotheses* and ultimately be *reproducible* [9].

This thesis claims that the ABS community needs functional programming because of its *scientific computing* nature, where results need to be reproducible and correct while simulations should be able to massively scale up as well. Due to the primarily computational character of most ABS models, not requiring asynchronous, non-deterministic IO or direct user interaction, a pure functional approach should be able to directly deliver these non-functional requirements. As pointed out above, the established object-oriented approach needs considerably high effort and might even fail to deliver these objectives due to its conceptually different approach to computing, as in general it lacks strong static typing, uses mutable shared state and has implicit side effects.

This thesis hypothesises that using pure functional programming for implementing ABS might be a remedy to this problem and would make the resulting simulations easier to test and verify, applicable to property-based testing, easy to add parallelism and concurrency, guaranteed to be reproducible already at compile time, have fewer potential sources of bugs and thus can raise the level of confidence in the correctness of an implementation to a new level. Further, it is well known that functional programming helps in structuring computation in a very clear and precise way, leading to a deeper understanding about problems. Thus, in this thesis the functional approach is regarded as a way to think and explore ABS in a more rigorous way, as a tool for developing abstractions and especially to develop a deeper and more complete understanding of the compu-

[1] It seems that by now, Gintis has made his code, written in Object Pascal, publicly available through his homepage `https://people.umass.edu/gintis/`

tational structure underlying ABS. It is the authors hope that this undertaking is to the whole benefit of the ABS discipline and will also feed back into the traditional object-oriented implementation techniques.

This thesis is *not* about comparing the object-oriented and pure functional programming paradigm in the context of implementing ABS. Such an approach would not be fruitful, as too much ink has been spilt already over which paradigm is better or worse. There seems to be no scientific evidence supporting that either one is truly superior over the other; each has its benefits and weaknesses and this thesis aims to finding out what they are when using pure functional programming in ABS.

1.1 Contributions

1. To the best knowledge of the author, this thesis is the first to *systematically* investigate the use of the functional programming paradigm, as in Haskell, to ABS, laying out in-depth technical foundations and identifying its benefits and drawbacks. Also, the use of functional programming, which focuses on explicit data-flow representation, is a strong match to scientific computing, which is data-centric as well. Thus, due to the increased interest in functional concepts added to object-oriented languages in recent years (lambda expressions and map, filter, reduce in Java 8, rise of functional frameworks in JavaScript, Pythons functional features,...), because of its established benefits in concurrent programming, testing and software development in general, presenting such foundational research gives this thesis significant impact. Further, a pure functional approach directly leads to less bugs and guaranteed reproducibility of repeated runs at compile time, resulting in implementations which are more likely to be correct, something of fundamental importance in all kind of scientific computing in general, thus giving this thesis considerable impact.

2. To the best knowledge of the author, this thesis is the first to show the use of Software Transactional Memory (STM) to implement concurrent ABS and its potential benefit over lock-based approaches. The use of STM is particularly compelling in pure functional programming because it can be guaranteed at compile time that retry semantics exclude non-repeatable persistent side effects. By showing how to employ STM it is possible to implement a simulation which allows massively large-scale ABS but without the low level difficulties of concurrent programming, making it easier and quicker to develop working and correct concurrent ABS models. The use of STM allows to approach concurrency still as a data-flow approach, without cluttering model code with concurrency semantics. Although purity is lost when using STM, it is still possible to retain certain guarantees about reproducibility, making it a highly attractive approach to concurrent scientific computing. Further, due to the increasing need for massively large-scale ABS in recent years [99], making this possible within a purely

functional approach as well, gives this thesis substantial impact.

3. To the best of the authors knowledge, this thesis is the first to present the use of property-based testing in ABS, which allows declarative specification testing of the implemented ABS directly in code with *automated* random test case generation. This is an addition to the established Test Driven Development process and a complementary approach to unit testing, ultimately giving the developers an additional, powerful tool to test the implementation on a more conceptual level. More specifically, the thesis shows how to encode full agent specifications and model invariants and do validation and verification including hypothesis testing with property-based testing. This should lead to simulation software which is more likely to be correct, thus making this a highly significant contribution with valuable impact.

1.2 Publications

Throughout the course of the Ph.D. five (5) papers were written and attempted to publish, out of which three (3) were accepted and published:

1. *The Art Of Iterating - Update Strategies in Agent-Based Simulation* [150]; submitted and accepted at the Social Simulation Conference 2017 - This paper derives four different update strategies and their properties possible in time-driven ABS and discusses them from a programming-paradigm agnostic point of view. It is the first paper which makes the very basics of update semantics clear on a conceptual level and is necessary to understand the options one has when implementing time- and event-driven ABS purely functional. Chapter 3 builds heavily on the contents of this paper. Further the introduction to ABS in Chapter 2.2 is covered to a large extent already in this paper.

2. *Pure Functional Epidemics* [149]; submitted and accepted at the IFL Conference 2018 - Using an agent-based SIR model, this paper establishes in technical detail *how* to implement time-driven ABS in Haskell using non-monadic FRP with Yampa and monadic FRP with Dunai. It outlines benefits and drawbacks and also touches on important points which were out of scope and lack of space in this paper but which will be addressed in more detail in this thesis. Chapter 4 is basically an identical copy of this paper, with minor extensions and restructuring. Further the introduction to FRP in Chapter 2.3 is covered to a large extent already in this paper.

3. *A Tale Of Lock-Free Agents* [151]; submitted to the TOMACS Journal in October 2018 and rejected in March 2019 - This paper is the first to discuss the use of Software Transactional Memory (STM) for implementing concurrent ABS both on a conceptual and on a technical level. It presents two case studies, with the agent-based SIR model as the first

and the Sugarscape being the second one. In both case studies it compares performance of STM and lock-based implementations in Haskell and object-oriented implementations of established languages. Although STM is now not unique to Haskell any more, this paper shows why Haskell is particularly well suited for the use of STM and is the only language which can overcome the central problem of how to prevent persistent side effects in retry semantics. Chapter 7 is based on the work of this paper with minor adjustments and extensions.

4. *The Agents' New Cloths? Towards Pure Functional Agent-Based Simulation* [152]; submitted and rejected at the Summer Simulation Conference 2019 - This paper summarizes the main benefits of using pure functional programming as in Haskell to implement ABS and discusses on a conceptual level how to implement it and also what potential drawbacks are and where the use of a functional approach is not encouraged. It is written as a conceptual review paper, which tries to 'sell' pure functional programming to the agent-based community without too much technical detail and parlance where it refers to the important technical literature from where an interested reader can start. The introduction to functional programming in Chapter 2.3 is covered to a large extent already in this paper.

5. *Show Me Your Properties! The Potential Of Property-Based Testing In Agent-Based Simulation* [153]; submitted and accepted at the Summer Simulation Conference 2019 - This paper introduces property-based testing on a conceptual level to agent-based simulation using the agent-based SIR model and the Sugarscape model as two case studies. Both Chapters 8 and 9 of Part IV and the Appendix B are substantially expanded discussions of the ideas presented in this paper.

1.3 Thesis structure

The thesis is divided into five parts which act as the thematic narrative throughout the text followed by an Appendix .

Part I opens the thesis by laying out the necessary prerequisites necessary to understand the ideas and motivation in the rest of the thesis.

Chapter 1 introduces the problem and presents the motivation, aim and hypotheses.

Chapter 2 presents related research and discusses the background necessary to understand the rest of the thesis. It presents a definition of ABS and gives an introduction to functional programming with advanced topics necessary to understand the concepts in this thesis. Further it discusses the methodology.

Chapter 3 discusses an architectural categorisation on how to implement ABS from a language-agnostic point of view.

Part II presents techniques of *how* to implement ABS in pure functional programming.

Chapter 4 derives a time-driven ABS implementation for an agent-based SIR model. Because it is the first chapter discussing how to implement ABS pure functionally, it goes quite into detail in order to lay out the basic concepts.

Chapter 5 presents an event-driven approach to ABS using an event-driven agent-based SIR and the highly complex Sugarscape model. It takes up the concepts derived in the previous Chapter 4, generalises them and pushes them forward to a more generic solution. Further it also gives a brief outline how to transform the time-driven SIR implementation into an event-driven one.

Part III presents parallel computation in ABS as the first of two parts of *why* pure functional ABS is of benefit.

Chapter 6 shows rather briefly how to achieve deterministic and pure parallelism in pure functional ABS.

Chapter 7 presents an in-depth discussion on how to implement concurrent ABS using Software Transactional Memory.

Part IV presents property-based testing in ABS as the second of two parts of *why* pure functional ABS is of benefit.

Chapter 8 shows how to use property-based testing to implement a full agent-specification test of the event- and time-driven SIR model in code and run it as property-tests.

Chapter 9 shows how to derive and encode invariants the simulations dynamics must uphold in property tests. It also shows how to compare the dynamics of two implementations of the same underlying model, namely the the time- and event-driven SIR implementations. Further it shows how to put model specifications into code and check them with property tests by comparing the System Dynamics simulation to the agent-based one.

Part V is the closing part which discusses and concludes the thesis.

Chapter 10 revisits and discusses initial motivation, aim and hypotheses. Further, it presents the drawbacks of the pure functional approach, discusses the Gintis case described in the Introduction and answers the question of whether agents map naturally to objects or not.

Chapter 11 concludes and presents further research.

Appendices contains additional material which relates to the overall research of this thesis but would be out of context in the respective chapters.

Appendix A shows a pure functional implementation of a System Dynamics (SD) SIR simulation using Functional Reactive Programming. It shows that it is possible to directly encode SD specification in pure functional code with extremely high guarantees in correctness. This implementation is used in Chapter 9 where the agent-based SIR dynamics are tested against the SD ones.

Appendix B contains a brief overview over the validation process we went through when trying to get our Sugarscape implementation from Chapter 5 in line with the results from the original specification [47]. Further we show how we can use property-based testing in an exploratory model to formulate and test hypotheses.

Appendix C discusses the deep question of whether it is possible to test or guarantee that a correct implementation of the SIR model does terminate or not. This question arises in the context of Chapters 9 and 8.

Chapter 2

Background

2.1 Related research and literature

The amount of research on using pure functional programming with Haskell in the field of ABS has been moderate so far. Most of the papers are related to the field of Multi-Agent Systems (MAS) and look into how agents can be specified using the belief-desire-intention paradigm [37, 89, 145].

A multi-method simulation library in Haskell called *Aivika 3* is described in the technical report [140]. It supports implementing Discrete Event Simulations (DES), System Dynamics and comes with basic features for event-driven ABS which is realised using DES under the hood. Further, it provides functionality for adding GPSS to models and supports parallel and distributed simulations. It runs within the `IO` Monad for realising parallel and distributed simulation but also discusses generalising their approach to avoid running in `IO`.

In his master thesis [16] the author investigates Haskells' parallel and concurrency features to implement (amongst others) *HLogo*, a Haskell clone of the NetLogo [165] simulation package, focusing on using Software Transactional Memory (STM) for a limited form of agent interactions. *HLogo* is a re-implementation of NetLogos API in Haskell where agents run within *IO* and thus can also make use of STM functionality. The benchmarks show that this approach does indeed result in a speedup especially under larger agent populations. The authors' thesis is one of the first on ABS using Haskell. Despite the concurrency and parallel aspect, this thesis approach is rather different: it avoids `IO` within the agents under all costs and explores the use of STM more on a conceptual level and compares case studies with lock-based and imperative implementations.

There exists some research [39, 138, 157] using the functional programming language Erlang [5] to implement concurrent ABS. The language is inspired by the Actor Model [1] and was created in 1986 by Joe Armstrong for Eriksson for developing distributed high reliability software in telecommunications. The Actor Model can be seen as quite influential to the development of the

concept of agents in ABS, which borrowed it from MAS [168]. It emphasises message-passing concurrency with share-nothing semantics (no shared state between agents), which maps nicely to functional programming concepts. The mentioned papers investigate how the actor model can be used to close the conceptual gap between agent specifications. Further, they show that using this kind of concurrency allows to overcome some problems of low level concurrent programming as well. Also [16] ported NetLogos API to Erlang mapping agents to concurrently running processes, which interact with each other by message passing. With some restrictions on the agent interactions the author could deliver a working implementation of the model, which shows that using concurrent message passing for parallel ABS is at least *conceptually* feasible. Despite the natural mapping of ABS concepts to such an actor language, it leads to simulations, where due to concurrency, despite same initial starting conditions, repeated runs might result in different dynamics.

The work [99] discusses a framework, which allows to run ABS on a GPU. Amongst others they use the SugarScape model [47] and scale it up to millions of agents on very large environment grids. They reported an impressive speedup of a factor of 9,000. Although their work is conceptually very different, this thesis draws inspiration from their work in terms of performance measurement and comparison to the Sugarscape model.

Using functional programming for DES was discussed in [89] where the authors explicitly mention the paradigm of Functional Reactive Programming (FRP) to be very suitable to DES.

A domain-specific language for developing functional reactive ABS was presented in [135, 158]. 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 Haskell code which they claim is also readable. This supports that FRP is a suitable approach to implement ABS in Haskell. Unfortunately, the authors do not discuss their mapping of ABS to FRP on a technical level, which would be of most interest to functional programmers.

Object-oriented programming and simulation have a long history together as the former one emereged out of Simula 67 [36], which was created for simulation purposes. Simula 67 already supported DES and was highly influential for today's object-oriented languages. Although the language was important and influential, in our research we look into different approaches, orthogonal to the existing object-oriented concepts.

Lustre is a formally defined, declarative and synchronous dataflow programming language for programming reactive systems [65]. While it has solved some issues related to implementing ABS in Haskell, it still lacks a few important features necessary for ABS. There seems to be no way of implementing an environment in Lustre as it is done in Chapters 4 and 5. Also, the language seems not to come with stochastic functions, which are but the very building blocks of ABS. Finally, Lustre does only support static networks, which is clearly a drawback in ABS in general where agents can be created and terminated dynamically during simulation.

In [20], the authors discuss the problem of advancing time in message-driven

agent-based socio-economic models. They formulate purely functional definitions for agents and their interactions through messages.

The authors of [21] are using functional programming as a specification for an agent-based model of exchange markets but leave the implementation for further research where they claim that it requires dependent types.

In his talk [146], Tim Sweeney CTO of Epic Games discussed programming languages in the development of game engines and scripting of game logic. Although the fields of games and ABS seem to be very different, Gregory [62] defines computer-games as *"[..] soft real-time interactive agent-based computer simulations"* (p. 9) and indeed, they have striking similarities: both are simulations which perform numerical computations and update objects in a loop either concurrently or sequential. In games these objects are called *game objects* and in ABS they are called *agents* but they are conceptually the same thing. Sweeney reports that reliability suffers from dynamic failure in languages like C++, for example random memory overwrites, memory leaks, out-of-bounds access of arrays, dereferencing null pointers, integer overflow, accessing uninitialized variables. He reports that 50% of all bugs in the game engine middleware Unreal can be traced back to such problems and presents dependent types as a potential rescue to those problems. The two main points Sweeney made were that dependent types could solve most of the run-time failures and that parallelism is the future for performance improvement in games. He distinguishes between pure functional algorithms which can be parallelised easily in a pure functional language and updating game objects concurrently using STM.

2.2 Agent-Based Simulation

This thesis understands ABS as a method and 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, 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 [101, 114, 139, 168]. Summarising, this thesis informally assumes the following about agents:

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

Epstein [46] identifies ABS to be especially applicable for analysing *"spatially distributed systems of heterogeneous autonomous actors with bounded information and computing capacity"*. Technically, ABS exhibits the following properties:

- Linearity and non-linearity - actions of agents can lead to non-linear behaviour of the system.
- Time - agents act over time, which is also the source of their pro-activity.
- State - agents encapsulate state, which can be accessed and changed during the simulation.
- Feedback loop - because agents act continuously and their actions influence each other and themselves in the future of subsequent time steps, feedback loops permeate every ABS.
- Heterogeneity - agents can have properties (age, height, sex,...) where the actual values can vary arbitrarily between individuals.
- Interactions - agents can be modelled after interactions with an environment and other agents.
- Spatiality and networks - agents can be situated within arbitrary environments, like spatial environments (discrete 2D, continuous 3D,...) or complex networks.

Note that there is no commonly agreed technical definition of ABS but the field draws inspiration from the closely related field of Multi-Agent Systems (MAS) [164, 168]. It is important to understand that MAS and ABS are two different fields where in MAS the focus is much more on technical details, implementing a system of interacting intelligent agents within a highly complex environment with the focus primarily on solving AI problems.

The field of ABS can be traced back to self-replicating von Neumann machines, cellular automata and Conway's Game of Life. The famous Schelling segregation model [134] is regarded as a pioneering example. ABS as a discipline was first picked up by social simulation, which explores social norms, institutions, reputation, elections and economics. Axelrod [8, 10] has called social simulation the third way of doing science, which he termed the *generative* approach, which is in opposition to the classical inductive (finding patterns in empirical data) and deductive (proving theorems). Thus, the generative approach can be seen as a form of empirical research and is a natural environment for studying social and interdisciplinary phenomena as discussed more in depth in the work of Epstein [45, 46]. He gives a fundamental introduction to agent-based social social simulation and makes the strong claim that *"If you didn't*

grow it, you didn't explain its emergence" [1]. Epstein puts much emphasis on the claim that ABS is indeed a scientific instrument as hypotheses which are investigated are empirically falsifiable. If the simulation exhibits an emergent pattern, then the model is *one* way of explaining it. On the other hand if it does not show the emergent pattern, then the hypothesis that the micro interactions amongst the agents generate the emergent pattern is falsified [2] and we have not found an explanation *yet*. So in summary, growing a phenomena is a necessary, but not sufficient condition for explanation [45].

The first large scale social ABS model which rose to some prominence was the *Sugarscape* model developed by Epstein and Axtell in 1996 [47]. Their aim was to *grow* an artificial society by simulation and connect observations in their simulation to phenomenon of real-world societies. It was this simulation which strongly advertised object-oriented programming to implement ABS. Due to this influence and also due to the general popularity of the object-oriented paradigm which started to rise in the early-to-mid 90s, object-oriented programming has become the de-factor standard in implementing ABS . We can distinguish between three categories of how ABS is implemented today:

1. Programming from scratch using object-oriented languages with Python, Java and C++ being the most popular ones.

2. Programming using a 3rd party ABS library using object-oriented languages where RePast and DesmoJ, both in Java, are the most popular ones.

3. Using a high-level ABS toolkit for non-programmers, which allow customisation through programming if necessary. By far the most popular one is NetLogo with an imperative programming approach followed by AnyLogic with an object-oriented Java approach.

To get a better idea and deeper understanding of ABS, the next sections present two different, well-known agent-based models to give examples of two different types: the explanatory SIR model and the exploratory Sugarscape model. Both are used throughout the thesis as use cases for developing pure functional ABS implementation techniques, concepts and test beds for Software Transactional Memory and property-based testing.

2.2.1 The SIR model

The *explanatory* SIR model is a very well studied and understood compartment model from epidemiology [93], which allows to simulate the dynamics of an infectious disease like influenza, tuberculosis, chicken pox, rubella and measles

[1] This can be seen as a fundamental constructivist approach to social science, which implies that the emergent properties are actually computable. When making connections from the simulation to reality, constructible emergence raises the question whether our existence is computable or not. When pushing this further, we can conjecture that the future of simulation will be simulated copies of our own existence, which potentially allows to simulate *everything*. This idea is not new and an interesting treatment of it can be found in [19, 141].

[2] This is fundamentally following Poppers theory of science [127].

Figure 2.1: States and transitions in the SIR compartment model.

spreading through a population. The reason for choosing this model is its simplicity as it is easy to understand fully but complex enough to develop basic concepts of pure functional ABS, which are then extended and deepened in the much more complex Sugarscape model of the next section.

In this model, people in a population of size N can be in either one of the 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 2.1.

This model was also formalized using System Dynamics (SD) [128]. In SD a system is modelled through differential equations, which allow expressing continuous systems, changing over time. They are solved by numerically integrating over time, which gives rise to the respective dynamics. The SIR model is modelled using the following equation, with the dynamics shown in Figure 2.2 .

$$\begin{aligned} \frac{\mathrm{d}S}{\mathrm{d}t} &= -infectionRate \\ \frac{\mathrm{d}I}{\mathrm{d}t} &= infectionRate - recoveryRate \\ \frac{\mathrm{d}R}{\mathrm{d}t} &= recoveryRate \end{aligned} \tag{2.1}$$

$$\begin{aligned} infectionRate &= \frac{I\beta S\gamma}{N} \\ recoveryRate &= \frac{I}{\delta} \end{aligned} \tag{2.2}$$

The approach of mapping the SIR model to an ABS is to discretise the population and model each person in the population as an individual agent. The transitions between the states are happening due to discrete events caused both by interactions amongst the agents and timeouts. The major advantage of ABS over SD is that it allows to incorporate spatiality and simulate heterogeneity of population e.g. different sex, age. This is not directly possible with other simulation methods of SD or DES [169].

In the ABS classification of [101], this model can be seen as an *Interactive ABMS*: agents are individual heterogeneous agents with diverse set character-

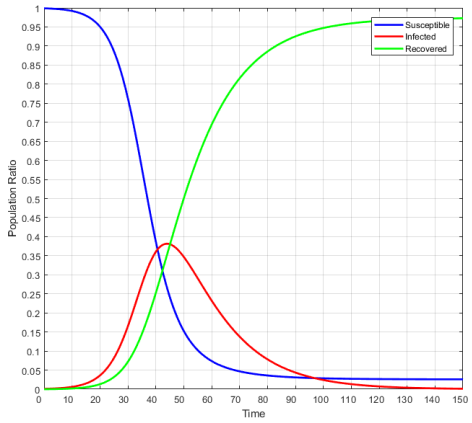


Figure 2.2: Dynamics of the SIR compartment model using the System Dynamics approach. Population Size $N = 1{,}000$, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time steps. Generated using our pure functional System Dynamics approach (see Appendix A).

istics; they have autonomic, dynamic, endogenously defined behaviour; interactions happen between other agents and the environment through observed states, behaviours of other agents and the state of the environment.

2.2.2 Sugarscape

The seminal Sugarscape model was one of the first models in ABS, developed by Epstein and Axtell in 1996 [47]. Their aim was to *grow* an artificial society by simulation and connect observations in their simulation to phenomenon observed in real-world societies, making it an *exploratory* model. In the model a population of agents move around in a discrete 2D environment, where sugar and spice grows, and interact with each other and the environment in many different ways. The main features of this model are (amongst others): searching, harvesting and consuming of resources, wealth and age distributions, population dynamics under sexual reproduction, cultural processes and transmission, combat and assimilation, bilateral decentralized trading (bartering) between agents with endogenous demand and supply, disease processes transmission and immunology.

The reasons for choosing the Sugarscape model as use case in this thesis are: it is quite well known in the ABS community; it was highly influential in sparking the interest in ABS; it is quite complex with non-trivial agent-interactions; the original implementation was done in Object Pascal and C with about 20.000 lines of code which includes GUI, graphs and plotting, where the authors used Object Pascal for programming the agents and C for low-level graphics [11]; the authors explicitly advocate object-oriented programming as

a good fit to ABS which begged the question whether and how well a pure functional implementation is possible.

In the ABS classification of [101], the Sugarscape can be seen as an *Adaptive ABMS*: agents are individual heterogeneous agents with diverse set characteristics; they have autonomic, dynamic, endogenously defined behaviour; interactions happen between other agents and the environment through observed states, behaviours of other agents and the state of the environment; agents can change their behaviour during the simulation through observing their own state and learning; populations can adjust their composition.

The full specification of the Sugarscape model itself fills a small book [47] of about 200 pages, so we will only give a very brief overview of the model in terms of actions which happen. Generally, the model is stepped in discrete, natural number time steps, also called ticks, where in each tick the following actions happens:

1. Shuffle all agents and process them sequentially. The reason why the agents are shuffled is to even-out the odds of being scheduled at a specific position - it is equally probable of being scheduled in any position. The semantics of the model follow the sequential update strategy (see Chapter 3) require to step the agents sequentially but ideally one wants to avoid any biases in ordering and pretend that agents act conceptually or statistically at the same time in parallel - the shuffling allows to do this by *running the agents sequentially which makes their behaviour appear statistically in parallel*. Every agent executes the following actions, where agents executed after the agent in the same tick, can already see the changes and interactions of preceding agents

   (a) The agent ages by 1 tick. An agent might have a maximum age, when reached will result in the removal of the agent from the simulation (see below).

   (b) Move to the nearest unoccupied site in sight with highest resource. In case of combat also sites occupied with agents from a different tribe are potential targets. Harvest all the resources on the site and in case of combat also reap the enemies resources or gather some combat reward. This is one of the primary reasons why the Sugarscape model needs to be stepped sequentially: because only one agent can occupy a site at a time, it would lead to conflicts when agents actually act at the same time.

   (c) Apply the agents' metabolism. Each agent needs to consume a given number of resources in each tick to satisfy its metabolism. The gathered resources can be stocked up during the harvesting process but if the agent does not have enough resources to satisfy its metabolism, it will be removed from the simulation (see below).

   (d) Apply pollution of the environment through the agent. Depending on how much the agent has harvested during its movement and con-

sumed in its metabolism process, it will leave a small fraction of pollution in the environment.

(e) Check if the agent has died from age or starved to death, in case it removes itself from the simulation and does not execute the next steps (the previous steps are executed independently from the age of the agent). Note that depending on the model configuration this could also lead to the re-spawning of a new agent which replaces the died agent.

(f) Engage with other neighbours in mating, which involves multiple synchronous interaction steps happening in the same tick: exchange of information and both agents agreeing on the mating action. If both agents agree to mate, the initiating agent spawns a new agent, with characteristics inherited from both parents. See Figure 2.3d.

(g) Engage in the cultural process, where cultural tags are picked up from other agents and passed on to other agents. This action is a one-way interaction where the neighbours do not reply synchronously.

(h) Engage in trading with neighbours where the initiating agent offers a given resource (sugar) in exchange for another resource (spice). The agent asks every neighbour and a trade will transact if it makes both agents better off. This action involves multiple synchronous interaction-steps within the same tick because of exchange of information and agreeing on the final transaction. See Figure 2.3d.

(i) Engage in lending and borrowing, where the agent offers loans to neighbours. This action also involves multiple synchronous interaction steps within the same tick because of exchange of information and agreeing on the final transaction.

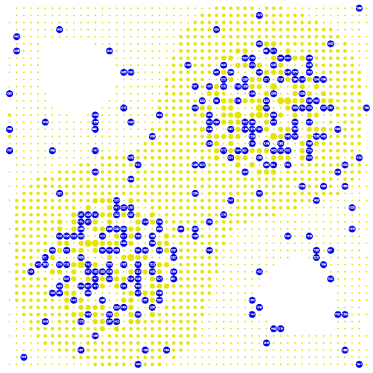
(j) Engage in disease processes, where the agent passes on diseases it has to other neighbour agents. This action is a one-way interaction where the neighbours do not reply synchronously.

2. Run the environment which consists of an NxN discrete grid

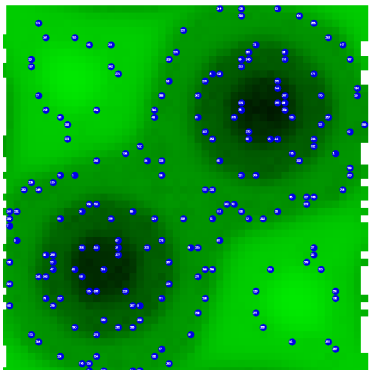
(a) Regrow resources on each site according to the model configuration: either with a given rate per tick as seen in Figure 2.3a, or immediately. Depending on whether seasons are enabled (see Figure 2.3c) the regrowing rate varies in different regions of the environment.

(b) Apply diffusion of pollution where the pollution generated by the agents spreads out slowly across the whole environment, see Figure 2.3b.

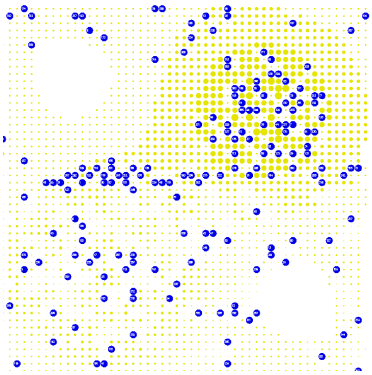
In Figure 2.3 visualisations of our Sugarscape implementation as discussed in Chapter 5.2 are shown.



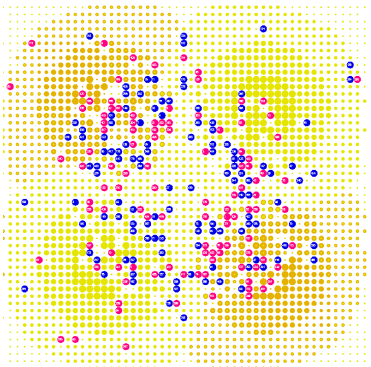
(a) *Animation II-2*: resource growback and infinite agent age.



(b) *Animation II-8*: active pollution with diffusion.



(c) *Animation II-7*: seasons, agents trying to migrate.



(d) *Figure IV-14*: trading, mating and finite life spans.

Figure 2.3: Visualisation of our Sugarscape implementation (see Chapter 5.2). The naming of the respective *Animation* and *Figure* is taken from [47].

2.3 Pure functional programming

To be able to understand the challenges of pure functional ABS and the solutions and concepts developed in this thesis, in this section we give a short introduction to functional programming, with an overview over its concepts and advanced features. As it is obviously beyond the focus of a thesis to give a full treatment of such a complex topic, we refer to additional literature and references for further discussions where appropriate.

Functional programming is called *functional* because it makes functions the main concept of programming, promoting them to first-class citizens. This means that functions can be assigned to variables, they can be passed as arguments to other functions and they can be constructed as return values from

functions. The roots of functional programming lie in the Lambda Calculus which was first described by Alonzo Church [28]. This is a fundamentally different approach to computing than imperative programming (including established object-orientation) which roots lie in the Turing Machine [155]. Rather than describing *how* something is computed as in the more operational approach of the Turing Machine, due to the more *declarative* nature of the Lambda Calculus, code in functional programming describes *what* is computed.

In [102] the author defines functional programming as a methodology attributing the following properties to it: programming without the assignment-operator; allowing for higher levels of abstraction; allowing to develop executable specifications and prototype implementations; connected to computer science theory; allowing to do algebraic reasoning. Further the author makes the subtle distinction between *applicative* and *functional* programming. Applicative programming can be understood as applying values to functions where one deals with pure expressions. In those expressions the value is independent from the evaluation order, also known as referential transparency. This means that such functions have no side effects and thus the outcome of their execution does not depend on the history or context of the system. Further, inputs and effects to an operation are obvious from the written form.

Note that applicative programming is not necessarily unique to the functional programming paradigm but can be emulated in an imperative language e.g. C as well. Functional programming is then defined by [102] as applicative programming with *higher-order* functions. These are functions which operate themselves on functions: they can take functions as arguments, construct new functions and return them as values. This is in stark contrast to first-order functions as used in applicative or imperative programming which just operate on data alone. Higher-order functions allow to capture frequently recurring patterns in functional programming in the same way like imperative languages captured patterns like `goto`, `while-do`, `if-then-else`, `for`. Common patterns in functional programming are (amongst others) the `map`, `fold`, `zip` functions. So functional programming is not really possible in this way in classic imperative languages like C as it is not possible to construct new functions and return them as results from functions. Object-oriented languages like Java provide mechanisms allowing to partially work around this limitation but are still far from *pure* functional programming.

The equivalence in functional programming to the *;* operator of imperative programming, which allows to compose imperative statements, is function composition. Function composition has no side effects as opposed to the imperative ; operator, which simply composes destructive assignment statements executed after another resulting in side effects. At the heart of modern functional programming is monadic programming which is polymorphic function composition: one can implement a user-defined function composition by allowing to run some code in-between function composition - this code of course depends on the type of the Monad one runs in. This allows to emulate all kind of effectful programming in an imperative style within a pure functional language (see Section 2.3.3 below). Although it might seem strange following an imperative style in a pure

functional language, some problems are inherently imperative in the way that computations need to be executed in a given sequence exhibiting some effects. Also, a pure functional language needs to have some way to deal with effects otherwise it would never be able to interact with the outside world and would be practically useless. The real benefit of monadic programming is that it is explicit about side effects and allows only effects which are fixed by the type of the Monad - the side effects which are possible are determined statically during compile time by the type system. Some general patterns can be extracted for example a `map`, `zip`, `fold` over Monads which results in effect-polymorphic behaviour.

2.3.1 Language of choice

In our research we are using the *pure* functional programming language Haskell. The paper of [76] gives a comprehensive overview over the history of the language, how it developed and its features and is very interesting to read and get accustomed to the background of the language. The reasons for choosing Haskell are:

- Rich feature-set - it has all fundamental concepts of the pure functional programming paradigm included, of which we explain the most important ones below. Further, Haskell has influenced a large number of languages, underlining its importance and influence in programming language design.
- Real-world applications - the strength of Haskell has been proven through a vast amount of highly diverse real-world applications [76, 77], is applicable to a number of real-world problems [118] and has a large number of libraries available [3].
- Modern - Haskell is constantly evolving through its community and adapting to keep up with the fast changing field of computer science. Further, the community is the main source of high-quality libraries.
- Highly advance type system - Haskell has a strong static type system, which catches all type errors already at compile time and does not allow to bypass the type system (unless *coerce* or other cheating functions like *unsafePerformIO* are used). Further, Haskell is a *pure* functional language and in our research it is absolutely paramount, that we focus on *pure* functional ABS, which avoids any *IO* type under all circumstances. This property is enabled by the advanced type system and its strong static nature.

A highly compelling example motivating the benefits of pure functional programming is the report [77], where in a prototyping contest of DARPA the Haskell prototype was by far the shortest with 85 lines of code (LoC) as compared to the C++ solution with 1105 LoC. The remarkable thing is that the

[3] `https://wiki.haskell.org/Applications_and_libraries`

Jury mistook the Haskell code as specification because its approach was to implement a small embedded domain specific language (EDSL) to solve the problem - this is a perfect proof how close an EDSL can get to a specification. When implementing an EDSL one develops and programs primitives e.g. types and functions in a host language (embed) in a way that they can be combined. The combination of these primitives then looks like a language specific to a given domain. The ease of development of EDSLs in pure functional programming is also a proof of the superior extensibility and composability of pure functional languages over object-orientation and is definitely one of its major strength. The classic paper [70] gives a wonderful way of constructing an EDSL to denotationally construct a picture reminiscent of the works of Escher. A major strength of developing an EDSL is that one can reason about and do formal verification. A nice introduction how to do reasoning in Haskell is given in [82].

For an excellent and widely used introduction to programming in Haskell we refer to [83]. Other, more exhaustive books on learning Haskell are [2, 98]. For an introduction to programming with the Lambda-Calculus we refer to [108]. For more general discussion of functional programming we refer to [76, 78, 102].

2.3.2 An example

Consider the factorial function in Haskell:

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

When looking at this function, the following can be identified:

1. Declarative - describe *what* the factorial function is rather than how to compute it. This is supported by *pattern matching* which allows to give multiple equations for the same function, matching on its input.

2. Immutable data - in functional programming there are no mutable variables, after a variable is assigned, it cannot change its contents. This also means that there is no destructive assignment operator which can re-assign values to a variable. To change values, recursion is employed.

3. Recursion - the function calls itself with a structurally smaller argument and will eventually reach the base case of 0. Recursion is the very meat of functional programming because it is the only way to implement loops in this paradigm due to immutable data.

4. Static types - the first line indicates the name and the type of the function. In this case the function takes one Integer as input and returns an Integer as output. Types are static in Haskell which means that there can be no type errors at run time, for example when one tries to implicitly cast one type into another because this is not supported by this kind of type system.

5. Explicit input and output - all data which are required and produced by the function have to be explicitly passed in and out of it. There exists no global mutable data whatsoever and data flow is always explicit.

6. Referential transparency - calling this function with the same argument will *always* lead to the same result, meaning one can replace this function by its value. This means that when implementing this function one can not read from a file or open a connection to a server. This is also known as *purity* and is indicated in Haskell in the types which means that it is also guaranteed by the compiler.

It may seem that one runs into efficiency problems in Haskell when using algorithms which are implemented in imperative languages through mutable data which allows in-place update of memory. The seminal work of [115] showed that when approaching this problem with a functional mind-set this does not necessarily be the case. The author presents functional data structures which are asymptotically as efficient as the best imperative implementations and discusses the estimation of the complexity of lazy programs.

2.3.3 Purity and side effects

One of the fundamental strengths of Haskell is its way of dealing with side effects in functions. A function with side effects has observable interactions with some state outside of its explicit scope. This means that its behaviour depends on the history of the system and that it loses its referential transparency character, which makes understanding and debugging much harder. Examples for side effects are (amongst others): modifying a variable, await an input from the keyboard, read or write to a file, open a connection to a server, drawing random numbers.

Obviously, to write real-world programs which interact with the outside world requires side effects. Haskell allows to indicate in the *type* of a function that it does or does *not* have side effects. Further, there are a broad range of different effect types available, to restrict the possible effects a function can have to only the required type. This is checked by the compiler, which means that code which tries to read from a file in a function which only allows drawing random numbers, will fail to compile. The most common side effect types are: `IO` allows all kind of input-output related side effects: reading and writing a file, creating threads, write to the standard output, read from the keyboard, opening network connections, mutable references; `Rand` allows drawing random numbers; `Reader` allows to read from an environment; `Writer` allows to write to a *monoid* environment; `State` allows to read and write shared state of a given type.

A function without any side effect type is called *pure*, and the `factorial` function discussed above is indeed pure. Below we give the `queryUser` function as an example of a function which is not pure. It constructs a computation which when executed, asks the user for its user name and compares it with a

given user configuration. In case the user name matches it returns `True`, and `False` otherwise after printing a corresponding message.

```
queryUser :: String -> IO Bool
queryUser username = do
  -- print text to console
  putStr "Type in user-name: "
  -- wait for user-input
  str <- getLine
  -- check if input matches user-name
  if str == username
    then do
      putStrLn "Welcome!"
      return True
    else do
      putStrLn "Wrong user-name!"
      return False
```

The `IO` in the first line indicates that the function runs in the `IO` effect and can thus (amongst others) print to the console and read input from it. What seems striking is that this looks very much like imperative code, which is no coincidence but intended. When we are dealing with side effects, ordering becomes important, thus Haskell introduced the so-called *do* notation which emulates an imperative style of programming. Whereas in imperative programming languages like C, commands are chained or composed together using the ; operator, in functional programming this is done using function composition: feeding the output of a function directly into the next function. The machinery behind the *do* notation does exactly this and desugars this imperative-style code into function compositions which run custom code between each line, depending on the type of effect the computation runs in. This approach of function composition with custom code in between each function allows to emulate a broad range of imperative-style effects, including the above mentioned ones. For a technical, in-depth discussion of the concept of side effects and how they are implemented in Haskell using Monads, we refer to the following papers: [91, 110, 159, 160, 161].

Although it might seem very restrictive at first, we get a number of benefits from making the type of effects we can use in the function explicit. First, we can restrict the side effects a function can have to a very specific type which is guaranteed at compile time. This means we can have much stronger guarantees about our program and the absence of potential errors already at compile time. Second, because running effects themselves is *pure*, we can execute effectful functions in a very controlled way by making the context of the effect explicit in the parameters to the effect execution. This allows a much easier approach to isolated testing because the history of the system is made explicit.

It is important to understand that the code fragments of effectful computations are in fact made up of enclosing lambda expressions, with the *do* notation being a syntactic sugared version. Thus functions which have an effect in their type can be seen as *pure* functions, which are referentially transparent and return such a fragment. This fragment, also often called *action*, results in an effect and a result when executed. We have to distinguish between the execution of

pure effects like `Rand`, `Read`, `Write`, `State` and the impure effect of `IO`. Pure effects are executed using special runner functions. They take an action together with initial values defining the history or context of the effect, for example an initial value for the `State` or the read-only value of the `Reader`, and run the action returning their result value. Thus, these pure effects can be executed in a referential transparent and completely controlled way. The impure `IO` effect works different though. There exists no dedicated `IO` execution function but it can only be executed from within the root `IO` action, which emanates from the `main :: IO ()` function of each Haskell program. Thus `IO` actions can only be run within an enclosing `IO` action, with the main `IO` action ultimately being executed by the Haskell runtime which is linked against the executable. The reason for that is that if we would have a way of executing IO actions within pure code we would lose all guarantees about referential transparency. There exists indeed the function `unsafePerformIO :: IO a → a`, which allows to execute an `IO` action within a pure function but its use is very limited and highly discouraged. Throughout this thesis and in all our code we have avoided the use of this function under all costs and it is not used anywhere, as avoiding `IO` is the very meaning of *purity* and *pure* functional programming.

2.3.3.1 Stacking effects using Monad Transformers

Often it is necessary to have multiple effects available, for example we want to manipulate a global state, write to some logging mechanism and need to be able to draw random numbers. The way this is achieved in Haskell is by using Monad Transformers [90], for which Haskell provides the two libraries *mtl* and *transformers*, which achieve the same things with slightly different philosophies. In our approach we primarily use *mtl* as it allows to overload functions with monadic type classes as explained below.

Although Monads share a common interface and properties, it is not possible to compose Monads in a general way as each Monad has different internals and semantics and thus it always depends on the Monad how to compose it into another arbitrary Monad [4]. Thus, the *mtl* library provides so called Transformer implementations of each of the standard Monads. A Transformer has an additional type parameter in its type constructor which has to be a Monad or another Transformer. This allows to stack multiple Monads or Transformers on top of each other. The stack is closed by using a non-Transformer Monad. Note that in *mtl* all non-Transformer Monads are actually Transformers with the `Identity` Monad as type parameter e.g. `State Int` is `StateT Int Identity`. Access to the various layers of the stack is achieved with the `lift :: Monad m ⇒ m a → t m a` function. Lets look at how we can define the type of a function which has multiple effects available:

```
data SimState = SimState { simStateAgents :: [SimAgent] ... }
```

[4] The technical details are quite involved and we don't go into them here but refer to the respective literature and tutorials on Monad Transformers.

```
simulationCore :: RandomGen g
               => Time
               -> StateT SimState (WriterT [String] (Rand g)) SimOut
simulationCore t = do
  -- get the agents from the simulation state
  -- encapsulated in StateT SimState
  as <- gets simStateAgents
  -- writing a logging output to the WriterT [String]
  -- here we need 1 lift
  lift (tell ["Next step " ++ show t])
  -- shuffle agents by running the MonadRandom action using the
  -- Rand Monad, need 2 lifts as it is the outermost monad
  asShuf <- lift $ lift $ randomShuffle as
  -- construct return value
  return (SimOut { ... })

randomShuffle :: MonadRandom m => [a] -> m [a]
```

The Monad stack consists of three effects: the `StateT` with `SimState` as its internal state as the innermost Monad. Note that although in the types it is the outermost, in terms of the Transformer stack it is the innermost, requiring no `lift` to access. `WriterT` with `[String]` as the logging facility is a parameter to the `StateT` Transformer, making it the second effect in the stack, thus requiring one `lift`. The stack is closed using the `Rand` Monad, which is the outermost effect, requiring two `lifts` to access it.

Note the use of `randomShuffle` and its type. It is an overloaded function, which has the `MonadRandom` type class in its type constraints. This indicates that it is a monadic action where `m` is of type `MonadRandom`, which supports the same functionality as `Rand`. This is the major benefit mtl provides, often resulting in much cleaner function types, not requiring to fix the order of the Monads in the stacks. Another benefit is that we do not need lifts any more; the drawback is that we cannot have multiple Monads of the same type which would be still possible in a fully qualified Monad stack. The benefits becomes particularly clear when more than one effect is required, for example we could have written the type of `simulationCore` as:

```
simulationCore :: (MonadState SimState m, MonadWriter [String] m, MonadRandom m)
               => Time -> m SimOut
```

A note on commutativity on Monad Transformers: because we are stacking effects on top of each other, subsequent effects can change the final outcome, depending on their position within the stack - this is called commutativity of Monads. All the Monads in the example above commute, which means it does not matter where they are positioned in the stack, the outcome will be the same. An exception to this is the `MaybeT` transformer, which introduces failure as an effect within a stack, thus when failure occurs, subsequent effects will not be applied any more, making `MaybeT` non-commutative.

2.3.4 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 this thesis *arrowized* FRP [79, 80] as implemented in the library Yampa [34, 75, 111] and Dunai [122] (see below) is used.

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, the system is sampled with.

$$\begin{aligned} Signal\,\alpha &\approx Time \to \alpha \\ SF\,\alpha\,\beta &\approx Signal\,\alpha \to Signal\,\beta \end{aligned}$$

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. Below, the relevant combinators and concepts used throughout the thesis are discuss briefly. For a more in-depth discussion we refer to [34, 75, 111].

Event An event in FRP is an occurrence at a specific point in time, which has no duration for example 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, (amongst others) 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. Note that the semantics of `switch` are that the signal function, into which is switched, is also executed at the time of switching.

Randomness In ABS, often there is the need to generate stochastic events, which occur based on a certain 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, following the exponential distribution. Note that at most one event will be generated and no backlog is kept. This means that when this function is not sampled with a sufficiently high frequency, depending on the rate, it will lose events.

Yampa also provides the combinator `noise :: (RandomGen g, Random b) ⇒ g → SF a b`, which generates a stream of noise by returning a random number in the default range for the type b, following the uniform distribution.

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 an SF for a given number of steps where in each step one provides the Δt and an input a. The function then returns the output of the signal function for each step. Note that the input is optional, indicated by `Maybe`. In the first step at $t = 0$, the initial `a` is applied and whenever the input is `Nothing` in subsequent steps, the last `a` which was not `Nothing` is re-used.

2.3.5 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 [79, 80].

In general, Arrows can be understood to be computations that represent processes, which take an input of a specific type, process it and output a value of a given type. The concept of processes, which signal functions are, maps naturally to Arrows which is the reason why Yampa is using them to represent their signal functions.

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 [119], 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 [123].

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

To create an Arrow, the `proc` keyword is used, which binds a variable after which the `do` of Patersons *do* notation [119] 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.

2.3.6 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 [121, 122]:

```
newtype MSF m a b = MSF {unMSF :: MSF m a b -> a -> m (b, MSF m a b)}
```

An MSF is also an Arrow, 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`. A part of the library Dunai is BearRiver, a wrapper, which re-implements Yampa on top of Dunai, which enables one to run arbitrary monadic computations in a signal function. BearRiver simply adds a type parameter `m` to each `SF`, which indicates the monadic context this signal function runs in.

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

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

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

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

```
    -- write to console
    arrM (liftIO . putStrLn) -< "hit floor with v " ++ show v ++
                                " after " ++ show s ++ " steps"
    returnA -< p
```

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

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

Dunai does not know about time in MSF, which is exactly what BearRiver builds on top. It does so by adding a `ReaderT Double`, which carries the Δt. This is the reason why we need one extra lift for accessing `StateT` and `RandT`. Thus `unMSF` returns a computation in the `ReaderT Double` Monad, which we need to peel away using `runReaderT`. This then results in a `StateT Int` computation, which we evaluate by using `runStateT` and the current number of steps as state. This then results in another monadic computation of `RandT` Monad, which we evaluate using `runRandT`. This finally returns an `IO` computation, which we simply evaluate to arrive at the final result.

2.4 Methodology

In this section we briefly motivate and justify our methods, to point out the scientific approach used in this thesis to address the aims and answer hypotheses put forward in Chapter 1. Fundamentally, the method we use is developing concepts step-by-step using the two well known agent-based models SIR, introduced in Chapter 2.2.1 and Sugarscape, introduced in Chapter 2.2.2. We put our approach into a broader context of how to implement ABS from a programming language agnostic view, discussed in Chapter 3, which serves as underlying assumptions and general direction to follow.

The first part of our method is dedicated to answer the question of how to implement ABS in a pure functional way, following a time-driven approach in Chapter 4 and an event-driven approach in Chapter 4. The reason for two techniques is that both are equally important in ABS and also that the concepts of event-driven ABS build on the ones developed in the preceding time-driven approach.

Generally, in both approaches, the aim is to develop a robust, maintainable and extensible implementation of the use-case models through which we develop concepts which can be adopted to ABS in general. The overall goal is a clear representation of agents with their local (immutable) state, a way for the agents

to interact with an (active) environment and one-directional and synchronous interactions between agents.

In the process of researching the pure functional event-driven approach to ABS, we also undertook a full and validated implementation of the Sugarscape model. This in itself together with the concepts developed, is already a sufficient proof that using a pure functional language to implement non-trivial ABS models is possible in a robust and maintainable way.

The second part of our method is dedicated to show the benefits of using the previously developed pure functional approach to ABS. It is split into two parts where in the first we investigate the hypothesis that pure functional programming makes it easy to apply parallel computation using parallelism and concurrency to ABS. The second part answers another central hypothesis namely that randomised property-based testing is a good match to test stochastic ABS implementations. In both parts we apply the concepts in questions directly to the implementations developed in the previous part and look at the resulting code, performance and implications to judge whether the outcome has the expected benefit or not as stated in the hypotheses.

Generally, all concepts we derive are driven by the hypotheses and aims from the Introduction and we continuously refer back to them, especially in the respective discussions and the final conclusion and discussion chapters. By doing this we are able to qualitatively assess whether the thesis has achieved the initial aims and answered the hypotheses in a satisfactory way.

Chapter 3

Implementing ABS

In this Chapter we briefly discuss general problems and considerations, ABS implementations need to solve independent from the programming paradigm. In general, an ABS implementation must solve the following fundamental problems:

1. How to represent an agent, its local state and its interface.
2. How to represent agent-to-agent interactions and enforcing their semantics.
3. How to represent an environment.
4. How to represent agent-to-environment interactions and enforcing their semantics.
5. How agents and an environment can initiate actions without external stimuli.
6. How to step the simulation.

We argue that the most fundamental concept of ABS is the *pro-activity* of both, agents and its environment. In computer systems, pro-activity, the ability to initiate actions on its own without external stimuli, is only possible when there is some internal stimulus, most naturally represented by a continuous increasing time-flow. Due to the discrete nature of computer systems, this time-flow must be discretized in steps as well and each step must be made available to the agent, acting as the internal stimulus. This allows the agent then to perceive time and become pro-active depending on time. So we can understand an ABS as a discrete time simulation where time is broken down into continuous, real-valued or discrete natural-valued time steps. Independent of the representation of the time-flow we have the two fundamental choices whether the time-flow is local to the agent or whether it is a system-global time-flow. Time-flows in computer systems can only be created through threads of execution where there

are two ways of feeding time-flow into an agent. Either it has its own thread-of-execution or the system creates the illusion of its own thread-of-execution by sharing the global thread sequentially among the agents where an agent has to yield the execution back after it has executed its step. Note the similarity to an operating system with cooperative multitasking in the latter case and real multiprocessing in the former.

Generally, there exist time- and event-driven approaches to ABS [107]. In time-driven ABS, time is explicitly modelled and is the main driver of the ABS dynamics. The semantics of models using this approach, center around time. As a representative example, which will be used in Chapter 4, we use the agent-based SIR model [100, 149]. Often such models are inspired by an underlying System Dynamics approach, where the continuous time-flow is the main driving force of the dynamics. It is clear that almost every ABS models time in some way, after all, modelling a virtual system over some (virtual) time is the very heart of Simulation. Still we want to distinguish clearly between different semantics of time representation in ABS: when time is seen as a continuous flow such as in the example of the agent-based SIR model, we talk about a truly time-driven approach. In other words: if an agent behaves as a time signal then we speak of a time-driven approach. This means that if the system is sampled with a $\Delta t = 0$ then, even though the agents are executed their behaviour must stay constant and must not change.

In the case where time advances in a discrete way either by means of events or messages, we talk about an event-driven approach. As a representative example, which will be used in Chapter 5 on event-driven ABS, we use an event-driven SIR and the Sugarscape model. In this model time is discrete and represented by the natural numbers where agents act in every tick. In such a model, the underlying semantics map more naturally to a DES core, extended by ABS features, as in the event-driven SIR and to a lesser extent in the Sugarscape model.

According to the definition of ABS in Chapter 2.2, an agent is a uniquely addressable entity with an identity, an internal state it has exclusive control over and can be interacted with by means of messages. In the established object-oriented approaches to ABS all this is implemented naturally by the use of objects: an object has a clear identity, encapsulates internal state and exposes an interface through public methods through which objects can interact with each other, also called messaging. The same applies to the environment and it is by no means clear how to achieve this in a pure functional approach where we don't have objects available. This is one of the central questions this thesis is trying to answer and it will be addressed in the subsequent Chapters 4 and 5.

Before we look into pure functional ABS implementation concepts in the next chapters, we need to discuss the concept of update strategies [150]. Generally, there are four strategies to approach time- and event-driven ABS, where the differences deal with how the simulation is stepped, the agents are executed and the interaction semantics work.

3.1 Sequential strategy

In this strategy there exists a globally synchronized time-flow and in each time step the simulation iterates through all the agents and updates one agent after another. Messages sent and changes to the environment made by agents are visible immediately, meaning that if an agent sends messages to other agents or changes the environment, agents which are executed after this agent will see these changes within the same time step. There is no source of randomness and non-determinism, rendering this strategy to be completely deterministic in each step. Messages can be processed either immediately or queued depending on the semantics of the model. If the model requires to process the messages immediately the model must be free of potential infinite-loops. Often in such models, the agents are shuffled when the model semantics require to average out the advantage of being executed as first. This strategy is of fundamental importance for event-driven ABS in Chapter 5. See Figure 3.1 for a visualisation of the control flow in this strategy.

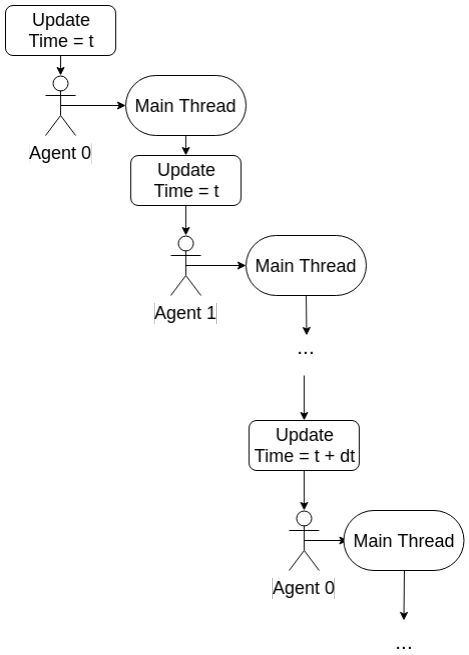


Figure 3.1: Control flow in the sequential strategy.

3.2 Parallel strategy

This strategy has a globally synchronized time-flow and in each time step iterates through all the agents and updates them in parallel. Messages sent and changes to the environment made by agents are visible in the next global step. We can

think about this strategy in a way that all agents make their moves at the same time. If one wants to change the environment in a way that it would be visible to other agents this is regarded as a semantic error in this strategy. First, it is not logical because all actions are meant to happen at the same time and second, it would implicitly induce an ordering, violating the semantics of the model, the *happens at the same time* idea. It does not make a difference if the agents are really executed in parallel or just sequentially - due to the isolation of information, this has the same effect. Also it will make no difference if we iterate over the agents sequentially or randomly, the outcome has to be the same: the strategy is event-ordering invariant as all events and updates happen *virtually at the same time*. This strategy is of fundamental importance for time-driven ABS in Chapter 4. See Figure 3.2 for a visualisation of the control flow in this strategy.

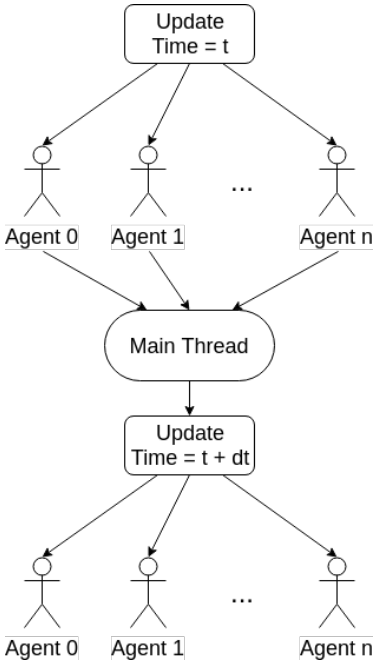


Figure 3.2: Control flow in the parallel strategy.

3.3 Concurrent strategy

This strategy has a globally synchronized time-flow but in each time step all the agents are updated in parallel with messages sent and changes to the environment are visible immediately. So this strategy can be understood as a more general form of the *parallel strategy*: all agents run at the same time but act concurrently. It is important to realize that when running agents, which are

able to see actions by others immediately, in parallel, we arrive at the very definition of concurrency: parallel execution with mutual read and write access to shared data. Of course this shared data access needs to be synchronized which in turn will introduce event orderings in the execution of the agents. At this point we have a source of inherent non-determinism: although when one ignores any hardware model of concurrency, at some point we need arbitration to decide which agent gets access to a shared resource first, arriving at non-deterministic solutions. This has the very important consequence that repeated runs with the same configuration of the agents and the model may lead to different results. This strategy is of fundamental importance for concurrent ABS in Chapter 7. See Figure 3.3 for a visualisation of the control flow in this strategy.

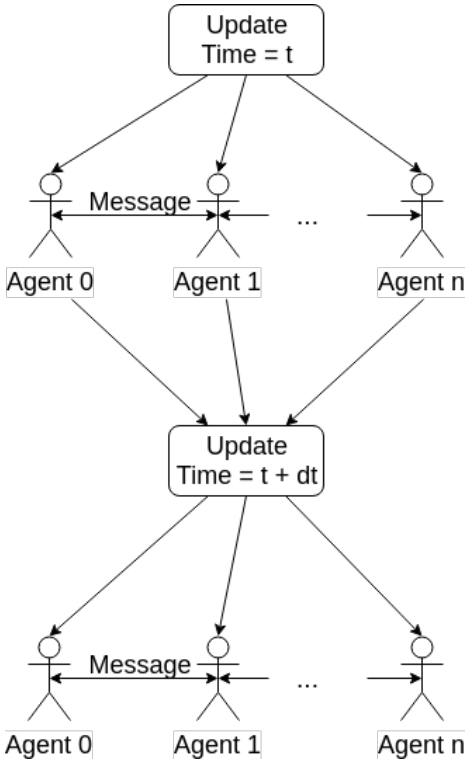


Figure 3.3: Control flow in the concurrent strategy.

3.4 Actor strategy

This strategy has no globally synchronized time-flow but all the agents run concurrently in parallel, with their own local time-flow. The messages and changes to the environment are visible as soon as the data arrive at the local agents - this can be immediately when running locally on a multiprocessor or

with a significant delay when running in a cluster over a network. Obviously this is also a non-deterministic strategy and repeated runs with the same agent and model configuration may (and will) lead to different results. It is of most importance to note that information and also time in this strategy is always local to an agent as each agent progresses in its own speed through the simulation. In this case one needs to explicitly *observe* an agent when one wants to extract information from it, for example for visualisation purposes. This observation is then only valid for this current point in time, local to the observer but not to the agent itself, which may have changed immediately after the observation. This implies that we need to sample our agents with observations when wanting to visualize them, which would inherently lead to well known sampling issues. A solution would be to invert the problem and create an observer agent which is known to all agents where each agent sends a *'I have changed'* message with the necessary information to the observer if it has changed its internal state. This also does not guarantee that the observations will really reflect the actual state the agent is in but is a remedy against the notorious sampling. The concept of Actors was proposed by [73] for which [63] and [31] developed semantics of different kinds. These works were very influential in the development of the concepts of agents and and can be regarded as foundational basics for ABS. We come back to this strategy in the context of concurrent ABS in Chapter 7. See Figure 3.4 for a visualisation of the control flow in this strategy.

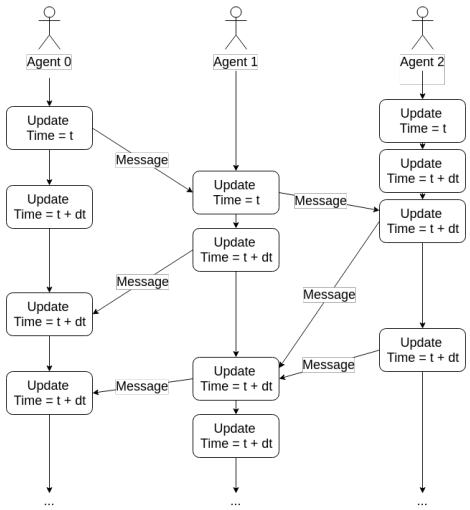


Figure 3.4: Control flow in the actor strategy.

3.5 Discussion

In the following chapters we discuss *how* to implement ABS from a pure functional perspective and *why* one would do so. More specifically, we show how

to approach the problems discussed in this chapter using pure functional programming. The *sequential* strategy will be covered in depth in Chapter 5 on event-driven ABS, the *parallel* one in Chapter 4 on time-driven ABS and the *concurrent* strategy is used in Chapter 7 on concurrent ABS. The *actor* strategy is not used in this thesis but its implementation follows directly from the Chapters 4 and 7: instead of globally synchronising in the main thread, a closed feedback loop is run in every agent thread.

As already outlined in Chapter 2.2, the established approaches implementing ABS use object-oriented programming and thus solve the problems outlined at the start of this chapter from this perspective, which is quite well understood by now, as high quality ABS frameworks like RePast [112] prove. In object-oriented programming an agent is mapped directly onto an object, encapsulating the agents state and providing methods, which implement the agents' actions. Object-orientation allows to expose a well-defined interface using public methods by which one can interact with the agent and query information from it. Agent objects can directly invoke other agents' methods, implicitly mutating the other agents' internal state, which makes direct agent interaction straight forward. Also with object-orientation, agents have global access to an environment for example through a Singleton [56] or a simple global variable, and can mutate the environments data by direct method calls.

All these language features are not available in functional programming and compared to object-orientation we face seemingly severe restrictions like immutable state, recursion and a static type system. Further, we restrict ourselves deliberately to *pure* functional programming and avoid running in the non-deterministic `IO` Monad under all costs. The question is then how to solve these problems in functional programming *and* use the restrictions to our advantage. In the next two chapters we show how to implement both a time-driven ABS using the agent-based SIR model as example (Chapter 4) and an event-driven ABS using also the SIR and the complex Sugarscape model as examples (Chapter 5). In both we present fundamental concepts of how to engineer an ABS from a pure functional programming perspective. This will then be used in subsequent chapters to discuss *why* one would follow a functional programming approach, identifying its benefits, advantages and also drawbacks over object-oriented approaches.

Part II:

Implementation techniques

Chapter 4

Pure functional time-driven ABS

In this chapter, we pose solutions to the problems outlined in the previous chapter, by deriving a pure functional approach for time-driven ABS using the example of the agent-based SIR model as introduced in Chapter 2.2.1. We start out with a first approach in Yampa and show its limitations. Then we generalise it to a more powerful approach, which utilises Monadic Stream Functions (MSF), a generalisation of FRP. Finally we add a structured environment, making the example more interesting and showing the real strength of ABS over other simulation methodologies like System Dynamics and Discrete Event Simulation [1].

4.1 First step: pure computation

As described in Chapter 2.3.4, Arrowized FRP [79] is a way to implement systems with continuous and discrete time-semantics, where the central concept is the signal function, which can be understood as a process over time, mapping an input- to an output-signal. Technically speaking, a signal function is a continuation which allows to capture state using closures and hides away the Δt. This has the effect, that the Δt is never exposed explicitly to the programmer, meaning the programmer can neither manipulate it nor define non-causal systems where a signal function depends on a signal in the future. Early, non-arrowized implementations of FRP had this flaw and it was only possible through arrowized FRP to solve this issue, as arrows parametrise also over the input-type, thus making it possible to deal with this issue.

As already pointed out, agents need to perceive time, which means that the concept of processes over time is an ideal match for our agents and our system as

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

a whole, thus we will implement them and the whole system as signal functions.

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 [18]. 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.

The *parallel* strategy matches the semantics of the agent-based SIR model due to the underlying roots in the System Dynamics approach. As discussed already in Chapter 3.2, in the parallel update strategy, the agents act conceptually all at the same time in lock-step. This implies that they observe the same system state during a time step and actions of an agent are only visible in the next time step - they are isolated from each other. As will become apparent, functional programming can be used to enforce the correct application of this strategy through the strong static type system, at compile time.

We start by defining the SIR states as Algebraic Data Type (ADT) and our agents as signal functions (SF), which receive the SIR states of all agents form the previous step as input and outputs the current SIR state of the agent. This definition, and the fact that Yampa is not monadic, guarantees already at compile, that the agents are isolated from each other, enforcing the *parallel* lock-step semantics of the model.

```
data SIRState = Susceptible | Infected | Recovered

type SIRAgent = SF [SIRState] SIRState

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

Depending on the initial state we return the corresponding behaviour. Note that we are passing a random-number generator instead of running in the *Rand* Monad because signal functions as implemented in Yampa are not capable of being monadic.

We see that the recovered agent ignores the random-number generator because a recovered agent does nothing, stays immune forever and can not get infected again in this model. Thus a recovered agent is a consuming state from which there is no escape, it simply acts as a sink, constantly returning *Recovered*:

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

Next, we implement the behaviour of a susceptible agent. It makes contact *on average* with β other random agents. For every *infected* agent it gets into

contact with, it becomes infected with a probability of γ. If an infection happens, it makes the transition to the *Infected* state. To make contact, it gets fed the states of all agents in the system from the previous time step, so it can draw random contacts. Observing the states of all agents from the previous step allows for a very simple, one-directional form of making contact between agents. Although simple, it works perfectly for this approach in particular and for time-driven ABS in general. We will discuss a more complex interaction mechanism between agents in the next Chapter 5 on event-driven ABS.

A susceptible agent behaves as susceptible until it becomes infected. Upon infection an *Event* is returned, which results in switching into the *infectedAgent* SF, which causes the agent to behave as an infected agent from that moment on. When an infection event occurs we change the behaviour of an agent using the Yampa combinator *switch*, which is quite elegant and expressive as it makes the change of behaviour at the occurrence of an event explicit. Note that to make contact *on average*, we use Yampas *occasionally* function which requires us to carefully select the right Δt for sampling the system as will be shown in results.

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

```
susceptibleAgent :: RandomGen g => g -> SIRAgent
susceptibleAgent g
    = switch
      -- delay switching by 1 step to prevent against transition
      -- from Susceptible to Recovered within one time step
      (susceptible g >>> iPre (Susceptible, NoEvent))
      (const (infectedAgent g))
  where
    susceptible :: RandomGen g => g -> SF [SIRState] (SIRState, Event ())
    susceptible g = proc as -> do
      -- generate make contact events with given rate
      makeContact <- occasionally g (1 / contactRate) () -< ()
      if isEvent makeContact
        then (do
          -- draw random element from the list
          a <- drawRandomElemSF g -< as
          case a of
            Infected -> do
              -- returns True with given probability
              i <- randomBoolSF g infectivity -< ()
              if i
                -- got infected, signal to switch through Event
                then returnA -< (Infected, Event ())
                else returnA -< (Susceptible, NoEvent)
            _        -> returnA -< (Susceptible, NoEvent))
        else returnA -< (Susceptible, NoEvent)
```

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

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

An infected agent recovers *on average* after δ time units. This is implemented by drawing the duration from an exponential distribution [18] with $\lambda = \frac{1}{\delta}$ and making the transition to the *Recovered* state after this duration. Thus the infected agent behaves as infected until it recovers after which it behaves as a recovered agent by switching into *recoveredAgent*. As in the case of the susceptible agent, we use the *occasionally* function to generate the event when the agent recovers. Note that the infected agent ignores the states of the other agents as its behaviour is completely independent of them.

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

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

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

What we need to implement next is a closed feedback oop - the heart of every agent-based simulation. The authors of [34, 111] discusses implementing this in Yampa. The function *stepSimulation* is an implementation of such a closed feedback loop. It takes the current signal functions and states of all agents, runs them all in parallel and returns this step's new agent states. Note the use

of *notYet*, which is required to delay switching by one step to break a potentially infinite recursive switching. This is necessary because we are recursively switching back into the *stepSimulation*, which would result in the immediate evaluation of the next step, overriding the output of the current step, recursively switching back into *stepSimulation* and so on. The combinator *notYet* breaks this by delaying the switching event by one step.

```
stepSimulation :: [SIRAgent] -> [SIRState] -> SF () [SIRState]
stepSimulation sfs as =
    dpSwitch
      -- feeding the agent states to each SF
      (\_ sfs' -> (map (\sf -> (as, sf)) sfs'))
      -- the signal functions
      sfs
      -- switching event, delay by one step to prevent
      -- infinite recursion
      (switchingEvt >>> notYet)
      -- recursively switch back into stepSimulation
      stepSimulation
  where
    switchingEvt :: SF ((), [SIRState]) (Event [SIRState])
    switchingEvt = arr (\ (_, newAs) -> Event newAs)
```

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

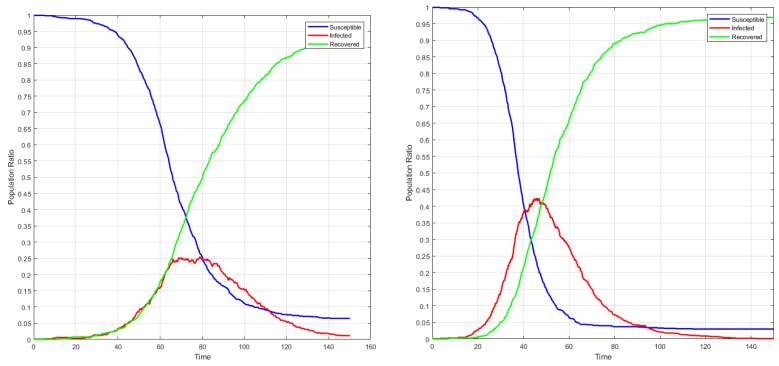
```
dpSwitch :: Functor col
         -- routing function
         => (forall sf. a -> col sf -> col (b, sf))
         -- SF collection
         -> col (SF b c)
         -- SF generating switching event
         -> SF (a, col c) (Event d)
         -- continuation to invoke upon event
         -> (col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

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

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.

4.1.1 Results

The dynamics generated by this implementation can be seen in Figure 4.1.



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

Figure 4.1: FRP simulation of agent-based SIR showing the influence of different Δt. Population size of 1,000 with contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time steps with respective Δt.

By following the FRP approach we assume a continuous flow of time, which means that we need to select a *correct* Δt, otherwise we would end up with wrong dynamics. The selection of a correct Δt depends in our case on *occasionally* in the susceptible behaviour, which randomly generates an event on average with *contact rate* following the exponential distribution. To arrive at the correct dynamics, this requires us to sample *occasionally*, and thus the whole system, with small enough Δt matching the frequency of events generated by *contact rate*. If we choose a too large Δt, we loose events, which will result in wrong dynamics as can be seen in Figure 4.1a. This issue is known as under-sampling and is described in Figure 4.2.

For tackling this issue we have three options. The first one is to use a smaller Δt as can be seen in 4.1b, which results in the whole system being sampled more often, thus reducing performance. The second option is to step the simulation with $\Delta t = 1$ and in each step, instead of using *occasionally*, to make a number of contacts drawn from the exponential distribution. Note that if we follow this option, we abandon the time-driven approach altogether because we don't abstract away from Δt and violate the fundamental abstraction of FRP, which assumes that time is continuous and signal functions are running conceptually infinitely fast and infinitely often [167]. This leaves us with the third option to implement super-sampling and apply it to *occasionally*, which allows us then to run the whole simulation with $\Delta t = 1.0$ and only sample the *occasionally* function with a much higher frequency.

The function *embed*, which allows to run a given signal function with provided Δt, does not help here because it does not return a signal function. What we need is a signal function which takes the number of super-samples n, the

(a) Under-sampling

(b) Super-sampling

Figure 4.2: A visual explanation of under-sampling and super-sampling. The black dots represent the time steps of the simulation. The red dots represent virtual events, which occur at specific points in continuous time. In the case of under-sampling, 3 events occur in between the two time steps but *occasionally* only captures the first one. By increasing the sampling frequency either through a smaller Δt or super-sampling all 3 events can be captured.

signal function *sf* to sample and returns a new signal function, which performs super-sampling on it. We provide a full implementation of such a function, which also gives an insight into how signal functions are implemented in Yampa, and how the Δt is hidden:

```
import FRP.Yampa.InternalCore

-- SF is the signal function defined for time t = 0 and returns
-- a continuation of type SF' which is the signal function
-- defined for t > 0: it receives an additional time-delta
-- data SF a b  = SF { sfTF :: a -> (SF' a b, b) }
-- data SF' a b = DTime -> a -> (SF' a b, b)

superSampling :: Int -> SF a b -> SF a [b]
superSampling n sf0 = SF { sfTF = tf0 }
  where
    -- no supersampling at time 0
    tf0 :: a -> (SF' a b, [b])
    tf0 a0 = (tfCont, [b0])
      where
        (sf', b0) = sfTF sf0 a0 -- running the SF
        tfCont    = superSamplingAux sf'

    superSamplingAux :: SF' a [b]
    superSamplingAux sf' = SF' tf
      where
        tf :: DTime -> a -> (SF' a b, [b])
        tf dt a = (tf', bs)
          where
            (sf'', bs) = superSampleRun n dt sf' a
            tf'        = superSamplingAux sf''

    superSampleRun :: Int -> DTime -> SF' a b -> a -> (SF' a b, [b])
    superSampleRun n dt sf a
        | n <= 1    = superSampleMulti 1 dt sf a []
        | otherwise = (sf', reverse bs)  -- reverse due to accumulator
      where
```

```
    superDt   = dt / fromIntegral n
    (sf', bs) = superSampleMulti n superDt sf a []

superSampleMulti :: Int -> DTime -> SF' a b -> a -> [b] -> (SF' a b, [b])
superSampleMulti 0 _ sf _ acc  = (sf, acc)
superSampleMulti n dt sf a acc = superSampleMulti (n-1) dt sf' a (b:acc)
  where
    (sf', b) = sfTF' sf dt a -- running the SF'
```

It evaluates the *SF* argument for n times, each with $\Delta t = \frac{\Delta t}{n}$ and the same input argument a for all n evaluations. At time 0 no super-sampling is performed and just a single output of the *SF* argument is calculated. A list of b is returned with length of n containing the result of the n evaluations of the *SF* argument. If 0 or less super samples are requested exactly one is calculated.

4.1.2 Discussion

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

- Time - the simulation occurs over virtual time which is modelled explicitly, divided into *fixed* Δt, where at each step all agents are executed.
- Agents - each agent is implemented as an individual, with the behaviour depending on its state. It is clear to see that agents behave as signals: when the system is sampled with $\Delta t = 0$ then their behaviour will stay constant and will not change because it is completely determined by the flow of time.
- Feedback - the output state of the agent in the current time step t is the input state for the next time step $t + \Delta t$.
- Environment - as environment we implicitly assume a fully-connected network (complete graph) where every agent 'knows' every other agent, including itself and thus can make contact with all of them.
- Stochasticity - it is an inherently stochastic simulation, which is indicated by the random-number generator and the usage of *occasionally*, *randomBoolSF* and *drawRandomElemSF*.
- Deterministic - repeated runs with the same initial random-number generator result in same dynamics. This may not come as a surprise but in Haskell we can guarantee that property statically already at compile time because our simulation runs *not* in the IO Monad. This guarantees that no external, uncontrollable sources of non-determinism can interfere with the simulation.
- Parallel, lock-step semantics - the simulation implements a *parallel* update strategy where in each step the agents are run isolated in parallel and don't see the actions of the others until the next step.

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

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

So far we have an acceptable implementation of an agent-based SIR approach. What we are lacking at the moment is a general treatment of an environment and an elegant solution to the random number correlation. In the next step we make the transition to Monadic Stream Functions (MSFs) as introduced in Dunai [122], which allows FRP within a monadic context and gives us a way for an elegant solution to the random number correlation.

4.2 Second step: going monadic

A part of the library Dunai is BearRiver, a wrapper re-implementing Yampa on top of Dunai, which allows us to easily replace Yampa with MSFs. This will enable us to run arbitrary monadic computations in a signal function, solving the problem of correlated random numbers through the use of the *Rand* Monad.

4.2.1 Identity Monad

We start by making the transition to BearRiver by simply replacing Yampas signal function by BearRivers', which is the same but takes an additional type parameter m, indicating the monadic context. If we replace this type parameter with the *Identity* Monad, we should be able to keep the code exactly the same, because BearRiver re-implements all necessary functions we are using from Yampa. We simply re-define the agent signal function, introducing the monad stack our SIR implementation runs in:

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

4.2.2 Rand 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 *Rand* Monad, which will allow us to run the whole simulation within the *Rand* Monad with the full features of FRP, finally solving the problem of correlated random numbers in an elegant way. We start by re-defining the SIRMonad and SIRAgent:

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

To access the *Rand* Monad functionality within the MSF context, overloaded functions are used. For the function *occasionally*, there exists a monadic pendant *occasionallyM* which requires a *MonadRandom* type class. Because we are now running within a *MonadRandom* instance we simply replace *occasionally* with *occasionallyM*.

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

4.2.3 Discussion

Running in the Random Monad elegantly solves the problem of correlated random numbers and guarantees that we will not have correlated stochastics as discussed in the previous section. In the next step we introduce the concept of an explicit discrete 2D environment.

4.3 Third step: adding an environment

So far we have implicitly assumed a fully connected network amongst agents, where each agent can see and knows every other agent. This is a valid environment and in accordance with the System Dynamics inspired implementation of the SIR model but does not show the real advantage of ABS to situate agents within arbitrary environments. Often, agents are situated within a discrete 2D environment [47] which is simply a finite NxM grid with either a Moore or von Neumann neighbourhood (Figure 4.3). Agents are either static or can move freely around with cells allowing either single or multiple occupants.

We can directly map the SIR model to a discrete 2D environment by placing the agents on a corresponding 2D grid with an unrestricted neighbourhood. The behaviour of the agents is the same but they select their interactions directly from the shared read-only environment, which will be passed to the agents as input. This allows agents to read the states of all their neighbours, which tells them if a neighbour is infected or not. To show the benefit over the System

(a) von Neumann (b) Moore

Figure 4.3: Common neighbourhoods in discrete 2D environments of Agent-Based Simulation.

Dynamics approach and for purposes of a more interesting approach, we restrict the neighbourhood to Moore (Figure 4.3b).

We also implemented this spatial approach in Java using the well known ABS library RePast [112], to have a comparison with a state of the art approach and came to the same results as shown in Figure 4.4. This supports, that our pure functional approach can produce such results as well and compares positively to the state of the art in the ABS field.

4.3.1 Implementation

We start by defining the discrete 2D environment for which we use an indexed two dimensional array. Each cell stores the agent state of the last time step, thus we use the *SIRState* as type for our array data. Also, we re-define the agent signal function to take the structured environment *SIREnv* as input instead of the list of all agents as in our previous approach. As output we keep the *SIRState*, which is the state the agent is currently in. Also we run in the *Rand* Monad as introduced before to avoid the random number correlation.

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

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

Note that the environment is not returned as output because the agents do not directly manipulate the environment but only read from it. Again, this enforces the semantics of the *parallel* update strategy through the types where the agents can only see the previous state of the environment and see the actions of other agents reflected in the environment only in the next step.

Note that we could have chosen to use a *StateT* transformer with the *SIREnv* as state, instead of passing it as input, with the agents then able to arbitrarily read/write, but this would have violated the semantics of our model because actions of agents would have become visible within the same time step.

The implementation of the susceptible, infected and recovered agents are almost the same with only the neighbour querying now slightly different.

Stepping the simulation needs a new approach because in each step we need to collect the agent outputs and update the environment for the next step. For

this we implemented a separate MSF, which receives the coordinates for every agent to be able to update the state in the environment after the agent was run. Note that we need use *mapM* to run the agents because we are running now in the context of the *Rand* Monad. This has the consequence that the agents are in fact run sequentially one after the other but because they cannot see the other agents actions nor observe changes in the shared read-only environment, it is *conceptually* a *parallel* update strategy where agents run in lock-step, isolated from each other at conceptually the same time.

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

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

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

4.3.2 Results

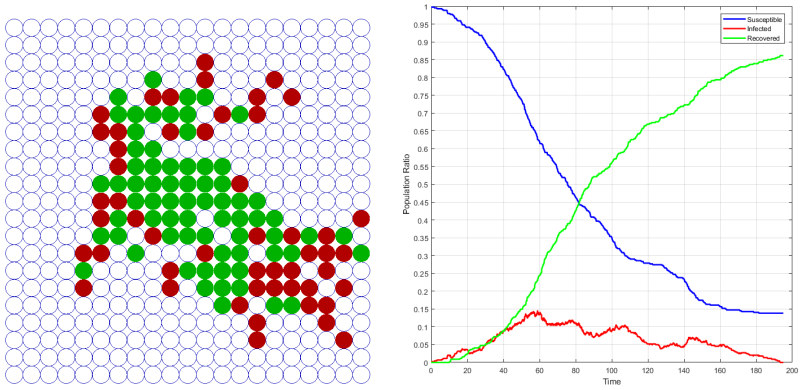
We implemented rendering of the environments using the gloss library which enabled us to cycle arbitrarily through the steps and inspect the spreading of the disease over time visually as seen in Figure 4.4.

Note that the dynamics of the spatial SIR simulation, which are seen in Figure 4.4b, look quite different from the reference dynamics of Figure 2.2. This is due to a much more restricted neighbourhood, resulting in far fewer infected agents at a time and a lower number of recovered agents at the end of the epidemic, meaning that fewer agents got infected overall.

4.3.3 Discussion

Introducing a structured environment with a Moore neighbourhood, shows the ability of ABS to place the heterogeneous agents in a generic environment, which is the fundamental advantage of an agent-based approach over other simulation methodologies and allows us to simulate much more realistic scenarios.

Note, that 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 environment and agent input and provide corresponding neighbourhood querying functions.



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

Figure 4.4: Simulating the agent-based SIR model on a 21x21 2D grid with Moore neighbourhood (Figure 4.3b), a single infected agent at the center and same SIR parameters as in Figure 2.2. Simulation run until $t = 200$ with fixed $\Delta t = 0.01$. Last infected agent recovers around $t = 194$. The susceptible agents are rendered as blue hollow circles for better contrast.

4.4 Discussion

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

4.4.1 Static guarantees

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.

Our approach can guarantee reproducibility already at compile time, which means that repeated runs of the simulation with the same initial conditions will always result in the same dynamics, something highly desirable in simulation in general. Although we allow side effects within agents, we restrict them to only the *Rand* Monad in a controlled, deterministic way and never use the *IO* Monad, which guarantees the absence of non-deterministic side effects within the agents and other parts of the simulation. This proves that this implementation is indeed *pure* computation. This can only be achieved through purity, which guarantees the absence of implicit side effects, which allows to rule out non-

deterministic influences at compile time through the strong static type system, something not possible with traditional object-oriented approaches.

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 [123]. Also by using FRP we gain all the benefits from it and can use research on testing, debugging and exploring FRP systems [120, 123].

Also we show how to implement the *parallel* update strategy [150] in a way that the correct semantics are enforced and guaranteed already at compile time through the types. This is not possible in traditional imperative implementations and poses another unique benefit over the use of functional programming in ABS.

The result of using FRP allows expressing continuous time-semantics in a very clear, compositional and declarative way, abstracting away the low-level details of time stepping and progress of time within an agent.

Using pure functional programming, we can enforce the correct semantics of agent execution through types where we demonstrate that this allows us to have both, sequential monadic behaviour, and agents acting *conceptually* at the same time in lock-step, something not possible using traditional object-oriented approaches.

4.4.2 Drawbacks

Despite the strengths and benefits we get by leveraging on FRP, there are errors that are not raised at compile time, e.g. we can still have infinite loops and run-time errors. This was for example investigated in [136] where the authors use dependent types to avoid some run-time errors in FRP. We suggest that one could go further and develop a domain specific type system for FRP that makes the FRP based ABS more predictable and that would support further mathematical analysis of its properties. Furthermore, moving to dependent types would pose a unique benefit over the traditional object-oriented approach and should allow us to express and guarantee even more properties at compile time. We leave this for further research.

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

We can conclude that the main difficulty of a pure functional approach evolves around the communication and interaction between agents, which is a direct consequence of the issue with agent identity. Agent interaction is straightforward in object-oriented programming, where it is achieved using method calls mutating the internal state of the agent, but that comes at the cost of a new class of bugs due to implicit data flow. In pure functional programming these data flows are explicit but our current approach of feeding back the states of

all agents as inputs is not very general. We address this problem in the next chapter.

4.4.3 Performance

Currently, the performance of this approach does not come close to imperative implementations. We compared the performance of the time-driven SIR as presented in Section 4.3 to an implementation in Java using the ABS library RePast [112]. We ran the simulation until $t = 100$ on a 51x51 (2,601 agents) with $\Delta t = 0.1$ (unknown in RePast) and averaged 8 runs. The performance results make the lack of speed of our approach quite clear: the pure functional approach needs around 72.5 seconds whereas the Java RePast version just 10.8 seconds on our machine to arrive at $t = 100$. It must be mentioned, that RePast does implement an event-driven approach to ABS, which can be much more performant [107] than a time-driven one, so the comparison is not completely valid.

As a remedy, we compared a time-driven SIR implementation we did in Java to the pure functional implementations of Chapter 4.1 without the *Rand* Monad and the environment. In the Java implementations we tried to follow conceptually similar approaches to the pure functional implementations but obviously that is not possible for every aspect. For example, we are not using any reactive programming library but we follow a similar time-sampling approach. We run for 150 time steps with 1,000 susceptible and 1 infected agent, $\beta = 5$, $\gamma = 0.05$, $\delta = 15$ and $\delta t = 0.01$. Further, we fixed the random-number generators to guarantee identical dynamics in every run and averaged 8 runs. The time-driven Java implementation averages at a performance of 0.5 seconds, compared to 27.6 seconds in Haskell. The implementation of 4.3 with the same configuration using a 10x10 environment with no neighbourhood restrictions averages at 19 seconds.

We expect a substantial performance improvement when switching to an event-driven approach [107] in the next Chapter 5.1. Further, the performance issue will be addressed more in-depth in the chapters on parallelism (Chapter 6) and concurrency (Chapter 7).

Chapter 5

Pure functional event-driven ABS

> The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.
>
> Dijkstra, EWD340

In this chapter we build on the previous discussion of update strategies in Chapter 3 and the implementation techniques presented in the time-driven approach of Chapter 4 to develop concepts for event-driven ABS in a pure functional way.

In event-driven ABS [107], the simulation is advanced through events: agents and the environment schedule events into the future and react to incoming events scheduled by themselves, other agents, the environment or the simulation kernel. Time is discrete in this approach: it advances step-wise from event to event, where each event has an associated receiver and Δt, indicating the delay to the current virtual simulation time when should be scheduled. This implies that time could stay constant, for example when an event is scheduled with $\Delta t = 0$ the virtual simulation time does not advance. Further, agents can schedule events to themselves, emulating a recurring behaviour, which in turn emulates pro-active behaviour. Because agents can adopt and change their state and behaviour when processing an event, this means that even if time does not advance, agents can change. This non-signal behaviour is the fundamental difference to the time-driven approach in Chapter 4. Further, this mechanism is used to implement synchronous agent interactions pure functionally as discussed in the respective sections below.

The event-driven approach makes the simulation kernel technically closely related to a Discrete Event Simulation (DES) [169]. Due to the necessity of imposing a correct ordering of events in this type of ABS, it needs to be stepped event by event, with the *sequential* update strategy, as introduced in Chapter

3.1. It is important to emphasise that only the semantics of the sequential update strategy allow the kind of features presented in the following sections, as the agents act one after the other, seeing the effects of previous agents in the same time step. This would not make sense in the parallel update strategy as used in time-driven ABS, where agents act conceptually at the same time - event-driven ABS is inherently sequential due to its fundamental reliance on effects as will become clearer in the sections below. Note that there exists also Parallel DES (PDES) [53], which processes events in parallel and deals with inconsistencies by reverting to consistent states. We hypothesise that a pure functional approach could be beneficial in such an approach due to persistent data structures and explicit handling of side effects but we leave this for further research.

We start introducing the concepts of agent identity and event scheduling utilising both *Reader* and *Writer* Monads. We do this by using an event-driven agent-based SIR model, inspired by [100]. We then use the highly complex Sugarscape model as introduced in Chapter 2.2.2, to develop more advanced features of ABS in a pure functional context: dynamic creation and removal of agents during simulation, adding a shared mutable environment, local mutable agent state and synchronous agent interactions.

5.1 Basics of event-driven ABS

In this section we derive the basics of event-driven ABS using the SIR model, as introduced in Chapter 2.2.1, with an event-driven approach inspired by [100]. Although it is a fundamentally different approach to ABS than the time-driven implementation in Chapter 4.1 both solutions are quantitatively equal as they produce the same class of dynamics. Qualitatively they fundamentally differ though in terms of expressivity and performance as we will see in the discussion.

The basics of event-driven ABS are the concept of agent identity, events and event-scheduling. We introduce them step-by-step using various Monads and then generalise to a *tagless final* approach, which has various benefits as pointed out in the respective section.

5.1.1 An event-driven SIR

Before we can derive implementation concepts, we first need to discuss how an event-driven SIR model works, as inspired by [100]. Fundamentally, what is required is to transform all time-dependent behaviour and agent interactions into the scheduling and receiving of events. For the SIR this should be trivial and straightforward, taking inspiration from the time-driven implementation, where we simply translate the occurrences of events generated by *occasionally* into scheduling of events. For agent interactions we also use events, making this more explicit than in the time-driven approach. As already pointed out, assuming that events have a receiver and a scheduling time given as Δt relative to the current simulation time, we end up with three event-types:

1. **MakeContact** - is used to let susceptible agents pro-actively make contact with β (contact rate) other agents per 1 time unit.
2. **Contact**$_{Sender,\ SIRState}$ - is used to make contact between agents where, agents reveal their state by sending or replying their current state.
3. **Recover** - is used to let infected agents recover pro-actively after the given δ (illness duration).

Now we can give a concise definition of all three agent behaviours:

Susceptible Agent

- A susceptible agent initially schedules a *MakeContact* event with $\Delta t = 1$ to itself.
- When receiving *MakeContact*, the agent sends a *Contact* event to β (contact rate) random other agents with $\Delta t = 0$ and *SIRState* of *Susceptible*, resulting in these events to be scheduled immediately. Further, the agent schedules *MakeContact* with $\Delta t = 1$ to itself, to keep the pro-active process of making contact with other agents active.
- When the agent receives a *Contact* event, it checks if it is from an infected agent (*SIRState* is *Infected*). If the event is not from an infected agent, it ignores it, otherwise it becomes infected with a given probability.

Infected Agent

- An infected agent initially schedules a *Recover* event to itself, with an exponentially distributed random Δt of δ (illness duration).
- When the agent receives a *Contact* event, it checks if it is from a susceptible agent (*SIRState* is *Susceptible*). If the event is not from a susceptible agent, it ignores it, otherwise it simply replies to this susceptible agent with a *Contact* event with $\Delta t = 0$ and and *SIRState* of *Infected*.

Recovered Agent The recovered agent does not change any more, reacts to no incoming events and schedules no events - it stays constantly *Recovered* forever.

It is easy to see that this behaviour emulates the time-driven one and indeed in Figure 5.1 it is also visually clear that it produces similar dynamics. A striking difference are the small spikes and steps in the dynamics, which stem from the fact that time advances discretely and not continuous as in the time-driven implementation. In Chapter 9, we use property-based testing to show that both implementations indeed produce similar distributions in their dynamics, thus putting the claim that both implementations are quantitatively equal on a much more robust ground.

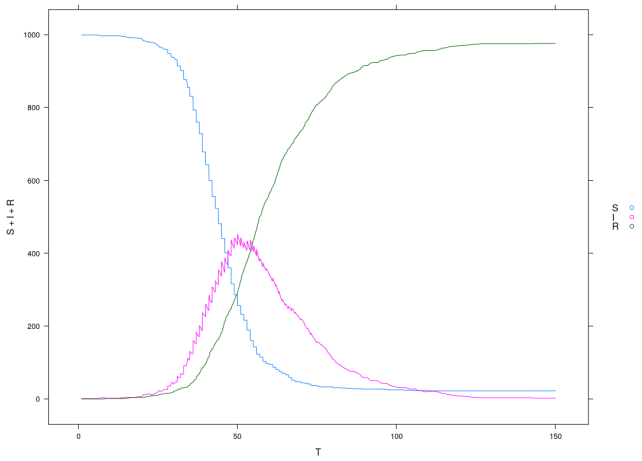


Figure 5.1: Dynamics of the event-driven SIR model. Population Size $N = 1{,}000$, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time steps.

5.1.2 Events, agent identity and scheduling

We can now start to discuss the concepts from an implementation perspective. First, we need to make the concept of an event explicit: they are of a given type, have a receiver and a timestamp in *absolute* simulation time when they shall be scheduled. We keep the event-type polymorphic and represent the receiver by an *AgentId* which is a simple *Int*. For efficient scheduling, the events are kept in a priority-queue [1], sorted ascending by the timestamp. Thus we define the following:

```
type Time         = Double
type AgentId      = Int
data QueueItem e = QueueItem e AgentId Time

-- the event priority-queue
type EventQueue e = PQ.MinQueue (QueueItem e)

-- implement Ord for QueueItem for acended sorting
instance Ord (QueueItem e) where
  compare (QueueItem _ _ t1) (QueueItem _ _ t2) = compare t1 t2
```

Next, we define a polymorphic type for the agent. In event-driven ABS, due to the fact that agents are not signals any more, we abandon time-aware signal functions of BearRiver from the previous chapter and focus solely on Monadic Stream Functions (MSF). In event-driven ABS agents receive events, thus as input to an *MSF* the polymorphic event type e is used. As output, the polymorphic output type o is used, which will be instantiated to a specific

[1] We are using the *Data.PQueue.Min* implementation from the *pqueue* package.

monomorphic type in the SIR model below. The question is now what Monad shall be used. For scheduling purposes (and because models might require it), agents should be able to *read* the current simulation time: this is accomplished through a *ReaderT Time*. Further, agents should be able to *read* the identities of the other agents available in the simulation so they can schedule events to them when necessary: this is accomplished through a *ReaderT [AgentId]*. Most importantly, agents have to be able to schedule events, meaning they have to be able to *write* the events into some sink where they are accumulated for scheduling: this is accomplished through a *WriterT [QueueItem e]*. Finally, the transformer stack needs to be extendible by other Monads, specified in concrete models like the SIR below, so we add another polymorphic type *m*, indicating the closing Monad (stack).

```
type ABSMonad m e   = ReaderT Time (WriterT [QueueItem e] (ReaderT [AgentId] m))
type AgentMSF m e o = MSF (ABSMonad m e) e o
```

Note that BearRivers *SF* has also a *ReaderT Double* as the innermost Monad but we deliberately avoided its use because the intended semantics of an *SF* are different: the value in the *ReaderT* of the *SF* represents the sampling time-delta and not the absolute time, as in the event-driven case.

We can already implement a few polymorphic functions, operating on the given Monad stack. First, we implement a function *allAgentIds* which simply returns the *AgentId* of all agents, the contents of the *ReaderT [AgentId]*. Second, we implement a function *scheduleEvent* which allows to schedule a given event to a given receiver into the future given a specific time-delay, relative to the current simulation time.

```
allAgentIds :: Monad m => (ABSMonad m e) [AgentId]
allAgentIds = lift (lift ask)

scheduleEvent :: Monad m
              => e         -- ^ event
              -> AgentId   -- ^ receiver
              -> Double    -- ^ time-delay
              -> (ABSMonad m e) ()
scheduleEvent e aid dt = do
  -- get current simulation time
  t <- ask
  -- construct queue item
  let q = QueueItem e aid (t + dt)
  -- write/append (tell) to the WriterT (QueueItem e)
  lift (tell [q])
```

Processing events can also be implemented generically and is straightforward, thus we only discuss the subtleties. For efficient lookup of event receivers all agents are organised into an *IntMap*, which also holds the current output of the agent, to allow sampling of the domain-state. In general, the domain-state is highly model specific, thus a generic implementation needs to offer some mechanism to update the domain-state after an event, a process we named domain-state sampling. Our approach is to call a function which receives the

agent map and returns a new domain-state for the current event/time step. These domain-states are appended to an infinite list which forms the output of the simulation.

The events are then processed in the order provided by the queue and each event is executed with the given receiver. Running a receiver is simply achieved using the agent map, where a reviver is looked up and its *MSF* is evaluated with the given event as input and the resulting monadic actions executed with the given information.

5.1.3 Parametrising for SIR

With the generic concepts now established, we show how to parametrise them to the concrete SIR model. First, we define the already well known states the agents can be in and the three different event types, as already introduced above.

```
data SIRState = Susceptible | Infected | Recovered
data SIREvent = MakeContact | Contact AgentId SIRState | Recover
```

Next, we parametrise the *ABSMonad* to the SIR model: because behaviour is stochastic, we need to make use of the *Rand* Monad, which also closes the Monad stack of *ABSMonad*. Further, the event type is obviously parametrised to *SIREvent*.

```
type SIRMonad g = ABSMonad (Rand g) SIREvent
```

Now we define a *SIRAgent* which can be understood as a constructing function, run once upon construction of the agent. This constructing functions runs in the *SIRMonad*, thus agents can already make full use of the functionality, so they can schedule initial events, depending on their initial state. This is important for the susceptible and infected agent, which both need to schedule initial events for pro-active behaviour. The constructing functions also takes the *AgentId* of the agent, thus making it available to the agent at construction time. It returns the initial agent-behaviour as *AgentMSF*.

```
type SIRAgent g
       = AgentId -> (SIRMonad g) (AgentMSF (SIRMonad g) SIREvent SIRState)
```

The implementation of the constructing function of type *SIRAgent* is straightforward and follows the specification given above. It makes use of the functions *scheduleMakeContact* and *scheduleRecovery* which are implemented using the generic *scheduleEvent* from above.

```
sirAgent :: RandomGen g
         => Int          -- ^ contact rate (beta)
         -> Double       -- ^ infectivity (gamma)
         -> Double       -- ^ illness duration (delta)
         -> SIRState     -- ^ initial state of the agent
         -> SIRAgent g
sirAgent beta gamma delta Susceptible aid = do
  -- on start, schedule MakeContact to itself
```

```
  scheduleMakeContact aid 1
  -- return susceptible behaviour
  return (susceptibleAgent aid beta gamma delta)
sirAgent _ _ delta Infected aid = do
  -- on start, schedule Recover to itself
  scheduleRecovery aid delta
  -- return infected behaviour
  return (infectedAgent aid)
sirAgent _ _ _ Recovered _ =
  -- simply return recovered behaviour
  return recoveredAgent

scheduleMakeContact :: RandomGen g => AgentId -> Double -> (SIRMonadT g) ()
scheduleMakeContact aid = scheduleEvent MakeContact aid

scheduleRecovery :: RandomGen g => AgentId -> Double -> (SIRMonadT g) ()
scheduleRecovery aid delta = do
  dt <- (lift . lift . lift) (randomExpM (1 / delta))
  scheduleEvent Recover aid dt

-- returns random value following exponential distribution with given lambda
randomExpM :: MonadRandom m => Double -> m Double
```

Now we are finally ready to implement the actual behaviour of an agent, where we discuss the full implementation of the susceptible agent behaviour. The basic structure should be already familiar from the time-driven approach, using *switch* to dynamically change the behaviour to *infectedAgent* in case of an infection. The behaviour is then a simple event handler, pattern matching on the incoming events:

```
susceptibleAgent :: RandomGen g
                 => AgentId        -- ^ agents id
                 -> Int            -- ^ contact rate (beta)
                 -> Double         -- ^ infectivity (gamma)
                 -> Double         -- ^ illness duration (delta)
                 -> SIRAgentMSF g
susceptibleAgent aid beta gamma delta =
    switch susceptibleAgentInfected (const (infectedAgent aid))
  where
    susceptibleAgentInfected :: RandomGen g
                             => MSF (SIRMonadT g) SIREvent (SIRState, Maybe ())
    susceptibleAgentInfected = proc e -> do
      -- handle incoming event in monadic action
      ret <- arrM handleEvent -< e
      case ret of
        Nothing -> returnA -< (Susceptible, ret)
        _       -> returnA -< (Infected, ret)
```

We strictly follow the specification as above. In case the agent receives *Contact* from an infected agent it might become infected with a given probability. If it becomes infected, it schedules the recovery as it will make the transition to an infected agent.

```
-- received Contact from an Infected agent
handleEvent :: RandomGen g => SIREvent -> (SIRMonadT g) (Maybe ())
```

```
handleEvent (Contact _ Infected) = do
  -- become infected with gamma probability
  r <- (lift . lift . lift) (randomBoolM gamma)
  if r
    -- got infected
    then do
      -- schedule Recovery to self, because switching to infected
      scheduleRecovery aid delta
      return (Just ())
    -- no infection
    else return Nothing

-- returns True with given probability
randomBoolM :: MonadRandom m => Double -> m Bool
```

In case the agent receivers *MakeContact* from itself, it will send *Contact* with *Susceptible* to β (contact rate) other agents without delay and *MakeContact* to itself with a delay of 1 time unit.

```
-- received MakeContact (from itself)
handleEvent MakeContact = do
  ais <- allAgentIds
  -- get beta random agents
  receivers <- (lift . lift . lift) (forM [1..beta] (const (randomElemM ais)))
  -- make contact with random agents
  mapM_ makeContactWith receivers
  -- re-schedule MakeContact to self
  scheduleMakeContact aid 1
  return Nothing

makeContactWith :: AgentId -> (SIRMonadT g) ()
makeContactWith receiver =
  -- schedule Contact event immediately
  scheduleEvent (Contact aid Susceptible) receiver 0

-- picks an element randomly from the (non empty) list
randomElemM :: MonadRandom m => [e] -> m e
```

The infected and recovered behaviours are conceptually equivalent and thus left as a trivial exercise to the reader.

5.1.4 Tagless Final

At this point, the basics of event-driven ABS should be clear: how events are represented and processed using an event queue, how agents are represented with an *MSF* and the idea behind the underlying polymorphic Monad transformer stack. Further, by parametrising the polymorphic concepts to the SIR model, we showed how to instantiate the generic concepts into a concrete model to arrive at a robust, maintainable and solid solution which is very likely to be correct up to the initial informal specification.

In this section we briefly want to show how the so-called *tagless final* approach [95] can be used to arrive at a cleaner and more extensible interface of our implementation, which is also open to different *interpretations*. The idea

behind *tagless final* is simple: specify the interface of operations in a type class and then write one or multiple interpreters for it, which simply means writing an instance implementation for the given type class. We start by defining the type class *MonadAgent* with all the necessary methods, making up the effectful API of our agents. Note that we need to enable two language extensions: *MultiParamTypeClasses* because we want to have more than a single type parameter in the type class - besides the Monad m, we also want to parametrise over the event type e; *FunctionalDependencies* because the event type e is determined by the Monad type m.

```
{-# LANGUAGE MultiParamTypeClasses  #-}
{-# LANGUAGE FunctionalDependencies #-}

class Monad m => MonadAgent e m | m -> e where
  randomBool  :: Double -> m Bool
  randomExp   :: Double -> m Double
  randomElem  :: [a] -> m a
  getAgentIds :: m [AgentId]
  getTime     :: m Time
  getMyId     :: m AgentId
  schedEvent  :: e -> AgentId -> Double -> m ()
```

The methods are self explaining. This type class is now used to replace the Monad stack by an overloaded type definition in the respective functions. Thus, the implementation of the agent constructing function and the agent behaviours are the same, with only the types changing slightly, lifts becoming obsolete and calls to function replaced by calls to methods of the type class. We don't give the full implementation again but only the type of the agent construction function as example, the types of the agent behaviours follow a similar pattern:

```
sirAgent :: MonadAgent SIREvent m  -- CHANGED: overloaded with type class
         => Int          -- ^ contact rate (beta)
         -> Double       -- ^ infectivity (gamma)
         -> Double       -- ^ illness duration (delta)
         -> SIRState     -- ^ initial state of the agent
         -> m (MSF m SIREvent SIRState) -- CHANGED: no Monad stack
```

Note that we added a *getMyId* method, which shall return the *AgentId* of the agent itself, avoiding the need for the agent of keeping the agent id around and also making it possible to implement more robust self-scheduling functions. For example, the *scheduleRecovery* function is implemented in the *tagless final* approach in the following way:

```
scheduleRecovery :: MonadAgent SIREvent m => Double -> m ()
scheduleRecovery delta = do
  -- draw random value from exponential distribution
  dt <- randomExp (1 / delta)
  -- get id of agent, no more need to pass it explicitly
  ai <- getMyId
  -- schedule Recover to itself
  schedEvent Recover ai dt
```

What we are missing is a *pure* interpreter for the agent implementation and the *MonadAgent* type class. We start by defining a *newtype*, which basically is a conceptually similar Monad stack as in the original implementation without the *tagless final* approach. We let Haskell automatically derive monadic type classes, Functor, Applicative and Monad instances which saves a lot of boiler plate code, for which the *GeneralizedNewtypeDeriving* language extension is required. Instead of the *Rand* Monad, a *StateT SimState* is used, which carries the random-number generator and other data for synchronous agent interactions as will be discussed in the respective sections.

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

newtype SIRAgentPure a = SIRAgentPure
  { unSirAgentPure :: ReaderT (Time, AgentId, [AgentId]) -- combined into one
                        (WriterT [QueueItem SIREvent]
                          (State SimState)) a}
  deriving (Functor, Applicative, Monad,
            MonadReader (Time, AgentId, [AgentId]),
            MonadWriter [QueueItem SIREvent],
            MonadState SimState)
```

Having this *newtype* we can now write a *pure* interpreter for the *MonadAgent*. The implementations are straightforward and should be self explanatory. To run *Rand* Monad actions, the function *runRandWithSimState* is used, which extracts the random-number generator from *SimState*, runs the action and puts the changed random-number generator back into the *SimState*.

```
{-# LANGUAGE FlexibleContexts           #-}
{-# LANGUAGE MultiParamTypeClasses      #-}

instance MonadAgent SIREvent SIRAgentPure where
  randomBool = runRandWithSimState . randomBoolM
  randomElem = runRandWithSimState . randomElemM
  randomExp  = runRandWithSimState . randomExpM
  -- schedEvent :: SIREvent -> AgentId -> Double -> m ()
  schedEvent e receiver dt = do
    t <- getTime
    tell [QueueItem e receiver (t + dt)]
  -- getAgentIds :: m [AgentId]
  getAgentIds = asks trd
  -- getTime :: m Time
  getTime = asks fst3
  -- getMyId :: m AgentId
  getMyId = asks snd3

fst3 :: (a,b,c) -> a
snd3 :: (a,b,c) -> b
trd :: (a,b,c) -> c
runRandWithSimState :: MonadState SimState m => Rand StdGen a -> m a
```

The main benefit of a *tagless final* approach is that it is a solution to the expression problem [95]: it is possible to add new interpreters of an embedded language and add new functionality without breaking the existing implementations. Interpretation in our case means that we can use different underlying

Monads to run the agents: if we want to guarantee purity, no *IO* Monad shall be used. Otherwise when concurrency with a lock-based approach or a lock-free approach is required *IO* or *STM* can be used in the underlying interpreter. Also, for reproducible unit testing, one can write custom test-interpreters where methods always return a-priori known results, similar to mocking. Adding new functionality is less an issue here but might become highly important when designing a more general ABS library, building on the *tagless final* approach. It would allow the user of such a library to extend existing agents or default behaviour with new, custom-built methods, without breaking the existing ones. We leave that for further research.

5.2 Advanced features of event-driven ABS

In the previous section we have established the basics of event-driven ABS. It is now clear how events are represented, agent identity is handled, agents receive and schedule events and how they are scheduled and domain-state sampled. Further, by using the *tagless final* approach, we arrived at an elegant, extensible and robust solution, which separates specification - agent and its behaviour - from implementation - the *pure* interpreter.

In this section we present more advanced concepts of event-driven ABS, necessary in models with much higher complexity than the simple SIR. We developed these concepts using the Sugarscape model as introduced in Chapter 2.2.2, thus will discuss them from this models perspective. More specifically we show how to create and remove agents dynamically during simulation, add a shared mutable environment, model local mutable agent state and finally how synchronous agent interactions can be implemented. Together with the basics of event-driven ABS, with these concepts established it should be possible to implement a wide range of event-driven ABS models.

5.2.1 An event-driven Sugarscape

The event-driven approach of Sugarscape alters slightly from the event-driven SIR, discussed before. In the SIR, the dynamics are driven by the pro-activity of the agents through the *MakeContact* and *Recover* events, which the agents (re-)schedule to themselves and thus drive time and the dynamics forward as a group without central authority. In Sugarscape, the semantics of the model are different, where in each time step all agents are executed in random order where they perform their actions and interact with each other. Time is advanced discretely, in natural numbers, centrally through the simulation kernel, by scheduling a *Tick* event to each of the agents in random order. Thus, events have no associated timestamp as there is no need for scheduling of events into the future. Indeed, beside the simulation kernel-specific *Tick* event, the model-specific events in Sugarscape are used solely for the purpose of agent interactions as will be discussed below. This follows the same approach of the event-driven SIR, where agent interactions between susceptible and infected are

implemented by scheduling events with a time-delay of 0. The Sugarscape implementation follows the same idea but does that without the use of time-delays: model specific events are scheduled immediately within the same *Tick* event.

The polymorphic event definition in Sugarscape is thus split into two parts: *Tick*, which is scheduled by the simulation kernel and indicates to the agent the start of a new time step; *DomainEvent*, which is scheduled by other agents to a specific receiver within a given *Tick* and received by the target agent within the same *Tick*. The *Tick* event carries the time-delta between steps to avoid the necessity of hardcoding it into the agent; *DomainEvent* also carries the sender of the event, to support easy replying to events which avoids the need to add the sender to the actual event type as was done in the event-driven SIR. Due to the discrete time semantics of Sugarscape, where time is advanced in natural numbered steps, time and time-delta between steps are represented both as *Int*.

```
type Time       = Int
type DTime      = Int
type AgentId    = Int
data ABSEvent e = Tick DTime
                | DomainEvent AgentId e
```

The fact that Sugarscape schedules events without timestamps has also implications for the simulation kernel, which does not require a priority queue. Although the Sugarscape kernel also keeps track of the agents using an *IntMap* for the agent mappings, it uses a list to keep track of the events. Processing of events is implemented in the pure function *processEvents*, which takes the *EventList*, the simulation state, which contains the agent mappings amongst others, and returns the new simulation state as soon as the event list is empty, indicating that all events in the current *Tick* have been processed and agents are idle.

```
type EventList e = [(AgentId, ABSEvent e)] -- (receiver, ABSEvent)
processEvents :: EventList e -> SimulationState g -> SimulationState g
```

In each call, the function extracts the event at the front of the *EventList*. In case the list is empty it returns the simulation state unchanged, otherwise looks up the receiver and runs it with the given event. Newly scheduled events of the receiver are prepended at the front of the *EventList* through a recursive call. The initial *EventList* passed to *processEvents* is a list with *Tick* events scheduled for every agent, in random order. The fact that the events an agent schedules, are prepended to the front of the *EventList*, ensures that those events are processed next, which is of utmost importance for a correct working of the direct agent interactions discussed below. This also implies that *processEvents* is a potentially non-terminating function, in case there is at least one agent which produces at least one event for every event it receives.

5.2.2 Dynamic agent creation and removal

Some models of ABS in general and Sugarscape in particular require the dynamic creation and removal of agents during simulation. The specific require-

ments here are that the agents themselves must be able to both remove themselves from the simulation and create new agents with given attributes. To achieve that, in such a simulation the output type of an agent must be richer than the one in the event-driven SIR. First, we define the output of an agent:

```
data AgentOut m e o = AgentOut
  { aoKill   :: Any              -- True if this agent should be removed
  , aoCreate :: [AgentDef m e o] -- a list of agents to create
  , aoEvents :: [(AgentId, e)]   -- a list of events (receiver, event)
  }
```

Note that *AgentOut* contains already the list of scheduled events, which makes it clear that scheduling of events in this approach is implemented different than in the event-driven SIR, where the agents Monad stack had a *WriterT* to write events to. The reason for that is that we treat agent-local abstractions different here because of the need to encapsulate local agent state as explained in subsequent sections.

If the agent wants to remove itself from the simulation, it simply sets *aoKill* to True; if it wants to create new agents it adds an agent definition *AgentDef* to the *aoCreate* list. The agent definition *AgentDef* holds the new id of the agent [2], the MSF of the agent to create and the initial output of the new agent, so it has a representation in the visual or textual output for the current step without the need to run the new agent.

```
data AgentDef m e o = AgentDef
  { adId      :: AgentId          -- unique agent-id
  , adMSF     :: AgentMSF m e o   -- the agent behaviour function
  , adInitOut :: o                -- the value of the initial output
  }
```

Further, the simulation must provide a *global* mechanism to create new, unique *AgentId* for the newly created agents. The generating of ids for the new agents have to occur within the parent agents themselves because in some models they might need this very id to communicate with their children - an indirection through the kernel would only complicate matters. We thus start with a data definition, holding the next agent id - if an agent creates a new agent it simply reads that value and increments it by 1.

```
data ABSState = ABSState { absNextId :: AgentId }
```

To make it *globally* available to all agents a *StateT ABSState* Monad transformer is used, which is also the innermost Monad of the Monad stack of Sugarscape [3].

```
type AgentMonad m = StateT ABSState m
```

[2] Note that an agent-controlled id makes it possible to re-use ids in case an agent dies and in case ids have no other purposes than identifying event receivers in a model

[3] In the Sugarscape implementation, *ABSState* also holds the current simulation time, which is omitted here for clarity reasons.

Finally, we can define the polymorphic type of the agent MSF, as it is used in Sugarscape, where it is parametrised with model specific types (see next sections). It is similar to the event-driven SIR, where the agent takes the *ABSEvent* as input but the output is now a tuple of *AgentOut* and the polymorphic agent output type *o*. The reason why the output type *o* is not part of *AgentOut* is to keep *AgentOut* a Monoid, which allows accumulative / iterative change to *AgentOut*, which is important for creating new agents and scheduling events, as explained in the agent-local abstractions below.

```
type AgentMSF m e o = MSF (AgentMonad m) (ABSEvent e) (AgentOut m e o, o)
```

5.2.3 Shared mutable pro-active environment

In many agent-based models, agents are placed on a discrete 2D grid environment and can move around and interact with the environment. Often, there exist specific constraints, for example that each position can only be occupied by one agent at most. This requires specific iteration semantics, which make it impossible that two agents end up at the same time on the same spot. In general, such models solve this problem by using the sequential strategy as described in Chapter 3.1, where agents are run in random order, one after another. This allows the agents to access the globally shared, mutable environment exclusively when its their turn and interact and change it without the danger of other agents interfering.

To implement a shared mutable and pro-active environment, first we define a generic discrete 2D grid environment, with polymorphic cells. The selection of the right data structure is crucial. Initially we used an *IArray* from the array package. This data structure has excellent read performance but in performance tests we experienced serious performance and memory leak issues with updates, leading to allocation of about 40 MByte / second on our machine. Clearly this is unacceptable for simulation purposes, which often requires software to run for hours, and thus needs a constant memory consumption and must prevent even slowly linearly increasing memory usage under all costs. The solution was to switch to *IntMap* from the containers package as an underlying data structure which solved both the performance and memory leak issues.

```
type Discrete2dCoord  = (Int, Int)
type Discrete2dCell c = (Discrete2dCoord, c)
type Discrete2d c     = Map.IntMap (Discrete2dCell c)
```

Having introduced the *AgentMSF* and fixed the *AgentMonad* with the *StateT ABSState* as innermost Monad, adding a globally shared, mutable environment is straightforward. The solution is to simply add another *StateT* transformer with the given environment as type. Below, we give the parametrised definition as in the Sugarscape implementation. Note that Sugarscape closes the Monad stack with the *Rand* Monad as stochastics play an important role in the Sugarscape model as well. Therefore, a full expansion of the Monad stack used in Sugarscape is *StateT ABState (StateT SugEnvironment (Rand g))*.

```
data SugEnvSite = SugEnvSite
  { sugEnvSiteSugarLevel    :: Double
  , sugEnvSiteSpiceLevel    :: Double
  , sugEnvSitePolutionLevel :: Double
  ...
  }

type SugEnvironment  = Discrete2d SugEnvSite
type SugAgentMonad g = AgentMonad (StateT SugEnvironment (Rand g))
```

When implementing pro-activity of the environment, we must make a clear distinction between the environments data structure, how agents access it and the environments behaviour. In the Sugarscape model, the behaviour of the environment is quite trivial: it simply regrows resources over time and diffuses pollution in case pollution is turned on. This behaviour is achieved by providing a pure function without any monadic context or MSF. This is not necessary because the environment how we implement it, does not encapsulate local state and it does not interact with agents through messages and vice versa. Thus a pure function which maps the environment to the environment is enough: $Time \rightarrow SugEnvironment \rightarrow SugEnvironment$. Further it also takes the current simulation time so it can implement seasons, where the speed of regrowth of resources is different in different regions and swaps after some time. This function is called in the simulation kernel after every *Tick*.

Generally, one can distinguish between four different types of environments in ABS:

1. *Passive read-only* - implemented in Chapter 4.3, where the environment itself is not modelled as an active process and is static information, e.g. a list of neighbours, passed to each agent. The agents cannot change the environment actively - in the case of Chapter 4.3 this is enforced at compile time by simply making it read only, as input to the agent but not adding it to the output type of an agent. Note the agents change the environment implicitly by changing their state but there is no notion of an active environment process.

2. *Passive read/write* - The environment is just shared data, which can be accessed and manipulated by the agents. Note that this forces some arbitration mechanism to prevent conflicting updates e.g. running the agents sequentially one after the other, to ensure that only one agent has access at a time.

3. *Active read/write* - as implemented above. To make it active a pure function is used where the environment data is owned by the simulation kernel and then made available to the agents through a *State* Monad. Another approach would be to implement the environment process as an agent, which is run together with all the other agents. This allows the environment to send and receive messages but the guarantees about when the environment will be run is lost if agents are run sequentially in random order.

4. *Active read-only* - can be implemented as above but instead of providing the environment data through a *State* Monad, a *Reader* Monad is used. The environment data is owned by the Simulation kernel and the process runs as a pure function as before but the data is provided in a read-only way through the *Reader* Monad. Note that the same can also be achieved by passing it as input only to the agent as was done in Chapter 4.3.

5.2.4 Agent-local abstractions

After having established Sugarscapes full Monad stack, we can now move on to specify the agent behaviour and develop advanced agent-local concepts and abstractions. Before we can parametrise the *AgentMSF* we need to define model specific data definitions for the event type *e* and the output type *o*. Thus, we define the event type *SugEvent*, which defines all event types of Sugarscape and the output type *SugAgentObservable*, which contains all observable properties, an agent wants to communicate to the outside world, for visualisation or exporting purposes.

```
data SugEvent = MatingRequest AgentGender
              | MatingReply
                 (Maybe (Double, Double, Int, Int, CultureTag, ImmuneSystem))
              ...

data SugAgentObservable = SugAgentObservable
  { sugObsSugMetab :: Int
  , sugObsSugLvl   :: Double
  , sugObsAge      :: Int
  ...
  }
```

We can now parametrise the *AgentMSF* with the right types for the Sugarscape model.

```
type SugAgentMSF g = AgentMSF (SugAgentMonad g) SugEvent SugAgentObservable
```

Next, we define the type of the top-level agent behaviour function. We want to make the unique agent id and the model configuration (scenario) explicit, so it will be passed as curried arguments to the function. Further, the initial agent state is passed as curried input as well.

```
data SugarScapeScenario = SugarScapeScenario
  { sgScenarioName    :: String
  , sgPollutionActive :: Bool
  ...
  }

data SugAgentState = SugAgentState
  { sugAgSugarMetab :: Int
  , sugAgVision     :: Int
  , sugAgSugarLevel :: Double
  , ...
  }
```

```
type SugarScapeAgent g
       = SugarScapeScenario -> AgentId -> SugAgentState -> SugAgentMSF g
```

Now we have fully specified types for the Sugarscape agent. The types indicate very clearly the intention and the interface. What is of very importance is that we don't have any impure *IO* monadic context anywhere in our type definitions and we can also guarantee that it will not get sneaked in. The transformer stack of the agents MSF is terminated through the *Rand* Monad thus it is simply not possible to add other layers.

An agent is fully represented by a top level *SugarScapeAgent* function, which encapsulates the whole agent behaviour. We next look at how to define agent-local behaviour, which is hidden behind the *SugarScapeAgent* function type: whereas the previously defined types are exposed to the whole simulation, the following deals with types and behaviour which are locally encapsulated and hidden from the simulation kernel. In the next sections we show how to encapsulate the agents' state locally while retaining mutability. Further, we explain how sending of events works in the Sugarscape implementation and how to achieve read-only access to the agents unique id and the model configuration.

5.2.4.1 Agent-local state

To implement the local encapsulation of the agents' state is straightforward with MSFs as they are continuations, allowing them to capture local data using closures. Fortunately we do not need to implement the low-level plumbing, as Dunai provides us with a suitable function: *feedback :: Monad m* $\Rightarrow$ *c* $\rightarrow$ *MSF m (a, c) (b, c)* $\rightarrow$ *MSF m a b*. It takes an initial value of type *c* and an MSF which takes in addition to its input *a* also the given type *c* and outputs in addition to type *b* also the type *c*, which clearly indicates the read/write property of type *c*. The function returns a new MSF which only operates on *a* as input and returns *b* as output by running the provided MSF and feeding back the *c* (with the initial *c* at the first call).

```
sugarscapeAgent :: RandomGen g => SugarScapeAgent g
sugarscapeAgent scen aid s0 = feedback s0 (proc (evt, s) -> do ... )
```

Before we can move on to write a function handling incoming events, we need to define the *agent-local* Monad stack. The event handler must be able to manipulate the agent-local state we just encapsulated through *feedback*, support reading the unique agent id and model scenario and scheduling of events.

Providing the local, mutable agent state is done using a *State* Monad. Providing the model configuration (scenario) and the unique agent id is done using the *Reader* Monad. Implementing event scheduling, the same mechanism as in the event-driven SIR is used by a *Writer* Monad. As the *Monoid* instance to the *WriterT*, the *AgentOut* is used: all fields of its data definition are *Monoid* instances, making *AgentOut* a *Monoid* as well - writing a type class instance for it is trivial. Now it is also clear why *AgentOut* contains the list of events.

Further, this also allows to easily add new agent definitions and mark an agent for removal throughout the agents behaviour as both are also part of *AgentOut*.

We thus define the following Monad which is local to the agent and is only used *within AgentMSF*.

```
type AgentLocalMonad g = WriterT (SugAgentOut g)
                           (ReaderT (SugarScapeScenario, AgentId)
                             (StateT SugAgentState (SugAgentMonad g)))
-- FULLY EXPANDS TO:
-- WriterT (SugAgentOut g)
--  (ReaderT (SugarScapeScenario, AgentId)
--    (StateT SugAgentState
--      (StateT ABSState
--        (StateT SugEnvironment
--          (Rand g)))))
```

Now we can define the MSF which handles an event. It has the *AgentLocalMonad* monadic context, takes an *ABSEvent* parametrised over *SugEvent* (thus it has also to handle *Tick*). What might come as a surprise is that it returns unit type, implying that the results of handling an event are only visible as side effects in the monad stack. This is intended. We could pass all arguments explicitly as input and/or output but that would complicate the handling code substantially, thus we opted for a monadic, imperative style handling of events.

```
type EventHandler g = MSF (AgentLocalMonad g) (ABSEvent SugEvent) ()
```

To run the handler, which has an extended monadic context within the *SugarScapeAgent* we make use of Dunais functionality which provides functions to run MSFs with additional monadic layers within MSFs. We use *runStateS*, *runReaderS* and *runWriterS* (*S* indicates the stream character) to run the *generalEventHandler*, providing the initial values for the respective Monads: *s* for the *StateT*, *(params, aid)* for the *ReaderT* and the *evt* as the input to the event handler. Note that *WriterT* does not need an initial value, it will be provided through the Monoid instance of *AgentOut*.

```
sugarscapeAgent :: RandomGen g => SugarScapeAgent g
sugarscapeAgent scen aid s0 = feedback s0 (proc (evt, s) -> do
  (s', (ao', _)) <- runStateS
                      (runReaderS
                        (runWriterS generalEventHandler)) -< (s, ((scen, aid), evt))
  let obs = sugObservableFromState s
  returnA -< ((ao', obs), s'))

sugObservableFromState :: SugAgentState -> SugAgentObservable
generalEventHandler :: RandomGen g => EventHandler g
```

Now it is also clear why the output of an agent is a tuple of *AgentOut* and the polymorphic type *o*: the latter one is parametrised to *SugAgentObservable*, which is not constructed through the use of *WriterT* but simply a projection of the agent state through *sugObservableFromState*. In the next section we explain the details of *generalEventHandler*, which implements the main event handling mechanisms of an agent.

5.2.4.2 Handling and sending of events

Now we are ready to implement handling of events on an agent-local level: we receive the events from the simulation kernel as input and run within a 6-layered Monad transformer stack which is part global (controlled by the simulation kernel) and part local to the agent (controlled by the agent itself). The layers are the following (inner to outer):

1. *WriterT (SugAgentOut g)*: agent-local, provides write-only functionality for constructing the agent output for the simulation kernel which indicates whether to kill the agent, a list of new agents to create and a list of events to send to receiving agents.
2. *ReaderT (SugarScapeScenario, AgentId)*: agent-local, provides the read only model configuration and unique agent id.
3. *StateT SugAgentState*: agent-local, provides the local mutable agent state.
4. *StateT ABSState*: global, provides unique agent-ids for new agents.
5. *StateT SugEnvironment*: global, provides the sugarscape environment which the agents can read and write.
6. *Rand g*: global, provides the random-number stream for all agents.

Finally we can implement the event handler *generalEventHandler*, which simply matches on the incoming events, extracts data and dispatches to respective handlers. What is crucial here to understand is that only the top level *sugarscapeAgent* and the *EventHandler* function are MSFs which simply dispatch to monadic functions, implementing the functionality in an imperative programming style. The main benefit of the MSFs are their continuation character, which allows to encapsulate local state. Further the Dunai library adds a lot of additional functionality of composing MSFs and running different monadic context on top of each other. It even provides exception handling through MSFs with the *Maybe* type, thus programming with exceptions in ABS models can be done as well. Note that we didn't make use of it, as the Sugarscape model simply does not specify any exception handling on the model level and there was also no opportunity to use exceptions from which to recover on a technical level - there are exceptions on a technical level but they are non-recoverable and should never occur at runtime, thus *error* is used, which terminates the simulation with an error message.

```
generalEventHandler :: RandomGen g => EventHandler g
generalEventHandler =
  continueWithAfter -- optionally switching the top event handler
    (proc evt ->
      case evt of
        Tick dt -> do
          mhdl <- arrM handleTick -< dt
          returnA -< ((), mhdl)
```

```
        (DomainEvent sender (MatingRequest otherGender)) -> do
          arrM (uncurry handleMatingRequest) -< (sender, otherGender)
          returnA -< ((), Nothing)
        ...)

handleTick :: RandomGen g => DTime -> AgentLocalMonad g (Maybe (EventHandler g))
handleMatingRequest :: AgentId -> AgentGender -> AgentLocalMonad g ()
```

Note the use of *continueWithAfter*, which is a customised version of the already known *switch* combinator. It allows to swap out the event-handler for a different one, which is the foundation for the synchronous agent interactions, as discussed in the sections below.

To see how an event handler works, we provide the implementation of *handleMatingRequest*. It is sent by an agent to its neighbours to request whether they want to mate with this agent. The handler receives the sender and the other agents gender and replies with *sendEventTo* which sends a *MatingReply* event back to the sender. The function *sendEventTo* operates on the *WriterT* to append (using *tell*) an event to the list of events this agent sends when handling this event. Note the use of *agentProperty*, which reads the value of a given field of the local agent state.

```
handleMatingRequest :: AgentId
                    -> AgentGender
                    -> AgentLocalMonad g ()
handleMatingRequest sender otherGender = do
  -- check if the agent is able to accept the mating request:
  -- fertile + wealthy enough + different gender
  accept <- acceptMatingRequest otherGender

  -- each parent provides half of its sugar-endowment for the new-born child
  acc <- if not accept
      -- can't mate, simply send Nothing in MatingReply
      then return Nothing
      else do
        sugLvl  <- agentProperty sugAgSugarLevel
        spiLvl  <- agentProperty sugAgSpiceLevel
        metab   <- agentProperty sugAgSugarMetab
        vision  <- agentProperty sugAgVision
        culTag  <- agentProperty sugAgCultureTag
        imSysGe <- agentProperty sugAgImSysGeno -- immune system genotype
        -- able to mate, send Just share in MatingReply
        return Just (sugLvl / 2, spiLvl / 2, metab, vision, culTag, imSysGe)

  sendEventTo sender (MatingReply acc)
```

Finally we have a look at how to actually run an agents' MSF using the function *runAgentMSF*. It is a *pure* function as well and thus takes all input as explicit arguments. It might look like an overkill to pass in 5 arguments and get a 6-tuple as result but this is the price we have to pay for pure functional programming: everything is explicit, with all its benefits and drawbacks.

```
runAgentMSF :: RandomGen g        -- ^ RandomGen type class, g is a random-number generator
            => SugAgentMSF g      -- ^ The agents MSF to run.
```

```
            -> ABSEvent SugEvent  -- ^ The event it receives.
            -> ABSState           -- ^ The ABSState (next agent id and current time)
            -> SugEnvironment     -- ^ The environment state
            -> g                  -- ^ The random-number generator
            -> (SugAgentOut g, SugAgentObservable, SugAgentMSF g, ABSState, SugEnvironment, g)
runAgentMSF msf evt absState env g = (ao, obs, msf', absState', env', g')
  where
    -- extract the monadic function to run
    msfAbsState = unMSF msf evt
    -- peel away one State layer: ABSState
    msfEnvState = runStateT msfAbsState absState
    -- peel away the second State layer: SugEnvironment
    msfRand     = runStateT msfEnvState env
    -- peel away the 3rd and last layer: Rand Monad
    (((((ao, obs), msf'), absState'), env'), g') = runRand msfRand g
```

Note that we run only the 3 *global* monadic layers in here, the 3 *local* layers are indeed completely local to the agent itself as shown above.

5.2.5 Synchronous agent interactions

With the concepts introduced so far we can achieve already a lot in terms of agent interactions: agents can react to incoming events, which are either the *Tick* event advancing simulation time by one step or a message sent by another agent (or the agent itself). This is enough to implement simple one-directional agent interactions where one agent sends a message to another agent but does not await an answer within the same *Tick*. This one-directional interactions is used in the model to implement the passing of diseases, the paying back of debt, passing on wealth to children upon death - the agent simply sends a message and forgets about it.

Unfortunately this mechanism is not enough to implement the other agent-interactions in the Sugarscape model, which are structurally richer: they need to be synchronous. In the use cases of mating, trading and lending two agents need to come to an agreement over multiple interactions steps within the same *Tick* which need to be exclusive and synchronous. This means that an agent A initiates such a multi-step conversation with another agent B by sending an initial message to which agent B has to react by a reply to agent A who upon reception of the message, will pick up computation from that point and reply with a new message and so on. Both agents must not interact with other agents during this conversation to guarantee resource constraints, otherwise it would become quite difficult and cumbersome to ensure that agents don't spend more than they have when trading with multiple other agents at the same time. Also the initiating agent A must be able to pick up processing of its *Tick* event from the point where it started the conversation with agent B because sending a message always requires the handling of the current event to exit and hand the control back to the simulation kernel. See Figure 5.2 for a visualisation of the sequence of actions.

The way to implement this is to allow an agent to be able to change its internal event-handling state: to switch into different event-handlers, after hav-

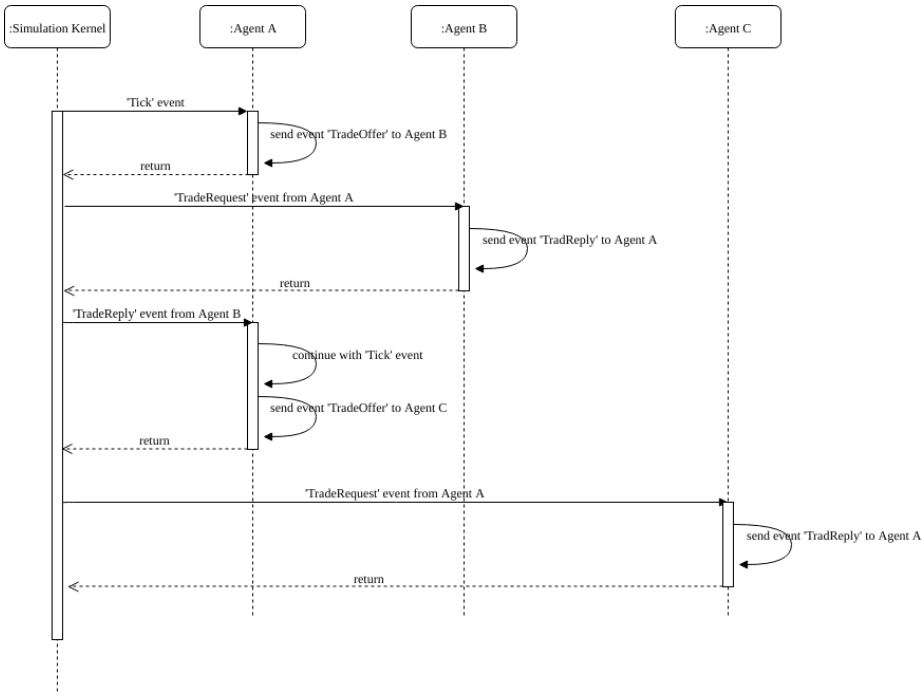


Figure 5.2: Sequence diagram of synchronous agent interaction in the trading use case. Upon the handling of the *Tick* event, agent A looks for trading partners and finds agent B within its neighbourhood and sends a *TradingOffer* message. Agent B replies to this message with *TradeReply* and agent A continues with the trading algorithm by picking up where it has left the execution when sending the message to agent B. After agent A has finished the trading with agent B, it turns to agent C, where the same procedure follows.

ing sent an event, to be able to react to the incoming reply in a specific way by encapsulating local state for the current synchronous interaction through closures and currying. Further, by making use of continuations the agent can pick up the processing of the *Tick* event after the synchronous agent interaction has finished. Key to this is the function *continueWithAfter* which we already shortly introduced through *generalEventHandler*. This function takes an MSF which returns an output of type *b* and an optional MSF. If this optional *Maybe* MSF is *Just* then the *next* input is handled by this new MSF. In case no new MSF is returned (*Nothing*), the MSF will stay the same. This is a more specialised version of the *switch* combinator introduced in Chapter 2.3.4 in the way that it doesn't need an additional function to produce the actual MSF continuation. Note that the semantics are different though: whereas *switch* runs the new MSF immediately, *continueWithAfter* only applies the new MSF in the *next* step. The implementation of the function is as follows:

```
continueWithAfter :: Monad m => MSF m a (b, Maybe (MSF m a b)) -> MSF m a b
continueWithAfter msf = MSF (\a -> do
  ((b, msfCont), msf') <- unMSF msf a
  let msfNext = fromMaybe (continueWithAfter msf') msfCont
  return (b, msfNext))
```

Finally, we can discuss the *Tick* handling function. It returns a value of type *Maybe (EventHandler g)* which if is *Just* will result in to a change of the top-level event handler through *continueWithAfter* as shown in *generalEventHandler* above. Note the use of continuations in the case of *agentMating, agentTrade* and *agentLoan*. All these functions return a *Maybe (EventHandler g)* because all of them can potentially result in synchronous agent interactions which require to change the top-level event handler. The function *agentDisease* is the last in the chain of agent behaviour, thus we are passing a default continuation which simply switches back into *generalEventHandler* to finish the processing of a *Tick* in an agent.

```
handleTick :: RandomGen g => DTime -> AgentLocalMonad g (Maybe (EventHandler g))
handleTick dt = do
  -- perform ageing of agent
  agentAgeing dt
  -- agent move, returns amount it of resources it harvested
  harvestAmount <- agentMove
  -- metabolise resources, depending on agents metabolism rate
  -- returns amount metabolised
  metabAmount <- agentMetabolism
  -- polute environment given harvest and metabolism amount
  agentPolute harvestAmount metabAmount

  -- check if agent has starved to death or died of age
  ifThenElseM
    (starvedToDeath `orM` dieOfAge)
    (do
      -- died of age or starved to deat: remove from simulation
      agentDies agentMsf
      return Nothing)
    -- still alive, perform the remaining steps of the behaviour
```

```
    -- pass agentContAfterMating as continuation to pick up after mating
    -- synchronous conversations have finished
    (agentMating agentMsf agentContAfterMating)

-- after mating continue with cultural process and trading
agentContAfterMating :: RandomGen g => AgentLocalMonad g (Maybe (EventHandler g))
agentContAfterMating = do
    agentCultureProcess
    -- pass agentContAfterTrading as continuation to pick up after trading
    -- synchronous conversations have finished
    agentTrade agentContAfterTrading

-- after trading continue with lending and borrowing
agentContAfterTrading :: RandomGen g  => AgentLocalMonad g (Maybe (EventHandler g))
agentContAfterTrading = agentLoan agentContAfterLoan

-- after lending continue with diseases, which is the step in a Tick event
agentContAfterLoan :: RandomGen g => AgentLocalMonad g (Maybe (EventHandler g))
agentContAfterLoan = agentDisease defaultCont

-- safter diseases imply switch back into the general event handler
defaultCont :: RandomGen g => AgentLocalMonad g (Maybe (EventHandler g))
defaultCont = return (Just generalEventHandler)
```

5.2.5.1 Tagless final

Although the indirect, continuation based approach to synchronous agent interactions as shown before works, it is quite cumbersome, fragile and it is easy to get something wrong. What would be more desirable is to have a truly synchronous approach, where the reply to an event happens directly as a result of the *sendEventTo* function: when calling *sendEventTo*, behind the scenes the receiving agent is executed and the result is returned directly to the caller, without any indirections. With the *tagless final* approach as introduced in the event-driven SIR above, this becomes possible in an elegant and robust way. We have developed this concept for the event-driven SIR only. Still, it should be equally applicable for the Sugarscape but we leave that for further research.

We start by extending the API by defining a *new* type class *MonadAgentSync*:

```
class Monad m => MonadAgentSync e m | m -> e where
  sendSync :: e -> AgentId -> m (Maybe [e])
```

The semantics behind the *sendSync* method shall be that it allows to send an event of type *e* to agent with id *AgentId*. If the agent cannot be found it will return *Nothing*, otherwise it will return *Just* the list of events the receiving agent replies to the sending agent.

The only place which has to be changed is the susceptible agent but due to the different semantics, parts of its behaviour needs to be rewritten. The receiving agents are left unchanged as at the moment a receiver has no means to distinguish between asynchronous and synchronous events and is not forced to reply to the sender in case of a synchronous event. It would be useful to have

some mechanism that in case of a synchronous event, the receiver can only reply to the sender. We leave that issue for further research.

Handling an incoming *Contact* from an *Infected* is no longer necessary as it will not happen, because the interactions go directly through *sendSync* and infected agents don't make contact pro-actively. Thus, the *MakeContact* handler has to be changed to take into account that the infection can happen directly there:

```
handleEvent MakeContact = do
  ais        <- getAgentIds
  ai         <- getMyId
  isInfected <- makeContact beta ai ais
  if isInfected
    -- got infected, signal event to switch
    then return (Just ())
    else do
      -- not infected, re-schedule MakeContact
      scheduleMakeContact
      return Nothing
```

The function *makeContact* recursively *makeContactWith* β (contact rate) other agents. Whereas previously, sending a *Contact* event to itself was not a problem, this is not allowed any more and must be prevented explicitly. The reason for that is discussed below in the implementation of the *sendSync* method of the pure interpreter. Note that sending to itself counts against the β contacts, as it would make no difference: receiving a *Contact* from a *Susceptible* has no effect on a susceptible agent anyway. If the case arises in a model that agents need to send events to themselves, it cannot happen through mechanisms like *sendSync* but it has to go through asynchronous scheduling of events which decouples the sending from the receiving.

```
makeContact :: (MonadAgent SIREvent m, MonadAgentSync SIREvent m)
            => Int       -- ^ number of contacts to make
            -> AgentId   -- ^ sender agent id
            -> [AgentId] -- ^ all agent ids (including self)
            -> m Bool
makeContact 0 _ _ = return False
makeContact n ai ais = do
  receiver <- randomElem ais
  -- prevent sending to self
  if ai == receiver
    -- self counts against beta contacts
    then makeContact (n-1) ai ais
    else do
      -- make contact
      ret <- makeContactWith receiver
      if ret
        -- got infected, stop
        then return True
        -- not infected, continue
        else makeContact (n-1) ai ais
```

Finally we can use *sendSync* to directly send events to a receiving agent, which replies with a list of events to the sender. Note that we needed to add

the new *MonadAgentSync* type class to the overloaded function, to make the *sendSync* method available.

```
makeContactWith :: (MonadAgent SIREvent m, MonadAgentSync SIREvent m)
                => AgentId -> m Bool
makeContactWith receiver = do
  ai     <- getMyId
  -- DIRECT SYNCHRONOUS AGENT INTERACTION HAPPENS HERE
  retMay <- sendSync (Contact ai Susceptible) receiver
  case retMay of
    -- receiver not found, no infection
    Nothing -> return False
    -- receiver found, replied with es events
    (Just es) -> do
      -- check if any event in replies is from Infected
      let fromInf = any (\ (Contact _ s) -> s == Infected) es
      if not fromInf
        -- none from Infected, no infection
        then return False
        -- at least one from Infected, might become infected
        else do
          r <- randomBool inf
          if r
            -- got infected, become infected
            then do
              scheduleRecoveryM ild
              return True
            else return False
```

The *sendSync* method needs to be implemented in a new *pure* interpreter, which follows exactly the same concept as before so we briefly discuss it conceptually. The method looks up the receiving agent, runs it with the given event of the sender and filters the replies to the sending agent. To do that, the method needs to have all agents available to actually execute them. This is achieved by keeping the agent mappings in the *SimState*, introduced in the initial section on *tagless final* of the event-driven SIR. They are managed using a *State* Monad, thus can be read and written, which both is necessary as after a successful run of the receiving agent, its new MSF needs to be put back into the agent mappings.

Now it becomes clear why an agent cannot send an event with *sendSync* to itself and why circular (agent A sendSync to agent B sendSync to agent C sendSync to agent A) *sendSync* are also not allowed. The new MSF of an agent which was just run and updated in the *SimState* will be overridden by the subsequent updates of runs which were initiated earlier. Fortunately, this can be conveniently checked within the *sendSync* method and an error or exception can be generated which is better than silently ignoring it, resulting in unexpected behaviour. It is possible because the initiating agents id is always known and because it is easy to keep track of the agent ids currently engaged in a *sendSync* by storing their ids in the *SimState*, thus arriving at some kind of call stack management.

Although the very direct *tagless final* agent interaction approach makes things under certain circumstances easier, it comes also with subtle drawbacks, thus it depends on the model semantics which approach to synchronous agent

interactions should be chosen. Still we think that this approach is another demonstration of the usefulness of a *tagless final* approach: we have shown how to extend the existing API with new operations without breaking the existing implementation. Also we think that we only scratched the surface with this approach of direct synchronous agent interactions but we leave a more in-depth exploration of it for further research.

5.3 Discussion

Transforming a time-driven into an event-driven approach should always be possible because the ability to schedule events with timestamps allows to map all features of time-driven ABS to an event-driven one - the discussion above should give a good direction of how this process works. Still for some models one can argue that the time-driven approach is much more expressive than an event-driven one, and we think this is certainly the case for the SIR model. The event-driven approach leads to much more fragmented logical flow and agent behaviour especially in the case of synchronous agent interactions. Still, we have shown a possible direction of reducing the fragmentation using the *tagless final* approach.

5.3.1 A related approach

The work of [20] tries to solve a similar problem. The authors also use Haskell to implement ABS and more specifically looked into the use of messages and the problem of when to advance time in models with arbitrary number synchronised agent interactions. The biggest difference is, that we approach our agents fundamentally different through the use of Monads and FRP. First in our approach an agent is only a single MSF and thus can not be directly queried for its internal state / its id or outgoing messages, instead of taking a list of messages, our agents take a single event/message and can produce an arbitrary number of outgoing messages together with an observable state - note that this would allow to query the agent for its id and its state as well by simply sending a corresponding message to the agents MSF and requiring the agent to implement message handling for it. Also the state of our agents is *completely* localised and there is no means of accessing the state from outside the agent, they are thus "fully encapsulated agents" [20]. Note that the authors of [20] define their agents with a polymorphic agent-state type s, which implies that without knowledge of the specific type of s there would be no way of accessing the state, rendering it in fact also fully encapsulated. The problem of advancing time in our approach is conceptually very similar though: after sending a tick message to each agent (in random order), we process all agents until they are idle: there are no more enqueued messages / events in the queue. The similarities in both approaches might hint at that this seems to be indeed the "right" way to go.

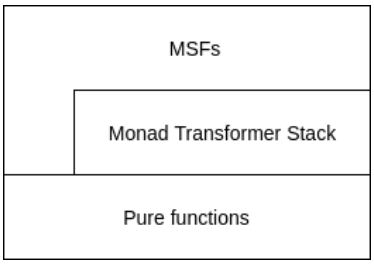


Figure 5.3: The architecture as a 3-layered system.

5.3.2 Layered architecture

The approach is designed as a 3-layered architecture, see Figure 5.3:

1. *Pure functions* are the working horses, which do the actual computations of the simulation. They are mostly used to build up the 2nd layer. Also layer 1 might access them to achieve pure computations when there is no need for effects.

2. *Monad transformer stack* (global and local) does the dirty work of effectful computation: sending messages, mutating the environment, reading model configuration, drawing random numbers, mutating agent state. This layer uses the pure functions to build up its functionality and also propagates between the 1st and 3rd layer.

3. *MSFs* (arrowized FRP) are the backbones of the architecture and define the dynamical structure of the system. This layer builds heavily on the 2nd layer and can also be seen as a highly delegation mechanism. Note that MSFs blur the distinctions between the monadic and the arrowized layer.

Separating those 3 concerns from each other makes the code more robust, easier to refactor and maintain. Further it makes code *much* easier to test as will be shown in Chapter IV.

5.3.3 Imperative nature

Both event-driven use cases make heavy use of the State Monad, thus one might ask what the benefits are of our pure functional approach - after all we seem to fall back into stateful, imperative style programming. On the other hand even our stateful programming is highly restricted to very specific types and operations. Further, in our monad stack we control the operations possible to the respective layers: e.g. sending messages/events is a write-only operation (as it should be), accessing the unique agent-id and the model-configuration is read-only (as it should be). All this is guaranteed at compile time, which makes it much more manageable, maintainable, robust, composable and testable. To

quote John Carmack [4]: *"A large fraction of the flaws in software development are due to programmers not fully understanding all the possible states their code may execute in."*. We claim that despite using an imperative style, the static guarantees of the types we operate on and the operations provided, it makes it easier to fully understand the possible states of the simulation code. This is directly related to the power of polymorphism in Haskell, which goes far beyond the polymorphism of existing object-oriented [5] languages. We see a particular instance of that in the polymorphism we developed in the concepts behind Sugarscape: we can compose effects depending on the model and we can easily swap out environment and events with very few changes with the benefit that the compiler will inform us about breaking changes. This is directly related to refactoring, which is very convenient and quickly becomes the standard in the development process: guided by defining types first and relying on the compiler to point out problems, results in very effective and quick changes without danger of bugs showing up at run time. This is not possible in dynamic object-oriented languages like Python because of its lack of compiler and types, and much less effective in Java due to its dynamic type system which is only remedied through strong IDE support.

5.3.4 Handling IO

This thesis directly capitalises on the fact that most ABS models are primarily of computational nature, thus CPU bound and do not involve IO *inside the agents* while running the simulation. The concurrent approach with STM is an exception but at least we retain the guarantees that the non-determinism within the agent behaviour originates from the concurrency using STM and nothing else. Even if some IO is required, like rendering the simulation as we did in Sugarscape, due to the loose coupling and compositional qualities of pure functional programming it is straightforward to separate these concerns and keep the impure rendering parts from the pure agent behaviour. If there arises the use case where agents absolutely need to perform some impure IO within their behaviour then there exist three options. The first one is to let agents construct *IO* actions and pass them to the simulation kernel for execution, requiring the simulation kernel to run now in *IO* instead of being pure. This is especially suitable for one-way *IO* actions, where an agent does not need to synchronously wait for a result. If a synchronous *IO* action is required with the agent waiting for a result, it could be communicated back from the simulation kernel. This keeps the agent behaviour still pure but with the consequence of indirection and higher complexity. The second option is to use a *tagless final* approach as discussed in Chapter 5.1.4, where the actions requiring IO are abstracted behind methods of the given type class, for which then an interpreter running in *IO* has to be written. The benefit is that this allows for a direct synchronous IO behaviour will still restricting the available operations to only the required ones

[4] `http://www.gamasutra.com/view/news/169296/Indepth_Functional_programming_in_C.php`

[5] Polymorphism is *not* unique to object-oriented programming.

instead of running fully in the *IO* Monad as would be the third option. In all cases everything becomes possible and all bets are off regarding static guarantees and reproducibility, whereas the *tagless final* approach provides most control.

5.3.5 Multiple types of agents

In the Sugarscape example we have only considered one type of agents, thus the whole population is a homogeneous one in regards of the *type* of the agent. It is quite straightforward to have heterogeneous agent types as well, which is accomplished through adding additional data declarations to the observable output and the agent-state. A consequence is that all agent types have to speak the same event-language because in regards of types the agents are treated the same way - this is also true for the monadic / effect stack: different agent types cannot have different effect types in this approach as they are seen as the same on the type level.

5.3.6 Performance

The event-driven implementation from this Chapter is around 60 - 70% faster than the time-driven implementation from Chapter 4.1, which is non-monadic and uses the FRP library Yampa. For the monadic time-driven approach of Chapter 4.3 the difference is much more dramatic: it is about 700 - 800% slower. These results dramatically highlight the problem of time-driven ABS: its performance cannot compete with an even-driven approach. This is exaggerated even more so when making use of MSFs as in Chapter 4.3. In this case, a time-driven approach becomes extremely expensive in terms of performance and one should consider an event-driven approach. In case the model is specified in a time-driven way, a transformation into an event-driven approach should always be possible as outlined above.

We compared an event-driven SIR implementation we did in Java to the Haskell one here. We run for 150 time steps with 1,000 susceptible and 1 infected agent, $\beta = 5$, $\gamma = 0.05$, $\delta = 15$. Further, we fixed the random-number generators to guarantee identical dynamics in every run and averaged 8 runs. The Java implementation averages at 1.2 seconds, whereas the Haskell implementation at 6.8 seconds. These performance figures are closer than the ones in the time-driven approach of the previous chapter. This shows that event-driven is indeed much better performing and also more flexible as [107] has pointed out. Interestingly, the time-driven Java implementation outperforms the event-driven one. Although, we have improved the performance substantially compared to the time-driven approach, we address it more in-depth in the chapters on parallelism 6 and concurrency 7.

5.3.7 Conclusion

Overall we think that this event-driven approach is quite feasible and is *the way to go* to implement ABS in a pure functional way. The time-driven approach is

quite expressive but is not as flexible and general as the event-driven one. Also performance is considerably better in event-driven approach.

We conclude that synchronous agent interaction was the most difficult part to figure out and get right and thus posed the greatest challenge. This concept is indeed cumbersome and clearly more complex than direct method invocation in object-oriented programming. Unfortunately, with the goal of staying pure we do not have much other options. Note, that we didn't aim to encapsulate its complexity behind domain-specific combinators but this is certainly possible and should reduce the difficulty and complexity considerably. This is left as further research and open work which should be untertaken in the future, when putting all the concepts of this thesis into a general purpose library for pure functional ABS in Haskell.

Part III:

Parallel computation

In the introduction in Chapter 1, this thesis hypothesised that functional programming should allow to easily apply parallel computation to ABS - this part tries to answer this hypothesis. Thus in this part of the thesis we perform a deeper investigation on the potential of parallel programming offered by pure functional programming to apply to ABS. Another motivation this part is the notorious performance and efficiency problems of our sequential implementations (see Chapter 10.1.1 for a summary), thus the work in this part can be seen as an attempt to at least mitigate the notorious performance problems of functional programming.

Pure functional programming as in Haskell is well known and accepted as a remedy against the difficulties and problems of parallel computation [76]. The reason for it is clear: immutable data and explicit control of side effects removes a large class of bugs due to data conflicts and data races. A fundamental benefit and strength of Haskell is, that it clearly distinguishes between parallelism and concurrency *in its types* [91]. It is very important for us to do so as well:

- **Parallelism** - In parallelism, code runs in parallel solely for the purpose of doing more work within the same time, without interfering with other code through shared data (references, mutexes, semaphores,...). An example is the function *map :: (a → b) → [a] → [b]*, which maps each element of type *a* to *b* using the function *(a → b)*. It is a pure function and thus no sharing of data either through some monadic context or through the function *(a → b)* is possible. This allows to run it in parallel: each function evaluation *(a → b)* could potentially be executed at the same time, if we had enough CPU cores. Whether it runs actually in parallel or not, has no influence on the outcome, it is not subject to any non-deterministic influences. Thus we identify parallelism with pure and deterministic execution of data transformations in parallel (data parallelism).

- **Concurrency** - Concurrency refers to the decomposability property of a program, algorithm, or problem into order-independent or partially-ordered components or units [97]. Those parts *can* be run in parallel which as a consequence *might* give rise to asynchronous, non-deterministic events [6].

[6] Note that the functional *concurrent* programming language Erlang [5], which uses the actor model for its concurrency model, was single-threaded from its conception in 1986 until around 2008. This might sound surprising but underlines the fact that concurrency per se has nothing to do with parallel execution.

> An example are two threads, running in parallel, which share data through a reference. Depending on the scheduling and the code which is run in each thread, this gives rise to very different access patterns - the events - to the shared data, with the potential for race conditions and dirty reads. In concurrency per definition, ordering is important and the challenge of implementing parallel, concurrent programs, is to write the program in a way that despite of these non-deterministic events it is still a correctly working program. Thus we identify concurrency with parallel, impure, non-deterministic execution of imperative-style and ordered monadic evaluation.

In the next two chapters we investigate the application of both parallelism and concurrency to our pure functional ABS approach. In general, we want to see if and how parallel and concurrent programming in Haskell is transferable to pure functional ABS and what the benefits are. In particular we are interested in speeding up the existing implementations by generally developing techniques that allow us to *run agents in parallel* [7].

Note that the focus here is primarily on the conceptual nature of how to apply parallelism and concurrency to pure functional ABS, thus we refrain from doing in-depth performance analysis up-front as it is beyond the scope of this work. Still, we are very well aware that mindlessly trying to apply parallel computation can actually result in loss of performance as a problem can only be sped up in so far as we can partition it and run those partitions in parallel. Further, parallel computation comes with an overhead and if the partitioning is too fine-grained, this overhead might eat up the speedup or make it even worse. Thus, in real-world problems, performance measurements have to come first, then one can investigate where and why the performance is lost. Only if this is understood properly one can decide whether parallelism or concurrency is applicable - or none at all because the problem is actually completely sequential.

Besides performance improvement, we are generally interested in the implications of the way Haskell deals with parallelism and concurrency in its types. In particular we ask about the ability of keeping deterministic guarantees about the reproducibility of our simulations. We hypothesise that parallelism will allow us to retain *all* static guarantees about reproducibility *and* gives us a noticeable speedup. Further we hypothesise, that in concurrency we might see a bigger speedup but sacrifice the very guarantee about reproducibility. However, we assume that by using Haskell's unique approach to Software Transactional Memory (STM), we don't lose this guarantee completely - it will just get weakened by guaranteeing that the non-deterministic influence is through concurrency only *and nothing else*.

[7] Note that we use the term *parallel* to identify both *parallelism* and *concurrency* and we distinguish between them whenever necessary using their respective terms.

Chapter 6

Parallelism in ABS

The promise of parallelism in Haskell is compelling: speeding up the execution but retaining all static compile time guarantees about determinism. In other words, using parallelism could give us a substantial performance improvement without sacrificing the static guarantees of reproducible outputs from repeated runs with initial conditions.

Generally, parallelism can be applied whenever the execution of code is order-independent, that is referential transparent, and has no implicit or explicit side effects. In this section we introduce the two most important parallelism concepts of Haskell, *evaluation* and *data-flow* parallelism, and discuss their potential use in pure functional ABS in general. We follow [104] and refer to it for an in-depth discussion. Further, we show how these concepts can be added to our previously discussed use cases of Chapters 4.1, 4.3 and Sugarscape 5.2 and compare their performance over the original sequential approaches.

6.1 Evaluation parallelism

Evaluation parallelism introduces so called strategies to evaluate lazy data structures in parallel. Examples are strategies to evaluate a list, or tuples in parallel where for each element a spark is created. The fundamental concept Haskell uses to achieve evaluation parallelism is its own non-strictness nature. Non-strictness means that expressions are not eagerly evaluated when defined, like in imperative programming languages but only evaluated when their result is actually needed. This is implemented internally using thunks, which are pointers to expressions. When the value of an expression is needed, this thunk is accessed and the expression is reduced until the next data constructor or Lambda expression is encountered. This is called Weak Head Normal Form (WHNF) evaluation because it only reduces the "head" of the expression, which could consist of sub expressions. This indirection, the separation of data creation from consumption / evaluation, indeed enables evaluation parallelism and Haskell provides two additional functions to support this:

- *par* :: $a \to b \to b$ Returns the second argument b but evaluates the first argument a in parallel. It is used when the result of evaluating a is required later.
- *seq* :: $a \to b \to b$ - Returns the second argument b but is strict in its first argument, which means it forces its evaluation to WHNF. It is used when the result of evaluating a is required now.

Internally, evaluation parallelism is handled through so called *sparks*, which are thunks evaluated in parallel. The Haskell runtime system manages sparks and distributes them to threads where they get executed. Due to their extremely light-weight nature, it is no problem to create tens of thousands of sparks. One has to bear in mind that although evaluating in parallel through sparks is extremely cheap, it still has some overhead. Thus, if the work-load of each element in a list might be too low for a spark, then one can distribute chunks of a list onto a single spark. It is important to understand, that all this works without side effects - the strategy combinators are all pure functions building on *par* and *seq*. This allows us to add parallelism to an algorithm by applying a parallel evaluation strategy to its result which e.g. is a lazy list - again this is possible through non-strictness, which separates the construction of data from its consumption.

6.1.1 Evaluation parallelism in ABS

Using compositional parallelism is exactly what we use to aim at adding evaluation parallelism for agent execution in the non-monadic SIR example 4.1. We know that the whole simulation is a completely pure computation because Yampa is non-monadic, thus it is guaranteed that there are no side effects. Thus, agents are run conceptually in parallel using *map*, which should enable us to add parallelism without needing to re-implement *dpSwitch* (the function running the agents in parallel).

The solution is to add evaluation parallelism in the agent-output collection phase: where the recursive switch into the *stepSimulation* function happens. It is there, where we use an evaluation strategy to evaluate the outputs of all agents in parallel. The agents will then be evaluated in parallel due to compositional parallelism, when we force the output of each in parallel. We give more details in the short case study 6.3.1 below.

6.2 Data-flow parallelism

When relying on a lazy data structure to apply parallelism is not an option, evaluation strategies as presented before are not applicable. Further, although lazy evaluation brings compositional parallelism, it makes it hard to reason about performance. Data-flow parallelism offers an alternative over evaluation strategies, where the programmer can give more details but gains more control: data dependencies are made explicit and reliance on lazy evaluation is avoided

[105]. Data-flow parallelism is implemented through the *Par* Monad, which provides combinators for expressing data flows: in this Monad it is possible to *fork* parallel tasks which communicate with each other through shared locations, so called *IVar*s. Internally these tasks are scheduled by a work-stealing scheduler, which distributes the work evenly on available processors at runtime. *IVars* behave like futures or promises: they are initially empty and can be written once. Reading from an empty *IVar* will cause the calling task (or main thread) to wait until it is filled. An example is a parallel evaluation of two fibonacci numbers:

```
runPar (do
  i <- new             -- create new IVar
  j <- new             -- create new IVar
  fork (put i (fib n)) -- fork new task compute fib n and put result into IVar i
  fork (put j (fib m)) -- fork new task compute fib m and put result into IVar j
  a <- get i           -- wait for the result from IVar i and collect it
  b <- get j           -- wait for the result from IVar j and collect it
  return (a,b)         -- return the sum
```

Note that data-flow parallelism makes it possible to express parallel evaluation of a list or a tuple as with evaluation strategies. The difference though is, that it does avoid lazy evaluation. More importantly, putting a value into an *IVar* requires the type of the value to have an instance of the *NFData* type class. This simply means that a value of this type can be fully evaluated, not just to WHNF but to evaluate the full expression to the value it represents.

6.2.1 Data-flow parallelism in ABS

The *Par* Monad seems to be a very suitable mechanism to enable agents to express data-flow parallelism within their behaviour. This is only possible with the monadic ABS approach as in the SIR implementation of Chapter 4.3 and the Sugarscape of Chapter 5.2. An important fact is that if the *Par* Monad is used, it has to be the outermost monad because it cannot be a transformer. This is emphasised by the fact that there exists no *ParT* transformer instance, like for other Monads. Making the *Par* Monad a transformer would have the semantics of running the *bind* in parallel. It is quite clear that this simply makes no sense: *bind* is a function for composing / sequencing monadic actions, which in general involves side effects of some kind. Side effects inherently impose some sequencing where evaluation of different sequences has different meanings in general - thus the sequential nature of *bind*. Therefore, running monadic code in parallel is simply not possible in general due to side effects [1] and thus there is no (meaningful) way to put the *Par* Monad into a transformer stack.

[1] Besides, it would be not very clear what we are running in parallel within the *bind* operator as there is nothing to parallelise in general e.g. no structure over which we can parallelise in general.

6.3 Case studies

In this section we go a little bit more into detail how to apply the parallelism concepts as already outline above to our use cases from Chapters 4.1, 4.3 and Sugarscape 5.2. We briefly demonstrate the technical details and refer to the full code in footnotes. Note that all timings are rough averages over multiple runs and not precise measurements because that is not the point here. We are only interested in showing what rough potential there is for speeding up computation through deterministic parallelism - we are not interested in high performance computation here but rather in conceptual comparisons between sequential and parallel implementations.

6.3.1 Non-monadic SIR

Evaluation Strategies As outlined above we want to apply parallelism to agent evaluation by composing the output with parallel evaluation by slightly changing the function *switchingEvt*. This function receives the output of all agents from the current simulation step and generates an event to recursively switch back into *stepSimulation* to compute the next simulation step. The code is as follows:

```
switchingEvt :: SF ((), [SIRState]) (Event [SIRState])
switchingEvt = arr (\ (_, newAs) -> parEvalAgents newAs)
  where
    -- NOTE: need a seq here otherwise would lead to GC'd sparks because
    -- the main thread consumes the output already when aggregating, so using seq
    -- will force parallel evaluation at that point
    parEvalAgents :: [SIRState] -> Event [SIRState]
    parEvalAgents newAs = newAs' `seq` Event newAs'
      where
        -- NOTE: chunks of 200 agents seem to deliver the best performance
        -- when we are purely CPU bound and don't have any IO
        newAs' = withStrategy (parListChunk 200 rseq) newAs
        -- NOTE: alternative is to run every agent in parallel
        -- only use when IO of simulation output is required
        -- newAs' = withStrategy (parList rseq) newAs
```

Which evaluation strategy results in the best performance depends on how we observe the results of the simulation. Due to Haskells non-strict nature, as long as no output is *observed*, nothing would get computed ever. We have developed three different ways to observe the output of this simulation and thus we measured the timings for all of them:

1. Printing the output of the last simulation step. This requires to run the simulation for the whole 150 time steps because each step depends on the output of the previous one. Because the simulation is completely CPU bound, the best performance increase turned out to run agents in batches where for this model 200 seems to deliver the best performance. If each agent is run in parallel, we still achieved a substantial performance increase but not as high as the batched version. An analysis showed that around

Output type	Parallel	Sequential	Factor
Print of last step (1)	3.9	16.38	4.24
Writing simulation output (2)	9.41	10.17	1.08
Appending current step (3)	9.73	10.04	1.03
(1) and (2) combined	5.02	19.68	3.92

Table 6.1: Timings of parallel vs. sequential non-monadic SIR.

1.5 million (!) sparks got created but most of them were never evaluated. There is a limit in the spark pool and we have obviously hit that.

2. Writing the aggregated output of the whole simulation to an export file. This requires in principle to run the simulation till the last time step but due to non-strictness, the writing to the export file begins straight away. This interferes with parallelism due to system calls which get interleaved with parallelism, leading to less performance increase than the previous one. It turned out that in this case running each agent in parallel didn't lead to reduced performance, because we are IO bound (see below).

3. Appending the aggregated output of the current step to an export file. This is necessary when we have a very long running simulation for which we want to write each step to the file as soon as it is computed. The function which runs this simulation is tail-recursive and can thus run forever, which is not possible in the previous case where the function is not necessarily tail-recursive and aggregates the outputs. Here we use a strategy which evaluates each agent in parallel as well.

4. A combined approach of 1 and 2 where the output of the last simulation step is printed and then the aggregate is written to a file.

The timings are reported in Table 6.1. All timings were measured with 1000 agents running for 150 time steps, and $\Delta t = 0.1$. We performed 8 runs and report the average timings in seconds. The parallel version was compiled with the '-threaded' option and used all 8 cores with the '-N' option. For the sequential implementation the '-threaded' option was removed as well as the evaluation strategies - it is purely sequential code. All experiments were carried out on the same machine [2]

The table clearly indicates, that in case we are purely CPU bound we get a quite impressive speedup of 4.24 on 8 cores - parallelism clearly pays off here, especially because it is so easy to add. On the other hand it seems that as soon as we are IO bound, the parallelism performance benefit is completely wasted. This does not come as a surprise and it is well established that generally as soon as IO is involved, performance benefits from parallelism will suffer. This point will be addressed by the use of concurrency where due to concurrent evaluation,

[2]Dell XPS 13 (9370) with Intel Core i7-8550U (8 cores), 16 GB, plugged in.

IO is decoupled from the computation, making the latter one completely CPU bound and resulting in an impressive speedup in such a case as well.

What comes a bit as a surprise is that in the case of the sequential implementation, the CPU bound implementation, which does no IO is actually slower than the ones which do IO. This can be attributed to lazy evaluation which seems to increase performance because IO can be performed actually while the simulation computes the next step, interleaving the evaluation and IO. Thus when comparing the parallel CPU bound approach (1) to the IO bound sequential ones (2) and (3) results in a lower speedup factor of roughly 2.6. The combined approach (4) then shows that we actually can have the substantial speedup of CPU bound (1) but still write the result to the file like in (2). This is of fundamental importance in simulation, because after all they often produce massive amounts of data which need to be stored somewhere.

Par Monad The book [104] mentions that *Par* Monad and evaluation strategies result roughly in the same performance in most of the benchmarks. And indeed: we also applied Par Monad here to run the agents in parallel by evaluating their output and we get about the same speedup in cases (1) and (4). The IO bound cases (2) and (3) perform slower: (2) is nearly 50% slower than its evaluation strategy pendant and (3) is about 25% slower. What is interesting is, that running all agents in their own task seems to be fine here in any case whereas it was slower in the evaluation strategy in the CPU bound case:

```
-- NOTE: with the Par monad, splitting the list into chunks seems not
-- to be necessary - we get the same speedup as in evaluation strategies
parMonadAgents :: [SIRState] -> Event [SIRState]
parMonadAgents newAs = Event (runPar (do
  -- simply return the value of the agent, resulting in a deepseq due to
  -- NFData instance of put in IVar
  ivs <- mapM (spawn . return) newAs
  mapM get ivs))
```

6.3.2 Monadic SIR

We can try to apply the same techniques of parallelising the agents as we did in the previous section in the non-monadic version of the SIR model. There is but a fundamental problem in this case, as we have already outlined in the section on data-flow parallelism: we are running the simulation in the monadic context of a *ReaderT* and *Rand* Monad stack. In monadic execution, depending on the monad (stack), we deal with side effects, which immediately necessitates the ordering of execution: whether an effectful expression is evaluated before another one can have indeed very fundamental differences and in general we have to assume that it does. Indeed: the way the agents are evaluated is through the *mapM* function, which evaluates them sequentially applying their side effects in sequence. It does not matter that the agents behave as if they are run in parallel without the possibility to interfere with each other, the simple fact that they are run within the *ReaderT (Rand g)* transformer stack requires sequencing. It is not the *ReaderT* which causes the delicate issue, it is rather the *Rand*

Monad, which basically behaves like a *State* Monad with the random-number generator as internal state, which gets updated with each draw. Due to this sequential evaluation, we can hypothesise that our approach is doomed from the beginning and that we will not see any speedup when we apply parallelism - on the contrary, we can expect the performance to be worse with it due to the overhead caused by it.

Indeed, when we put our hypotheses to a test [3] we see exactly that behaviour: the sequential implementation, which does not use any parallelism and is not compiled with the -threaded option takes on average 41.76 seconds to finish. When adding parallelism with evaluation strategies in the same way as we did in non-monadic SIR, we end up with 49.63 seconds on average to finish - a clear performance *decrease*! For the *Par* Monad approach its even worse, which averages at 52.98 seconds to finish. These timings clearly show that 1) agents which are run in a monadic context with *mapM* are not applicable to parallelism, 2) the parallelism mechanisms add a substantial overhead which is in accordance with the reports in [104].

Still we don't give up completely and want to see if running the agents sequentially but some *Par* monadic code *within* them could gain us some speedup. The function we target is the neighbourhood querying function, which looks up the 8 (Moore) surrounding neighbours of an agent. It is a pure function and uses *map* and is thus perfectly suitable to parallelism. We simply extend the transformer stack by putting the *Par* Monad innermost and then run the *neighbours* function within the *Par* Monad:

```
-- type simplified for explanatory reasons
neighbours :: Disc2dCoord -> SIREnv -> Par [SIRState]
neighbours (x, y) e = do
    ivs <- mapM (\c -> spawn (return (e ! c))) nCoords
    mapM get ivs
  where
    nCoords = ... -- create neighbours coordinates
```

Unfortunately the performance is even worse than without it, averaging at 66.68 seconds to finish. The workload seems to be too low for parallelism to pay off. Further, when keeping the *Par* Monad as outermost Monad but using the original pure *neighbours* function without *Par* we arrive at an average of 55.9 seconds to finish when running multi-threaded on 8 cores and 45.56 seconds when compiled with threading enabled but running on a single core. These measurements demonstrate that using the *Par* Monad and parallelism in general can lead to a substantially *reduced* performance, due do massive overhead and too fine-grained parallelism.

This leaves us basically without any options of parallelism for the monadic SIR model. Still, we will come back to this use case in the chapter on concurrency, where we will show that by using concurrency it is possible to achieve a substantial speedup even in monadic computations.

[3] We used the same experiment setup as in the non-monadic implementation.

6.3.3 Sugarscape

As already shown in the case of the monadic SIR implementation from the previous section, running agents in parallel within a monadic context does not bring us any speedup - on the contrary, we get penalised with a substantial performance loss due to the overhead incurred by adding parallelism. Further, running the agents in the *Par* Monad alone incurs also a substantial overhead, thus ultimately these roads are dead ends for our Sugarscape implementation as well.

Still, there is one direction for parallelism left, we didn't explore so far: the behaviour of the environment, which is a pure computation, using maps and folds [4] over an *IntMap*. It might look like we are out of luck as it seem that we cannot parallelise the updating of an *IntMap* but the work of [104] shows that it is indeed possible through a combination of the *Par* Monad and the *Applicative* type class. The *IntMap* provides the function *traverseWithKey :: Applicative t* $\Rightarrow$ *(Key* $\rightarrow$ *a* $\rightarrow$ *t b)* $\rightarrow$ *IntMap a* $\rightarrow$ *t (IntMap b)*. We can use this whenever we need to traverse the whole *IntMap* to update every cell in the list. The obvious use case for this is the regrowing of resources (sugar and spice) in every step.

Unfortunately, measurements quickly reveal that this parallelism makes the performance worse than the sequential - obviously the regrowing of resources is not computation heavy and the parallelism incurs more overhead than the speedup it provides.

Another thing we can parallelise is the computation of the pollution diffusion which uses *map* to compute the new pollution level of each cell. Using *'withStrategy (parList rseq)'* is applied to the list of all cells but the parallelism is too fine-grained: we actually get worse performance than without it - another case for premature optimisation without hard facts profiling.

Thus we end up with the same conclusion as with the monadic SIR implementations: there is practically no opportunity to parallelise the implementation and we refer to the concurrency chapter where we show how to achieve substantial performance improvements when we employ concurrency instead of parallelism.

6.4 Parallel runs

Often one needs to perform a large number of runs of the same simulation. The most prominent use cases for this are:

- Parameter Sweeps / Variations - to explore the parameter space and the dynamics under varying parameter configurations, the same simulation is run with varying parameters and the results recorded for statistical analysis.

[4] In general, folds can be parallelized only when the operation being folded is associative, and then the linear fold can be turned into a tree. Although applicable, we don't follow that approach here and leave it for further research.

- Stochastic replications - due to ABS stochastic nature, running a simulation only once does not allow to generalise or predict overall behaviour - one might have just hit an (un)fortunate special case. To counter this problem, in ABS multiple replications of the simulation are run with same initial model parameters but with different random-number streams. All the results are collected and analysed stochastically (averaged, median,...) from which then more general properties can be derived.

In each case thousands of runs of the same simulation with different model parameters and / or varying random-number streams are needed, requiring a considerable amount of computing power.

Parallelism is a remedy to this problem because in each of these cases individual runs do not interfere with each other and thus can be seen as isolated from each other, like referential transparent, pure computations. Our approaches shown in Part II make this very explicit: the top level functions can always be made pure computations because we are ruling out *IO* and thus even though Monads are employed in many cases, they are still pure. A benefit of our approach is that it is guaranteed at compile time, that individual runs do not interfere with each other and thus there is no danger that parallel runs influence each other.

All this allows to implement parameter sweeps and stochastic replications both through evaluation and data-flow parallelism making another very compelling use case - probably the most striking one - for the use of parallelism in ABS. We hypothesise that data-flow parallelism is better suited for this task because it makes parallelism more explicit as it is indeed a data-flow problem: we pass parameters to single replications which are run and return their results. To apply this we simply run the top level replication logic in the *Par* Monad where replications are run in parallel by forking tasks and results are handed back through *IVars*. If we want the convenience of having a monadic random-number generator within the *Par* Monad, one can use the combined *ParRand* Monad which provides both.

6.5 Discussion

In this chapter we briefly explored how to apply parallelism to our pure functional ABS approach and ran case studies on our existing models to get a rough estimate of what performance increase we can expect. In general, we aimed at running agents in parallel, employing both techniques of evaluation and data-flow parallelism. Because of the quite sequential nature of the agent behaviour itself, there is much less potential for parallelism *within* an agent, thus the idea was to run them all in parallel. This should create enough workload as an agent is an obvious unit of partitioning which can indeed be run in parallel under given circumstances.

Although we showed how to apply the techniques, unfortunately the case studies showed that performance improvement was only possible in the case

of the non-monadic SIR as introduced in Chapter 4.1. The speedup stemmed from the fact that the agents ran indeed in parallel as our original goal was, thus resulting in a significant speedup factor of over 4.

Unfortunately all attempts in parallelising the monadic SIR and Sugarscape implementations failed, which was expected. As soon as we switch to monadic agents, evaluation parallelism is out of the window, as agents can't be run in parallel any more because side effects require to impose a sequential ordering (which is exactly what the idea behind a Monad is).

We further showed how to apply parallelism *within* a SIR agent and for updating the environment of the Sugarscape in parallel using the *Par* Monad. It didn't show any speedup as well but this was not the primary objective: we rather explored conceptually to demonstrate how it can be used - other models might benefit massively from such an approach as they might contain much more potential for data-flow parallelism.

We didn't discuss data parallelism on large array structure or parallelism on GPU as they are used in massively large numerical computation. These techniques achieve tremendous speedups but are not applicable to ABS in general but only in very model specific cases where e.g. each agent needs to crunch through arrays of numbers to perform numerical computations. We refer to [104] for a more in-depth discussion of both topics in Haskell and leave the application to pure functional ABS for further research.

Concluding, we see a direct consequence of the fact that types reflect the semantics of our model: when our agents are pure they can be run in parallel and independent from each other but if they are monadic, then they are not applicable to parallelism. In the next chapter, we show how to approach this problem and come up with a solution where we can run monadic agents in parallel. This is only possible within a concurrent context, where we utilise Software Transactional Memory, which means we have to sacrifice determinism in our solution. Still, by favouring Software Transactional Memory using the *STM* Monad instead of resorting to *IO* we get the guarantee that the the only source of non-determinism is due to the concurrency of *STM and nothing else*. Further, we will show that an additional benefit of using *STM* over *IO* is that the *STM* approach reaches a considerable higher speedup compared to a lock-based approach based on *IO*.

Chapter 7

Concurrent ABS

In an ideal world, we would like to solve all our problems using parallelism but unfortunately, it can't be applied to all parallel problems and ABS is no exception. As soon as there are data dependencies, like we have them in the Sugarscape model in the form of the read/write environment and synchronous agent interactions, and to a lesser extent in the monadic SIR with the *Rand* Monad, we cannot avoid concurrency. More general, this is due to the fact that agents are executed within a monadic context, from which the sequencing of effectful computations immediately follows - this is the very meaning of the Monad abstraction. Indeed, we have shown both by argument and measurement in the previous chapter the very fact that parallelism is simply not applicable to monadic execution of agents due to sequencing of effects, which renders all attempts of running monadic agents in parallel void. In this chapter we discuss the use of concurrency to run agents which have a monadic context in parallel - which is the only way we can execute monadic agents at the same time.

Traditional approaches to concurrency follow a lock-based approach, where sections which access shared data are synchronised through synchronisation primitives like mutexes, semaphores, monitors,... The lock-based path is a well trodden one, with all problems and benefits well established. In this chapter we follow a different path and look into using Software Transactional Memory (STM) for implementing concurrent ABS, which promises to overcome the problems of lock-based approaches. Although STM exists in other languages as well, Haskell was one of the first to natively build it into its core, thus it is a natural choice to follow that direction when already investigating pure functional ABS.

Unfortunately, as soon as we employ concurrency, we lose all static guarantees about reproducibility and the use of STM is no exception. Still, STM has the unique benefit that it can guarantee the lack of persistent side effects at compile time, allowing unproblematic retries of transactions, something of fundamental importance in STM as will be described below. This implies also another *very* compelling advantage of STM over unrestricted lock-based approaches: by using STM, we can reduce the side effects allowed substantially

and guarantee at compile time, that the differences between runs of same initial conditions will only stem from the fact that we run the simulation concurrently - *and from nothing else*. All this makes the use of STM very compelling and to our best knowledge we are the very first to investigate the use of STM for implementing concurrent ABS in a systematic way.

The paper [42] gives a good indication how difficult and complex constructing a correct concurrent program is and shows how much easier, concise and less error-prone an STM implementation is over traditional locking with mutexes and semaphores. More important, it shows that STM consistently outperforms the lock-based implementations. We follow this work and compare the performance of lock-based and STM implementations and hypothesise that the reduced complexity and increased performance will be directly applicable to ABS as well.

We present two case studies using the already introduced SIR (Chapter 2.2.1) and Sugarscape (Chapter 2.2.2) models. We compare the performance of lock-based and STM implementations in each case where we investigate both the scaling performance under increasing number of CPUs and agents. We show that the STM implementations consistently outperform the lock-based ones and scale much better to increasing number of CPUs both on local machines and on Amazon Cloud Services.

7.1 Software Transactional Memory

Software Transactional Memory (STM) was introduced by [137] in 1995 as an alternative to lock-based synchronisation in concurrent programming which, in general, is notoriously difficult to get right. This is because reasoning about the interactions of multiple concurrently running threads and low level operational details of synchronisation primitives is *very hard*. The main problems are:

- Race conditions due to forgotten locks;
- Deadlocks resulting from inconsistent lock ordering;
- Corruption caused by uncaught exceptions;
- Lost wake-ups induced by omitted notifications.

Worse, concurrency does not compose. It is very difficult to write two functions (or methods in an object) acting on concurrent data which can be composed into a larger concurrent behaviour. The reason for it is that one has to know about internal details of locking, which breaks encapsulation and makes composition dependent on knowledge about their implementation. Therefore, it is impossible to compose two functions e.g. where one withdraws some amount of money from an account and the other deposits this amount of money into a different account: one ends up with a temporary state where the money is in

none of either accounts, creating an inconsistency - a potential source for errors because threads can be rescheduled at any time.

STM promises to solve all these problems for a low cost by executing actions *atomically*, where modifications made in such an action are invisible to other threads and changes by other threads are invisible as well until actions are committed - STM actions are atomic and isolated. When an STM action exits, either one of two outcomes happen: if no other thread has modified the same data as the thread running the STM action, then the modifications performed by the action will be committed and become visible to the other threads. If other threads have modified the data then the modifications will be discarded, the action rolled-back and automatically restarted.

7.1.1 STM in Haskell

The work of [67, 68] added STM to Haskell, which was one of the first programming languages to incorporate STM with composable operations into its main core. There exist various implementations of STM in other languages as well (Python, Java, C#, C/C++, etc) but we argue, that it is in Haskell with its type system and the way how side effects are treated where it truly shines.

In the Haskell implementation, STM actions run within the *STM* Monad. This restricts the operations to only STM primitives as shown below, which allows to enforce that STM actions are always repeatable without persistent side effects because such persistent side effects (e.g. writing to a file, launching a missile) are not possible in the *STM* Monad. This is also the fundamental difference to *IO*, where all bets are off because *everything* is possible as there are basically no restrictions because *IO* can run everything.

Thus the ability to *restart* an action without any visible effects is only possible due to the nature of Haskells type system: by restricting the effects to *STM* only, ensures that only controlled effects, which can be rolled back, occur.

STM comes with a number of primitives to share transactional data. Amongst others the most important ones are:

- *TVar* - a transactional variable which can be read and written arbitrarily;
- *TMVar* - a transactional *synchronising* variable which is either empty or full. To read from an empty or write to a full *TMVar* will cause the current thread to block and retry its transaction when *any* transactional primitive of this action has changed.
- *TArray* - a transactional array where each cell is an individual transactional variable *TVar*, allowing much finer-grained transactions instead of e.g. having the whole array in a *TVar*;
- *TChan* - a transactional channel, representing an unbounded FIFO channel, based on a linked list of *TVar*.

Further STM also provides combinators to deal with blocking and composition:

- *retry :: STM ()* - Retries an *STM* action. This will cause to abort the current transaction and block the thread it is running in. When *any* of the transactional data primitives has changed, the action will be run again. This is useful to await the arrival of data in a *TVar* or put more general, to block on arbitrary conditions.

- *orElse :: STM a → STM a → STM a* - Allows to combine two blocking actions where either one is executed but not both. The first actions is run and if it is successful its result is returned. If it retries, then the second is run and if that one is successful its result is returned. If the second one retries, the whole *orElse* retries. This can be used to implement alternatives in blocking conditions, which can be obviously nested arbitrarily.

To run an *STM* action the function *atomically :: STM a → IO a* is provided, which performs a series of *STM* actions atomically within the *IO* Monad. It takes the *STM* action which returns a value of type *a* and returns an *IO* action which returns a value of type *a*. Note that the *IO* action can only be executed from within an *IO* Monad, either within the main thread or an explicitly forked thread.

STM in Haskell is implemented using optimistic synchronisation, which means that instead of locking access to shared data, each thread keeps a transaction log for each read and write to shared data it makes. When the transaction exits, the thread checks whether it has a consistent view to the shared data or not - it checks whether other threads have written to memory it has read and thus it can identify whether a roll-back is required or not.

In the paper [69] the authors use a model of STM to simulate optimistic and pessimistic STM behaviour under various scenarios using the AnyLogic simulation package. They conclude that optimistic STM may lead to 25% less retries of transactions. The authors of [124] analyse several Haskell STM programs with respect to their transactional behaviour. They identified the roll-back rate as one of the key metric which determines the scalability of an application. Although STM might promise better performance, they also warn of the overhead it introduces which could be quite substantial in particular for programs which do not perform much work inside transactions as their commit overhead appears to be high.

7.1.2 An example

We provide a short example to demonstrate the use of STM. To make it more interesting and to show the retry semantics, we use it within a *StateT* transformer where *STM* is the outermost monad. It is important to understand that *STM* does not provide a transformer instance for very good reasons. If it would provide a transformer then we could make *IO* the outermost Monad and perform *IO* actions within *STM*. This would violate the retry semantics as in case of a retry, *STM* is unable to undo the effects of *IO* actions in general. This stems from the fact, that the *IO* type is simply too powerful and we cannot dis-

tinguish between different kinds of *IO* actions in the type, be it simply reading from a file or actually launching a missile. Lets look at the example code:

```
stmAction :: TVar Int -> StateT Int STM Int
stmAction v = do
  -- print a debug output and increment the value in StateT
  Debug.trace "increment!" (modify (+1))
  -- read from the TVar
  n <- lift (readTVar v)
  -- await a condition: content of the TVar >= 42
  if n < 42
    -- condition not met: retry
    then lift retry
    -- condition met: return content ot TVar
    else return n
```

In this example, the *STM* is the outermost Monad in a stack with a *StateT* transformer. When *stmAction* is run, it prints an "increment!" debug message to the console and increments the value in the *StateT* transformer. Then it awaits a condition: as long as *TVar* is less then 42 the action will retry whenever it is run. If the condition is met, it will return the content of the *TVar*. We see the combined effects of using the transformer stack: we have both the *StateT* and the *STM* effects available. The question is how this code behaves if we actually run it. To do this we need to spawn a thread:

```
stmThread :: TVar Int -> IO ()
stmThread v = do
  -- the initial state of the StateT transformer
  let s = 0
  -- run the state transformer with initial value of s (0)
  let ret = runStateT (stmAction v) s
  -- atomically run the STM block
  (a, s') <- atomically ret

  putStrLn("final StateT state     = " ++ show s' ++
           ", STM computation result = " ++ show a)
```

The thread simply runs the *StateT* transformer layer with the initial value of 0 and then the *STM* computation through *atomically* and prints the result to the console. Note that *a* is the result of *stmAction* and *s'* is the final state of the *StateT* computation. To actually run this example we need the main thread to updated the *TVar* until the condition is met within *stmAction*:

```
main :: IO ()
main = do
  -- create a new TVar with initial value of 0
  v <- newTVarIO 0
  -- start the stmThread and pass the TVar
  forkIO (stmThread v)

  forM_ [1..42] (\i -> do
    -- use delay to 'make sure' that a retry is happening for ever increment
    threadDelay 10000
    -- write new value to TVar using atomically
    atomically (writeTVar v i))
```

If we run this program, we will see `'increment!'` printed 43 times, followed by `'final StateT state = 1, STM computation result = 42'`. This clearly demonstrates the retry semantics: *stmAction* is retried 42 times and thus prints `'increment!'` 43 times to the console. The *StateT* computation however is carried out only once and is always rolled back when a retry is happening. The rollback is easily possible in pure functional programming due to persistent data structure: simply throw away the new value and retry with the original value. This example also demonstrates that any *IO* actions which happen within an *STM* action are persistent and can obviously not be rolled back - Debug.trace is an *IO* action masked as pure using *unsafePerformIO*.

7.2 Software Transactional Memory in ABS

In this section we give a short overview of how we apply STM to pure functional ABS. In both case studies we fundamentally follow a time-driven, parallel approach as introduced in Chapter 3.2, where the simulation is advanced by a given Δt and in each step all agents are executed. To employ parallelism, each agent runs within its own thread and agents are executed in lock-step, synchronising between each Δt, which is controlled by the main thread. See Figure 7.1 for a visualisation of our concurrent, time-driven lock-step approach.

By running each agent in a thread will guarantee the execution in parallel even if the agent has a monadic context. This forces us to evaluate each agents monadic context separately instead of running them all in a common context. Note that ultimately we are ending up in the *IO* context because *STM* can be only transacted from within an *IO* context due to non-deterministic side effects. This is no contradiction to our original claim: yes we are running in *IO* but not the agent behaviour itself, which is a fundamental difference.

An agent thread will block until the main thread sends the next Δt and runs the *STM* action atomically with the given Δt. When the *STM* action has been committed, the thread will send the output of the agent action to the main thread to signal it has finished. The main thread awaits the results of all agents to collect them for output of the current step e.g. visualisation or writing to a file.

As will be described in subsequent sections, central to both case studies is an environment which is shared between the agents using a *TVar* or *TArray* primitive, through which the agents communicate concurrently with each other. To get the environment in each step for visualisation purposes, the main thread can access the *TVar* and *TArray* as well.

7.2.1 Adding STM to agents

We briefly discuss how to add STM to agents on a technical level and also show how to run them within their own threads. We use the SIR implementation as example - applying it to the Sugarscape implementation works exactly the same way and is left as a trivial exercise to the reader.

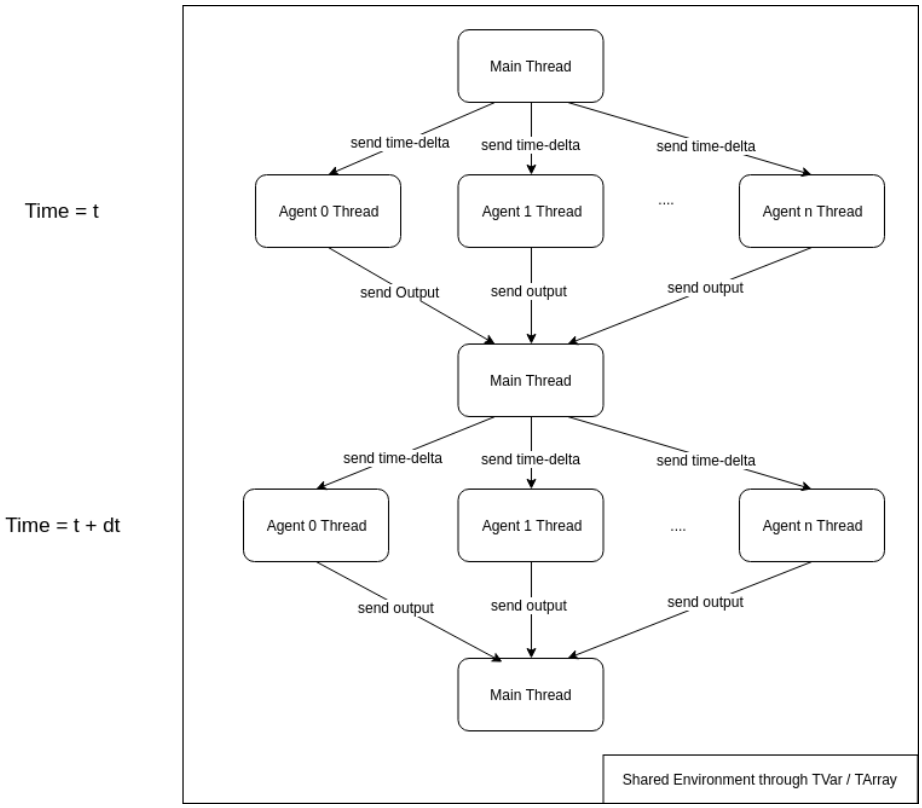


Figure 7.1: Diagram of the parallel time-driven lock-step approach.

The first step is to simply add the *STM* Monad as the *outermost* level to the already the existing transformer stack. Further, the environment is now passed as a transactional data primitive to the agent at *construction time*. Thus the agent does not receive the *SIREnv* as input any more but receives it through currying when constructing its initial MSF. Further, the agent modifies the *SIREnv* directly through the *TVar*, as demonstrated in the case of the infected agent.

```
-- Make Rand a transformer to be able to add STM as innermost monad
type SIRMonad g = RandT g STM
-- Input to agent is now an empty tuple instead of the Environment
type SIRAgent g = SF (SIRMonad g) () SIRState

-- The MSF construction function takes now the TVar with the environment.
sirAgent :: RandomGen g => TVar SIREnv -> Disc2dCoord -> SIRState -> SIRAgent g

-- The infected agent behaviour is nearly the same except that
-- the agent modifies the environment through the TVar
infected :: RandomGen g => SF (SIRMonad g) () (SIRState, Event ())
infected = proc _ -> do
  recovered <- occasionally illnessDuration () -< ()
  if isEvent recovered
    then (do
      -- update the environment through the TVar
      arrM_ (lift $ lift $ modifyTVar env (changeCell coord Recovered)) -< ()
      returnA -< (Recovered, Event ()))
    else returnA -< (Infected, NoEvent)
```

The agent thread is straightforward: it takes *MVar* synchronisation primitives to synchronise with the main thread and simply runs the agent behaviour each time it receives the next *DTime*:

```
agentThread :: RandomGen g
            => Int              -- ^ Number of steps to compute
            -> SIRAgent g       -- ^ Agent behaviour MSF
            -> g                -- ^ Random-number generator of the agent
            -> MVar SIRState    -- ^ Synchronisation back to main thread
            -> MVar DTime       -- ^ Receiving DTime for next step
            -> IO ()
agentThread 0 _ _ _ _ = return () -- all steps computed, terminate thread
agentThread n sf rng retVar dtVar = do
  -- wait for dt to compute current step
  dt <- takeMVar dtVar

  -- compute output of current step
  let sfReader = unMSF sf ()
      sfRand   = runReaderT sfReader dt
      sfSTM    = runRandT sfRand rng
  -- run the STM action atomically within IO
  ((ret, sf'), rng') <- atomically sfSTM

  -- post result to main thread
  putMVar retVar ret

  -- tail-recursion to next step
  agentThread (n - 1) sf' rng retVar dtVar
```

Computing a simulation step is now trivial within the main thread: all agent threads *MVars* are signalled to unblock followed by an immediate block on the *MVars* into which the agent threads post back their result. The state of the current step is then extracted from the environment, which is stored within the *TVar* which the agent threads have updated.

```
simulationStep :: TVar SIREnv      -- ^ environment
               -> [MVar DTime]     -- ^ sync dt to threads
               -> [MVar SIRState] -- ^ sync output from threads
               -> DTime            -- ^ time-delta
               -> IO SIREnv
simulationStep env dtVars retVars dt = do
  -- tell all threads to continue with the corresponding DTime
  mapM_ (`putMVar` dt) dtVars
  -- wait for results but ignore them, SIREnv contains all states
  mapM_ takeMVar retVars
  -- return state of environment when step has finished
  readTVarIO env
```

The difference to an implementation which uses *IO* are minor but far reaching. Instead of using *STM* as outermost Monad, we use *IO*, thus running the whole agent behaviour within the *IO* Monad. Instead of receiving the environment through a *TVar*, the agent receives it through an *IORef*. It also receives an *MVar* which is the synchronisation primitive to synchronise the access to the environment in the *IORef* amongst all agents. Agents grab and release the synchronisation lock of the *MVar* when they enter and leave a critical section in which they operate on the environment stored in the *IORef*.

7.3 Case study I: SIR

Our first case study is the SIR model as introduced in Chapter 2.2.1. The aim of this case study is to investigate the potential speedup a concurrent *STM* implementation gains over a sequential one under varying number of CPU cores and agents. The behaviour of the agents is quite simple and the interactions are happening indirectly through the environment, where reads from the environment outnumber the writes to it by far. Further, a comparison to a lock-based implementation with the *IO* Monad is done to understand that *STM* is also able to outperform traditional concurrency, *in a pure functional ABS setting* while still retaining its greater static guarantees than *IO* [1].

1. Sequential - this is the original implementation as discussed in Chapter 4.3, where the discrete 2D environment is shared amongst all agents as read-only data and the agents are executed sequentially within the main thread without any concurrency.

[1] The code of all three implementations is available at `https://github.com/thalerjonathan/phd/tree/master/public/stmabs/code/SIR`

OS	Fedora 28, 64-bit
RAM	16 GByte
CPU	Intel i5-4670K @ 3.4GHz
HD	250Gbyte SSD
Haskell	GHC 8.2.2

Table 7.1: Machine and Software specs for all experiments

	Cores	Duration
Sequential	1	72.5
Lock-Based	1	60.6
	2	42.8
	3	38.6
	4	41.6
STM	1	53.2
	2	27.8
	3	21.8
	4	**20.8**

Table 7.2: Experiments on 51x51 (2,601 agents) grid with varying number of cores. Timings in seconds (lower is better).

2. STM - this is the same implementation as the *Sequential* one but agents run now in the *STM* Monad and have access to the discrete 2D environment through a transactional variable *TVar*. This means that the agents now communicate indirectly by reads and writes through the *TVar*.

3. Lock-Based - this follows the *STM* implementation, with the agents running in *IO*. They share the discrete 2D environment using an *IORef* and have access to an *MVar* lock to synchronise access to it.

Each experiment was run until $t = 100$ and stepped using $\Delta t = 0.1$. For each experiment we conducted 8 runs on our machine (see Table 7.1) under no additional work-load and report the mean. In the experiments we varied the number of agents (grid size) as well as the number of cores when running concurrently - the numbers are always indicated clearly.

7.3.1 Constant grid size, varying cores

In this experiment we held the grid size constant to 51 x 51 (2,601 agents) and varied the cores. The results are reported in Table 7.2.

The *STM* implementation running on 4 cores shows a speedup factor of 3.6 over *Sequential*, which is a quite impressive number when considering that we can achieve at most a factor of 4 when running on 4 cores. It seems that *STM* allow us to push the practical limit very close to the theoretical one, whereas the *Lock-Based* approach just arrives at a factor of 1.74 on 4 cores.

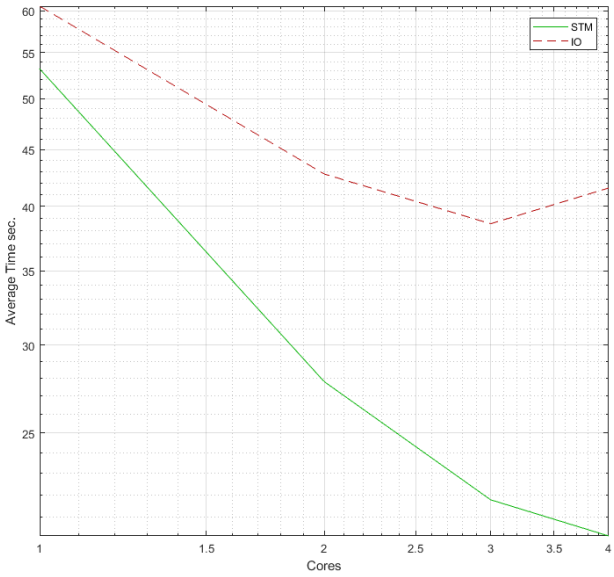


Figure 7.2: Comparison of performance and scaling on multiple cores of STM and Lock-Based. Note that the Lock-Based implementation seems to perform slightly worse on 4 than on 3 cores probably due to lock-contention.

Comparing the performance and scaling to multiple cores shows that the *STM* implementation significantly outperforms the *Lock-Based* one and scales better to multiple cores. The *Lock-Based* implementation performs best with 3 cores and shows slightly worse performance on 4 cores as can be seen in Figure 7.2. This is no surprise because the more cores are running at the same time, the more contention for the lock, thus the more likely synchronisation happening, resulting in higher potential for reduced performance. This is not an issue in *STM* because no locks are taken in advance.

7.3.2 Varying grid size, constant cores

In this experiment we varied the grid size and used always 4 cores. The results are reported in Table 7.3 and plotted in Figure 7.3.

It is clear that the *STM* implementation outperforms the *Lock-Based* implementation by a substantial factor.

7.3.3 Retries

Of very much interest when using STM is the retry-ratio, which obviously depends highly on the read-write patterns of the respective model. We used the

Grid-Size	STM	Lock-Based	Ratio
51 x 51 (2,601)	**20.2**	41.9	2.1
101 x 101 (10,201)	**74.5**	170.5	2.3
151 x 151 (22,801)	**168.5**	376.9	2.2
201 x 201 (40,401)	**302.4**	672.0	2.2
251 x 251 (63,001)	**495.7**	1,027.3	2.1

Table 7.3: Performance on varying grid sizes. Timings in seconds (lower is better). Ratio compares STM to Lock-Based.

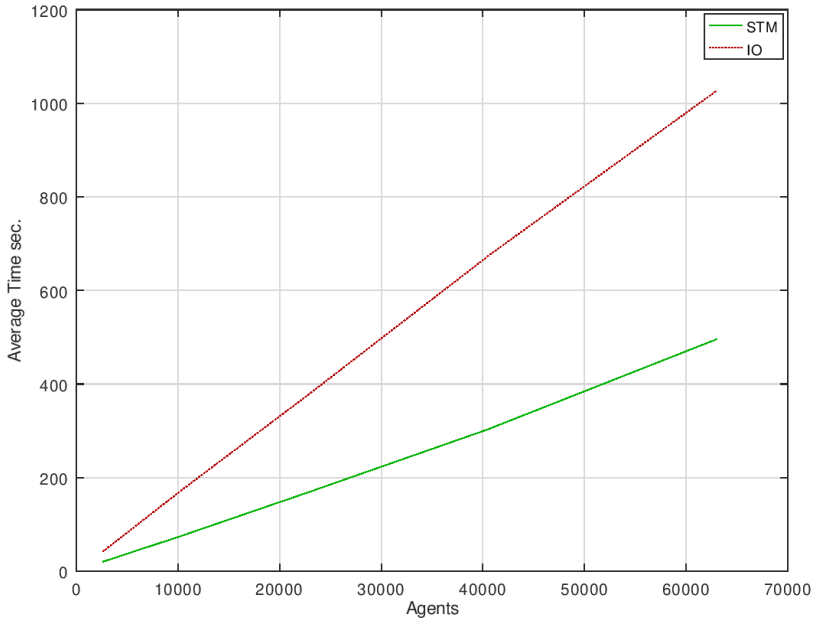


Figure 7.3: Performance on varying grid sizes.

Grid-Size	Commits	Retries	Ratio
51 x 51 (2,601)	2,601,000	1306.5	0.0
101 x 101 (10,201)	10,201,000	3712.5	0.0
151 x 151 (22,801)	22,801,000	8189.5	0.0
201 x 201 (40,401)	40,401,000	13285.0	0.0
251 x 251 (63,001)	63,001,000	21217.0	0.0

Table 7.4: Retry ratios on varying grid sizes on 4 cores.

	Cores	51x51	251x251
Lock-Based	16	72.5	1830.5
	32	73.1	1882.2
STM	16	**8.6**	**237.0**
	32	12.0	248.7

Table 7.5: Performance on varying cores on Amazon S2 Services. Timings in seconds (lower is better).

stm-stats library to record statistics of commits, retries and the ratio. The results are reported in Table 7.4.

Independent of the number of agents we always have a retry-ratio of 0.0. This indicates that this model is *very* well suited to STM, which is also directly reflected in the much better performance over the *Lock-Based* implementation. Obviously this ratio stems from the fact that in our implementation we have *very* few writes, which happen only in case when an agent changes from Susceptible to Infected or from Infected to Recovered. On the other hand, there are a very large number of reads due to indirect agent interaction. For *STM* this is no problem because no lock is taken but the *Lock-Based* approach is forced to conservatively take the lock to ensure mutual exclusive access to the critical section across all agents.

7.3.4 Going large scale

To test how far we can scale up the number of cores in both the *Lock-Based* and *STM* cases, we ran two experiments, 51x51 and 251x251, on Amazon EC2 instances with a larger number of cores than our local machinery, starting with 16 and 32 to see if we are running into decreasing returns. The results are reported in Table 7.5.

As expected, the *Lock-Based* approach doesn't scale up to many cores because each additional core brings more contention to the lock, resulting in an even more decreased performance, even worse than the *Sequential* implementation. This is particularly obvious in the 251x251 experiment because of the much larger number of concurrent agents. The *STM* approach returns better performance on 16 cores but fails to scale further up to 32 where the perfor-

mance drops below the one with 16 cores. In both STM cases we measured a retry-ratio of 0, thus we assume that with 32 cores we become limited by the overhead of STM transactions [124] because the workload of an STM action in our SIR implementation is quite small.

Compared to the *Sequential* implementation, *STM* reaches a speedup factor of 8.4 on 16 cores, which is still impressive but is much further away from the theoretical limit than in the case of only 4 cores - a further indication that this model in particular and our approach in general does not scale up arbitrarily.

7.3.5 Discussion

The timing measurements speak a clear language: running in *STM* and sharing state using a transactional variable *TVar* is much more time-efficient than both the *Sequential* and *Lock-Based* approach. On 4 cores *STM* achieves a speedup factor of 3.6, nearly reaching the theoretical limit. Obviously both *STM* and *Lock-Based* sacrifices determinism: repeated runs might not lead to same dynamics despite same initial conditions. Still, by sticking to *STM*, we get the guarantee that the source of this non-determinism is concurrency within the *STM* monad but *nothing else*. This we can not guarantee in the case of the *Lock-Based* approach as all bets are off when running within *IO*. The fact to have *both* the better performance *and* the stronger static guarantees in the *STM* approach makes it *very* compelling.

7.4 Case study II: Sugarscape

The second case study is the Sugarscape model as introduced in Chapter 2.2.2. In this case study we look into the potential performance improvement in a model with much more complex agent behaviour and dramatically increased writes on the shared environment.

We implemented the *Carrying Capacity* (p. 30) section of Chapter II of the Sugarscape book [47]. In each step agents search (move to) the cell with the highest sugar they see within their vision, harvest all of it from the environment and consume sugar because of their metabolism. Sugar regrows in the environment over time. Only one agent can occupy a cell at a time. Agents don't age and cannot die from age. If agents run out of sugar due to their metabolism, they die from starvation and are removed from the simulation. The authors report that the initial number of agents quickly drops and stabilises around a level depending on the model parameters. This is in accordance with our results as we show in Appendix B and guarantees that we don't run out of agents. The model parameters are as follows:

- Sugar Endowment: each agent has an initial sugar endowment randomly uniform distributed between 5 and 25 units;
- Sugar Metabolism: each agent has a sugar metabolism randomly uniform distributed between 1 and 5;

- Agent Vision: each agent has a vision randomly uniform distributed between 1 and 6, same for each of the 4 directions (N, W, S, E);
- Sugar Growback: sugar grows back by 1 unit per step until the maximum capacity of a cell is reached;
- Agent Population: initially 500 agents;
- Environment Size: 50 x 50 cells with toroid boundaries which wrap around in both x and y dimension.

Note that in this implementation (as in the full Chapter II of the book), no direct and no synchronous agent interactions as we implemented them in Chapter 5 are happening. As in the SIR example, all agents interact with each other indirectly through the shared environment. This allows us to regard the implementation as a time-driven, parallel one wherein each step agents act conceptually at the same time.

We compare four different implementations [2]:

1. Sequential - All agents are run after another (including the environment) and the environment is shared amongst the agents using a *StateT* transformer.
2. Lock-Based - All agents are run concurrently in the *IO* monad and the environment is shared between the agents, using an *IORef* with the access synchronised through an *MVar* lock.
3. STM TVar - All agents are run concurrently in the *STM* monad and the environment is shared using a *TVar* between the agents.
4. STM TArray - All agents are run concurrently in the *STM* monad and the environment is shared using a *TArray* between the agents.

We follow [99] and measure the average number of steps per second of the simulation over 60 seconds. For each experiment we conducted 8 runs on our machine (see Table 7.1) under no additional work-load and report the average. In the experiments we varied the number of cores and agents when running concurrently - the numbers are always indicated clearly.

Ordering The model specification requires to shuffle agents before every step ([47], footnote 12 on page 26). In the *Sequential* approach we do this explicitly but in the *Lock-Based* and both *STM* approaches we assume this to happen automatically due to race-conditions in concurrency, thus we arrive at an effectively shuffled processing of agents: we implicitly assume that the order of the agents is *effectively* random in every step. The important difference between the two approaches is that in the *Sequential* approach we have full control over

[2] The code is freely available at `https://github.com/thalerjonathan/phd/tree/master/public/stmabs/code/SugarScape`

	Cores	Steps	Retries
Sequential	1	39.4	N/A
Lock-Based	1	43.0	N/A
	2	51.8	N/A
	3	57.4	N/A
	4	58.1	N/A
STM *TVar*	1	**47.3**	0.0
	2	53.5	1.1
	3	57.1	2.2
	4	53.0	3.2
STM *TArray*	1	45.4	0.0
	2	**65.3**	0.02
	3	**75.7**	0.04
	4	**84.4**	0.05

Table 7.6: Steps per second (higher is better) and retries on 50x50 grid with 500 initial agents on varying cores.

this randomness but in the *STM* not - also this means that repeated runs with the same initial conditions might lead to slightly different results. This decision leaves the execution order of the agents ultimately to Haskell's Runtime System and the underlying OS. We are aware that by doing this, we make assumptions that the threads run uniformly distributed (fair) but such assumptions should not be made in concurrent programming. As a result we can expect this fact to produces non-uniform distributions of agent runs but we assumed that for this model this does not has a significance influence - in case of doubt, we could resort to shuffling the agents before running them in every step. We agree that this very problem would deserve in-depth research on its own, where also the influence of non-deterministic ordering on the correctness and results of ABS has to be analysed. We introduce techniques allowing to perform such analyses in the next Part on property-based testing but leave it for further research as this issue is beyond the focus of this thesis.

7.4.1 Constant agent population

In a first approach we compare the performance of all implementations on varying numbers of cores. The results are reported in Table 7.6 and plotted in Figure 7.4.

As expected, the *Sequential* implementation is the slowest, followed by the *Lock-Based* and *TVar* approach whereas *TArray* is the best performing one.

We clearly see that using a *TVar* to share the environment is a very inefficient choice in this model: *every* write to a cell leads to a retry independent whether the reading agent reads that changed cell or not, because the data structure can not distinguish between individual cells. By using a *TArray* we can avoid

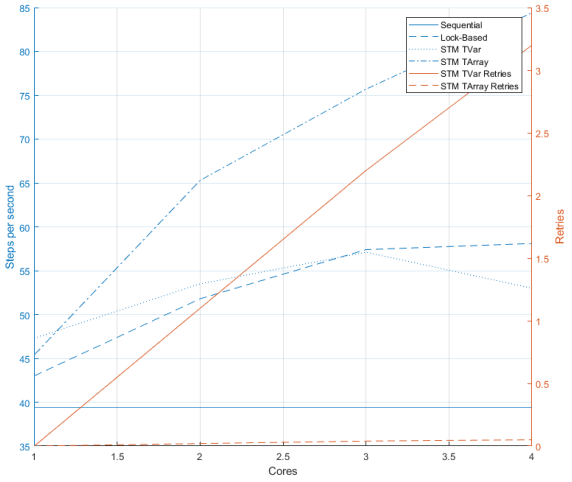


Figure 7.4: Steps per second (higher is better) and retries on 50x50 grid and 500 initial agents on varying cores.

the situation where a write to a cell in a far distant location of the environment will lead to a retry of an agent which never even touched that cell. Also the *TArray* seems to scale up by 10 steps per second for every core added. It will be interesting to see how far this could go with the Amazon experiment, as we seem not to hit a limit with 4 cores yet.

The inefficiency of *TVar* is also reflected in the nearly similar performance of the *Lock-Based* implementation which even outperforms it on 4 cores. This is due to very similar approaches because both operate on the whole environment instead of only the cells as *TArray* does. This seems to be a bottleneck in *TVar* reaching the best performance on 3 cores, which then drops on 4 cores due to an increasing retries ratio. The *Lock-Based* approach seems to reduce its returns on increased number of cores, hitting a limit with 4 cores as well.

7.4.2 Scaling up agent population

So far we kept the initial number of agents at 500, which due to the model specification, quickly drops and stabilises around 200 due to the carrying capacity of the environment as described in the book [47] section *Carrying Capacity* (p. 30).

We now want to measure the performance of our approaches under increased number of agents. For this we slightly change the implementation: always when an agent dies it spawns a new one which is inspired by the ageing and birthing feature of Chapter III in the book [47]. This ensures that we keep the number of agents roughly constant (still fluctuates but doesn't drop to

Agents	Sequential	Lock-Based	TVar (3 cores)	TVar (4 cores)	TArray
500	14.4	20.2	20.1	18.5	**71.9**
1,000	6.8	10.8	10.4	9.5	**54.8**
1,500	4.7	8.1	7.9	7.3	**44.1**
2,000	4.4	7.6	7.4	6.7	**37.0**
2,500	5.3	5.4	9.2	8.9	**33.3**

Table 7.7: Steps per second (higher is better) on 50x50 grid with varying number of agents with 4 (and 3) cores except Sequential (1 core).

low levels) over the whole duration. This ensures a constant load of concurrent agents interacting with each other and demonstrates also the ability to terminate and create concurrent agents (threads) dynamically during the simulation.

Except for the *Sequential* approach we ran all experiments with 4 cores (TVar with 3 as well). We looked into the performance of 500, 1,000, 1,500, 2,000 and 2,500 (maximum possible capacity of the 50x50 environment). The results are reported in Table 7.7 and plotted in Figure 7.5.

As expected, the *TArray* implementation outperforms all others substantially. Also as expected, the *TVar* implementation on 3 cores is faster than on 4 cores as well when scaling up to more agents. The *Lock-Based* approach performs about the same as the *TVar* on 3 cores because of the very similar approaches: both access the *whole* environment. Still the *TVar* approach uses one core less to arrive at the same performance, thus strictly speaking outperforming the *Lock-Based* implementation.

What seems to be very surprising is that in the *Sequential* and *TVar* cases the performance with 2,500 agents is *better* than the one with 2,000 agents. The reason for this is that in the case of 2,500 agents, an agent can't move anywhere because all cells are already occupied. In this case the agent will not rank the cells in order of their pay-off (max sugar) to move to but just stays where it is. We hypothesise that due to Haskells laziness the agents actually never look at the content of the cells in this case but only the number which means that the cells themselves are never evaluated which further increases performance. This leads to the better performance in case of *Sequential* and *TVar* because both exploit laziness. In the case of the *Lock-Based* approach we still arrive at a lower performance because the limiting factor are the unconditional locks. In the case of the *TArray* approach we also arrive at a lower performance because it seems that STM perform reads on the neighbouring cells which are not subject to lazy evaluation. In Haskell it is notoriously difficult to reason about efficiency and this behaviour of improved performance due to Haskells lazyness is no exception. We leave an in-depth investigation for further research as it is beyond the focus of this thesis.

We also measured the average retries both for *TVar* and *TArray* under 2,500 agents where the *TArray* approach shows best scaling performance with 0.01 retries whereas *TVar* averages at 3.28 retries. Again this can be attributed to

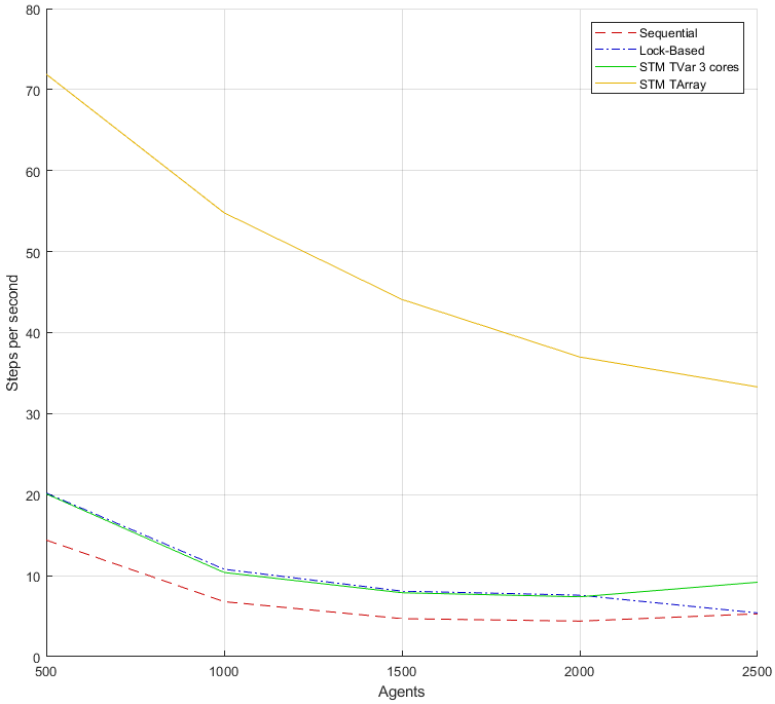


Figure 7.5: Steps per second (higher is better) on 50x50 grid and varying number of agents with 4 (and 3) cores except Sequential (1 core).

	Cores	Carrying Capacity	Rebirthing
Lock-Based	16	53.9	4.4
	32	44.2	3.6
STM TArray	16	**116.8** (0.23)	**39.5** (0.08)
	32	**109.8** (0.41)	**31.3** (0.18)

Table 7.8: Steps per second (higher is better) on varying cores on Amazon S2 Services.

the better transactional data structure which reduces retry-ratio substantially to near-zero levels.

7.4.3 Going large scale

To test how far we can scale up the number of cores in both the *Lock-Based* and *TArray* cases, we ran the two experiments (carrying capacity and rebirthing) on Amazon EC2 instances with increasing number of cores starting with 16 and 32 to see if we run into decreasing returns. The results are reported in Table 7.8.

As expected, the *Lock-Based* approach doesn't scale up to many cores because each additional core brings more contention to the lock, resulting in even more decreased performance. This is particularly obvious in the rebirthing experiment because of the much larger number of concurrent agents. The *TArray* approach returns better performance on 16 cores but fails to scale further up to 32 where the performance drops below the one with 16 cores. We indicated the retry-ratio in brackets and see that they roughly double from 16 to 32, which is the reason why performance drops as at this point.

7.4.4 Comparison with other approaches

The paper [99] reports a performance of 17 steps in RePast, 18 steps in MASON (both non-parallel) and 2,000 steps per second on a GPU on a 128x128 grid. Although our *Sequential* implementation, which runs non-parallel as well, outperforms the RePast and MASON implementations of [99], one must be very well aware that these results were generated in 2008, on current hardware of that time.

The very high performance on the GPU does not concern us here as it follows a very different approach than ours. We focus on speeding up implementations on the CPU as directly as possible without locking overhead. When following a GPU approach one needs to map the model to the GPU which is a delicate and non-trivial matter. With our approach we show that speedup with concurrency is very possible without the low-level locking details or the need to map to GPU. Also some features as bilateral trading between agents, where a pair of agents needs to come to a conclusion over multiple synchronous steps, is difficult to implement on a GPU whereas this should be easier using STM.

Note that we kept the grid-size constant because we implemented the environment as a single agent which works sequentially on the cells to regrow the sugar. Obviously this doesn't really scale up on parallel hardware and experiments show that the performance goes down dramatically when we increase the environment to 128x128 with same number of agents which is the result of Amdahl's law where the environment becomes the limiting factor of the simulation. Depending on the underlying data structure used for the environment we have two options to solve this problem. In the case of the *Sequential* and *TVar* implementation we build on an indexed array, which we can be updated in parallel using the existing data-parallel support in Haskell. In the case of the *TArray* approach we have no option but to run the update of every cell within its own thread. We leave both for further research as it is out of scope of this thesis.

7.4.5 Discussion

This case study showed clearly that besides being substantially faster than the *Sequential* implementation, *STM* is also able to perform considerably better than a *Lock-Based* approach even in the case of a model with much higher complexity in agent behaviour and dramatically increased number of writes to the environment.

Further, this case study demonstrated that the selection of the right transactional data structure is of fundamental importance when using *STM*. Selecting the right transactional data structure is highly model-specific and can lead to dramatically different performance results.

In this case the *TArray* performed best due to many writes but in the SIR case study a *TVar* showed good enough results due to the very low number of writes. When not carefully selecting the right transactional data structure which supports fine-grained concurrency, a lock-based implementation might perform as well or even outperform the STM approach as can be seen when using the *TVar*.

Although the *TArray* is the better transactional data structure overall, it might come with an overhead, performing worse on low number of cores than a *TVar* approach but has the benefit of quickly scaling up to multiple cores. Depending on the transactional data structure, scaling up to multiple cores hits a limit at some point. In the case of the *TVar* the best performance is reached with 3 cores. With the *TArray* we reached this limit around 16 cores.

Note that the comparison between the *Lock-Based* approach and the *STM TArray* implementation is a bit unfair due to a very different locking structure. A more suitable comparison would have been to use an indexed Array with a tuple of (MVar, IORef) in each cell to support fine-grained locking on cell-level. This would be a more just comparison to the *STM Array* where fine-grained transactions happen on the cell-level. We hypothesise that *STM* will still outperform the *IO* approach but to a lesser degree - we leave the proof of this for further research.

7.5 Discussion

In this chapter we have shown how to apply concurrency to monadic ABS to gain substantially speedup. We developed a novel approach, using STM, which to our best knowledge has not been discussed systematically in the context of ABS so far. This new approach outperforms a traditional lock-based implementation running in the *IO* Monad *and* guarantees that the differences between runs with same initial conditions stem from the non-determinism within *STM but nothing else*. The latter point can't be guaranteed by the lock-based approach as it runs in the *IO* Monad, which allows literally anything from reading from a file, to launching a missile. Further, with STM, concurrency becomes more a control-flow oriented concern, so using STM allows us to treat the concurrent problem within an agent as a data-flow oriented one without cluttering the model code with operational details of concurrency. This gives strong evidence that STM should be favoured over lock-based approaches in general for implementing concurrent ABS in Haskell.

Note that in this chapter we assumed that concurrent execution has no qualitative influence on the dynamics: although repeated runs with same initial conditions (might) lead to different results due to non-determinism, the dynamics follow still the same distribution as the one from the sequential implementation. To verify this we can make use the techniques of property-based testing as shown in Chapter 9 but we leave it for further research.

The next step would be to add synchronous agent interactions as they occur in the Sugarscape use cases of mating, trading and lending. We have started to do work on this already and could implement also one-directional agent interactions as they occur in the disease transmission, payback of loans and notification of inheritance upon the death of a parent agent. We use the *TQueue* primitive to emulate the behaviour of mailboxes through which agents can post events to each other. The result is promising but needs more investigation. We have also started looking into synchronous agent interactions using STM which is a lot trickier and is very susceptible to dead-locks (which are still possible in STM!). We have yet to prove how to implement reliable synchronous agent interactions without deadlocks in *STM*. It might be very well the case that a truly concurrent approach is doomed due to the following [104] (Chapter 10. Software Transactional Memory, "What Can We Not Do with STM?"): *"In general, the class of operations that STM cannot express are those that involve multi-way communication between threads. The simplest example is a synchronous channel, in which both the reader and the writer must be present simultaneously for the operation to go ahead. We cannot implement this in STM, at least compositionally [..]: the operations need to block and have a visible effect — advertise that there is a blocked thread — simultaneously."*.

Further, *STM* is not fair because *all* threads, which block on a transactional primitive, have to be woken up, thus a FIFO guarantee cannot be given. We hypothesise that for most models, where the *STM* approach is applicable, this has no qualitative influence on the dynamics as agents are assumed to act con-

ceptually at the same time and no fairness is needed. We leave the test of this hypothesis for future research.

We didn't look into applying distributed computation to our approach. One direction to follow would be to use the *Cloud Haskell* library, which is very similar to the concurrency model in Erlang. We leave this for further research as it is beyond the scope of this thesis.

Part IV:

Property-based testing

When implementing an Agent-Based Simulation (ABS) it is of fundamental importance that the implementation is correct up to some specification and that this specification matches the real world in some way. This process is called verification and validation (V&V), where *validation* is the process of ensuring that a model or specification is sufficiently accurate for the purpose at hand whereas *verification* is the process of ensuring that the model design has been transformed into a computer model with sufficient accuracy [132]. In other words, validation determines if we are we building the *right model*, and verification if we are building the *model right* up to some specification [12].

One can argue that ABS should require more rigorous programming standards than other computer simulations [126]. Due to the fact that researchers in ABS are looking for an emergent behaviour in the dynamics of the simulation, they are always tempted to look for surprising behaviour and expect something unexpected from their simulation. Also, due to ABS' *constructive / exploratory* nature [45, 46], there exists some uncertainty about the dynamics the simulation will produce before running it. The authors [117] see the current process of building ABS as a discovery process, where models of an ABS often lack an analytical solution in general, which makes verification much harder if there is no such solution. Thus it is often very difficult to judge whether an unexpected outcome can be attributed to the model or has in fact its roots in a subtle programming error [55].

In general this implies that it is not possible to prove that a model is valid in general but that the best we can do is to *raise the confidence* in the correctness of the simulation. Therefore, the process of V&V is not the proof that a model is correct but it is the *process* of trying to show that the model is *not incorrect*. The more checks one carries out which show that it is not incorrect, the more confidence we can place in the models validity. To tackle such a problem in software, engineers have developed the concept of test-driven development (TDD).

TDD was popularised in the early 00s by Kent Beck [14] as a way to a more agile approach to software engineering, where instead of doing each step (requirements, implementation, testing,...) as separated from each other, all of them are combined in shorter cycles. Put shortly, in TDD tests are written for each feature before actually implementing it, then the feature is fully implemented and the tests for it should pass. This cycle is repeated until the implementation of all requirements has finished. Traditionally TDD relies on so called unit tests which can be understood as a piece of code which when run isolated, tests some functionality of an implementation. Thus we can say that test-driven development in general and unit testing together with some measure of code-coverage in particular, guarantee the correctness of an implementation up to some informal degree, which has been proven to be sufficiently enough through years of practice in the software industry all over the world.

Related work

The work [33] was the first to discuss how to apply TDD to ABS, using unit testing to verify the correctness of the implementation up to a certain level. They show how to implement unit tests within the RePast Framework [112] and make the important point that such a software needs to be designed to be sufficiently modular otherwise testing becomes too cumbersome and involves too many parts. The paper [6] discusses a similar approach to DES in the AnyLogic software toolkit.

The paper [116] proposes Test Driven Simulation Modelling (TDSM) which combines techniques from TDD to simulation modelling. The authors present a case study for maritime search-operations where they employ ABS. They emphasise that simulation modelling is an iterative process, where changes are made to existing parts, making a TDD approach to simulation modelling a good match. They present how to validate their model against analytical solutions from theory using unit tests by running the whole simulation within a unit test and then perform a statistical comparison against a formal specification.

The paper [64] gives an in-depth and detailed overview over verification, validation and testing of agent-based models and simulations and proposes a generic framework for it. The authors present a generic UML class model for their framework which they then implement in the two ABS frameworks RePast and MASON. Both of them are implemented in Java and the authors provide a detailed description how their generic testing framework architecture works and how it utilises unit testing with JUnit to run automated tests. To demonstrate their framework they provide also a case study of an agent-base simulation of synaptic connectivity where they provide an in-depth explanation of their levels of test together with code.

Towards property-based testing

According to [41], unit testing in Haskell is quite common and robust but generally speaking it tends to be of less importance in Haskell since the type system makes an enormous amount of invalid programs completely inexpressible by construction. Unit tests tend to be written later in the development lifecycle and generally tend to be about the core logic of the program and not the intermediate plumbing [41]. Although it would be interesting to see how we can apply unit testing to our approach, it is straightforward, nothing new and does not constitute unique research.

Thus, in this chapter we introduce an additional technique for TDD: *property-based testing*, which can be seen complementary to unit testing. Property-based testing has its origins in Haskell [29, 30, 133], where it was first conceived and implemented. It has been successfully used for testing Haskell code for years and also been proven to be useful in the industry [81]. We show and discuss how this technique can be applied to test pure functional ABS implementations.

To our best knowledge property-based testing has never been looked at in the context of ABS and this thesis is the first one to do so.

The main idea of property-based testing is to express model specifications and laws directly in code and test them through *automated* and *randomised* test data generation. Thus one hypothesis of this thesis is that due to ABS *stochastic* and *exploratory / generative / constructive* nature, property-based testing is a natural fit for testing ABS in general and pure functional ABS implementations in particular. It thus should pose a valuable addition to the already existing testing methods in this field, worth exploring.

To substantiate and test our hypothesis, we conducted a few case studies. First, we look into how to express and test agent specifications for both the time- and event-driven SIR implementations in Chapter 8. Then we show how to encode model invariants of the SIR implementation and validate it against the formal specification from SD using property-tests in Chapter 9. Note that we explicitly exclude obvious applications of property-testing like boundary-checks of the environment, helper functions of agents,... as although they are used within ABS, there is nothing new in testing them.

Property-based testing has a close connection to model-checking [106], where properties of a system are proved in a formal way. The important difference is that the checking happens directly on code and not on the abstract, formal model, thus one can say that it combines model checking and unit testing, embedding it directly in the software development and TDD process without an intermediary step. We hypothesise that adding it to the already existing testing methods in the field of ABS is of substantial value as it allows to cover a much wider range of test cases due to automatic data generation. This can be used in two ways: to verify an implementation against a formal specification and to test hypotheses about an implemented simulation. This puts property-based testing on the same level as agent- and system testing, where not technical implementation details of e.g. agents are checked like in unit tests but their individual complete behaviour and the system behaviour as a whole.

The work [116] explicitly mentions the problem of test coverage which would often require to write a large number of tests manually to cover the parameter ranges sufficiently enough - property-based testing addresses exactly this problem by *automating* the test data generation. Note that this is closely related to data generators [64], load generators and random testing [27]. Property-based testing though goes one step further by integrating this into a specification language directly into code, emphasising a declarative approach and pushing the generators behind the scenes, making them transparent and focusing on the specification rather than on the data generation.

Property-based testing

Property-based testing allows to formulate *functional specifications* in code which then a property-based testing library tries to falsify by *automatically*

generating test data, covering as much cases as possible. When a case is found for which the property fails, the library then reduces the test data to its simplest form for which the test still fails e.g. shrinking a list to a smaller size. It is clear to see that this kind of testing is especially suited to ABS, because we can formulate specifications, meaning we describe *what* to test instead of *how* to test. Also the deductive nature of falsification in property-based testing suits very well the constructive and exploratory nature of ABS. Further, the automatic test-generation can make testing of large scenarios in ABS feasible because it does not require the programmer to specify all test cases by hand, as is required in e.g. traditional unit tests.

Property-based testing was introduced in [29, 30] where the authors present the QuickCheck library in Haskell, which tries to falsify the specifications by *randomly* sampling the test space. According to the authors of QuickCheck *"The major limitation is that there is no measurement of test coverage."* [29]. Although QuickCheck provides help to report the distribution of test cases it is not able to measure the coverage of tests in general. This could lead to the case that test cases which would fail are never tested because of the stochastic nature of QuickCheck. Fortunately, the library provides mechanisms for the developer to measure coverage in specific test cases where the data and its (expected) distribution is known to the developer. This is a powerful tool for testing randomness in ABS as will be shown in subsequent chapters.

As a remedy for the potential coverage problems of QuickCheck, there exists also a deterministic property-testing library called SmallCheck [133], which instead of randomly sampling the test-space, enumerates test cases exhaustively up to some depth. It is based on two observations, derived from model-checking, that (1) *"If a program fails to meet its specification in some cases, it almost always fails in some simple case"* and (2) *"If a program does not fail in any simple case, it hardly ever fails in any case* [133]. This non-stochastic approach to property-based testing might be a complementary addition in some cases where the tests are of non-stochastic nature with a search space too large to test manually by unit testing but small enough to enumerate exhaustively. The main difficulty and weakness of using SmallCheck is to reduce the dimensionality of the test case depth search to prevent combinatorial explosion, which would lead to exponential number of cases. Thus one can see QuickCheck and SmallCheck as complementary instead of in opposition to each other.

A brief overview of QuickCheck

To give a good understanding of how property-based testing works with QuickCheck, we give a few examples of property-tests on lists, which are directly expressed as functions in Haskell. Such a function has to return a *Bool* which indicates *True* in case the test succeeds or *False* if not and can take input arguments which data is automatically generated by QuickCheck.

```
-- append operator (++) is associative
append_associative :: [Int] -> [Int] -> [Int] -> Bool
```

```haskell
append_associative xs ys zs = (xs ++ ys) ++ zs == xs ++ (ys ++ zs)

-- The reverse of a reversed list is the original list
reverse_reverse :: [Int] -> Bool
reverse_reverse xs = reverse (reverse xs) == xs

-- reverse is distributive over append (++)
-- This test fails for explanatory reasons, for a correct
-- property xs and ys need to be swapped on the right-hand side!
reverse_distributive :: [Int] -> [Int] -> Bool
reverse_distributive xs ys = reverse (xs ++ ys) == reverse xs ++ reverse ys

-- running the tests
main :: IO ()
main = do
  quickCheck append_associative
  quickCheck reverse_reverse
  quickCheck reverse_distributive
```

When we run the tests using *main*, we get the following output:

```
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
*** Failed! Falsifiable (after 5 tests and 6 shrinks):
[0]
[1]
```

We see that QuickCheck generates 100 test cases for each property-test and it does this by generating random data for the input arguments. Note that we have not specified any data for our input arguments; QuickCheck is able to provide a suitable data generator through type inference: for lists and all the existing Haskell types there exist custom data generators already. Note that we have to use a monomorphic list, in our case *Int*, and cannot use polymorphic lists because QuickCheck would not know how to generate data for a polymorphic type. Still, by appealing to genericity and polymorphism, we get the guarantee that the test case is the same for all types of a lists.

QuickCheck generates 100 test cases by default and requires all to pass - if there is a test case which fails, the overall property-test fails and QuickCheck shrinks the input to a minimal size which still fails and reports it as a counter example. This is the case in the last property-test *reverse_distributive* which is wrong as *xs* and *ys* need to be swapped on the right-hand side. In this run, QuickCheck found a counter-example to the property after 5 tests and applied 6 shrinks to find the minimal failing example of *xs = [0]* and *ys = [1]*. If we swap *xs* and *ys*, the property-test passes 100 test cases just like the other two did. Note that it is possible to configure QuickCheck to generate more or less random test cases, which can be used to increase the coverage if the sampling space is quite large - this will become useful later.

Generators

QuickCheck comes with a lot of data generators for existing types like *Strings, Int, Double, Lists,...* but in case one wants to randomize custom data types one has to write custom data generators. There are two ways to do this: fix them at compile time by writing an *Arbitrary* instance or write a run-time generator running in the *Gen* Monad. The advantage of having an *Arbitrary* instance is that the custom data type can then be used as random argument to a function as in the examples above.

Lets implement a custom data generator for the *SIRState* for both cases. We start with the run-time option, running in the *Gen* Monad:

```
genSIRState :: Gen SIRState
genSIRState = elements [Susceptible, Infected, Recovered]
```

This implementation makes use of the *elements :: [a]* $\rightarrow$ *Gen a* functions, which picks a random element from a non-empty list with uniform probability. If a skewed distribution is needed, one can use the *frequency :: [(Int, Gen a)]* $\rightarrow$ *Gen a* function, where a frequency can be specified for each element. For example generating on average 80% Susceptible, 15% Infected and 5% Recovered can be achieved using this function:

```
genSIRState :: Gen SIRState
genSIRState = frequency [(80, Susceptible), (15, Infected), (5, Recovered)]
```

Implementing an *Arbitrary* instance is straightforward, one only needs to implement the *arbitrary :: Gen a* method:

```
instance Arbitrary SIRState where
  arbitrary = genSIRState
```

When we have a random *Double* as input to a function but want to restrict its random range to (0,1) because it reflects a probability, we can do this easily with *newtype* and implementing an *Arbitrary* instance. The same can be done for limiting the simulation duration to a lower range than the full *Double* range.

```
newtype Probability = P Double deriving Show
newtype TimeRange   = T Double deriving Show

instance Arbitrary Probability where
  arbitrary = P <$> choose (0, 1)

instance Arbitrary TimeRange where
  arbitrary = T <$> choose (0, 50)
```

The simulations we run all rely on a random-number generator, thus we need a randomly initialised random-number generator each time we run a simulation. This can be easily achieved by drawing a seed from the full *Int* range and creating an *StdGen* from it:

```
genStdGen :: Gen StdGen
-- min/maxBound are defined in the Haskell Prelude and
-- define the smallest and largest value of a Bounded type
genStdGen = mkStdGen <$> choose (minBound, maxBound)

instance Arbitrary StdGen where
  arbitrary = genStdGen
```

This generator then can be used to write another custom data generator which generates simulation runs. Here we give an example for the time-driven SIR:

```
genTimeSIR :: [SIRState]  -- ^ Population
           -> Double      -- ^ Contact rate
           -> Double      -- ^ Infectivity
           -> Double      -- ^ Illness duration
           -> Double      -- ^ Time Delta
           -> Double      -- ^ Time Limit
           -> Gen [(Double, (Int, Int, Int))]
genTimeSIR as cor inf ild dt tMax
  = runTimeSIR as cor inf ild dt tMax <$> genStdGen
```

Distributions

As already mentioned, QuickCheck provides functions to measure the coverage of test cases. This can be done using the *label :: Testable prop* $\Rightarrow$ *String* $\rightarrow$ *prop* $\rightarrow$ *Property* function. It takes a *String* as first argument and a testable property and constructs a *Property*. QuickCheck collects all generated labels, counts their occurrences and reports their distribution. For example it could be used to get a rough idea about the length of the random lists created in the *reverse_reverse* property shown above:

```
reverse_reverse_label :: [Int] -> Property
reverse_reverse_label xs
  = label ("length of random-list is " ++ show (length xs))
          (reverse (reverse xs) == xs)
```

When running the test, we see the following output:

```
+++ OK, passed 100 tests:
 5% length of random-list is 27
 5% length of random-list is 15
 5% length of random-list is 0
 4% length of random-list is 4
 4% length of random-list is 19
 ...
```

Coverage

The most powerful functions to work with test case distributions though are *cover* and *checkCoverage*. The function *cover :: Testable prop* $\Rightarrow$ *Double* $\rightarrow$

Bool → *String* → *prop* → *Property* allows to explicitly specify that a given percentage of successful test cases belong to a given class. The first argument is the expected percentage, the second argument is a *Bool* indicating whether the current test case belongs to the class or not; the third argument is a label for the coverage; the fourth argument is the property which needs to hold for the test case to succeed.

Lets look at an example - we use *cover* to express that we expect 15% of all test cases to have a random list with at least 50 elements.

```
reverse_reverse_cover :: [Int] -> Property
reverse_reverse_cover xs
  = cover 15 (length xs >= 50) "Length of random-list at least 50"
             (reverse (reverse xs) == xs)
```

When repeatedly running the test, we see the following output:

```
+++ OK, passed 100 tests (10% length of random-list at least 50).
Only 10% Length of random-list at least 50, but expected 15%.
+++ OK, passed 100 tests (21% length of random-list at least 50).
```

As can be seen, QuickCheck runs the default 100 test cases and prints a warning if the expected coverage is not reached. This is a useful feature but it is up to us to decide whether 100 test cases are suitable and whether we can really claim that the given coverage will be reached or not. Fortunately, QuickCheck provides the powerful function *checkCoverage :: Testable prop* ⇒ *prop* → *Property* which does this for us. When *checkCoverage* is used, QuickCheck will run an increasing number of test cases until it can decide whether the percentage in *cover* was reached or cannot be reached at all. The way QuickCheck does it, is by using sequential statistical hypothesis testing [162], thus if QuickCheck comes to the conclusion that the given percentage can or cannot be reached, it is based on a robust statistical test giving strong confidence in the result.

When we run the example from above but now with *checkCoverage* we get the following output:

```
+++ OK, passed 12800 tests
    (15.445% length of random-list at least 50).
```

We see that after QuickCheck has run 12,800 tests it came to the statistically robust conclusion that indeed at least 15% of the test cases have a random list with at least 50 elements.

Emulating failure

As already mentioned, *all* test cases have to pass for the whole property-test to succeed. If just a single test case fails, the whole property-test fails. This is sometimes too strong, especially when we are dealing with stochastic systems like ABS models.

The function *cover* can be used to emulate failure of test cases and get a measure of failure. Instead of computing the *True/False* property in the last *prop*

argument, we set the last argument always to *True* and compute the *True/False* property in the second *Bool* argument, indicating whether the test case belongs to the class of passed tests or not. This has the effect that *all* test cases are successful but that we get a distribution of failed/successful ones. In combination with *checkCoverage*, this is a particularly powerful pattern for testing ABS which allows us to test hypotheses and statistical tests on distributions as will be shown in the following chapters.

Chapter 8

Testing agent specifications

In this chapter we are showing how to use QuickCheck to encode full agent-specifications directly in code as property-tests. These properties serve then both as formal specification and tests at the same time - a fundamental strength of property-based testing, not possible with unit testing in this strong expressive form. Besides the high expressivity, QuickCheck also allows us to state statistical coverage for certain cases, which allows to express statistical properties of the agents behaviour, something also not directly possible with unit-testing. This is a very strong indication that property-based testing is a natural fit to test agent-based simulation. We discuss both time- and event-driven implementations of the agent-based SIR model as introduced in Chapter 2.2.1.

8.1 Event-driven specification

In this section we present how QuickCheck can be used to test event-driven agents by expressing their *specification* as property-tests in the case of the event-driven SIR implementation from chapter 5.1.

In general, testing event-driven agents is fundamentally different and more complex than testing time-driven agents, as their interface surface is generally much larger: events form the input to the agents to which they react with new events - the dependencies between those can be quite complex and deep. Using property-based tests we can encode the invariants and end up with an actual specification of their behaviour, acting as documentation, regression test within a TDD and property tests.

Note that the concepts presented here are applicable with slight adjustments to the Sugarscape implementation as well but we focused on the SIR one as its specification is shorter and does not require as much in-depth details - after all we are interested in deriving concepts, not dealing with specific technicalities.

With event-driven ABS a good starting point in specifying and then testing the system is simply relating the input events to expected output events. In the SIR implementation we have only three events, making it feasible to give a full

formal specification - note that the Sugarscape implementation has more than 16 events, which makes it much harder to test it with sufficient coverage, giving a good reason to primarily focus on the SIR implementation.

8.1.1 Deriving the specification

We start by giving the full *specification* of the susceptible, infected and recovered agent by stating the input-to-output event relations. The susceptible agent is specified as follows:

1. *MakeContact* - If the agent receives this event it will output β *Contact ai Susceptible* events, where ai is the agents self id. The events have to be scheduled immediately without delay, thus having the current time as scheduling timestamp. The receivers of the events are uniformly randomly chosen from the agent population. The agent doesn't change its state, stays *Susceptible* and does not schedule any other events than the ones mentioned.

2. *Contact _ Infected* - If the agent receives this event there is a chance of uniform probability γ (infectivity) that the agent becomes *Infected*. If this happens, the agent will schedule a *Recover* event to itself into the future, where the time is drawn randomly from the exponential distribution with $\lambda = \delta$ (illness duration). If the agent does not become infected, it will not change its state, stays *Susceptible* and does not schedule any events.

3. *Contact _ _* or *Recover* - If the agent receives any of these (other) events it will not change its state, stays *Susceptible* and does not schedule any events.

This specification implicitly covers that a susceptible agent can never transition from a *Susceptible* to a *Recovered* state within a single event - it can only make the transition to *Infected* or stays *Susceptible*. The infected agents are specified as follows:

1. *Recover* - If the agent receives this, it will not schedule any events and make the transition to the *Recovered* state.

2. *Contact sender Susceptible* - If the agent receives this, it will reply immediately with *Contact ai Infected* to *sender*, where *ai* is the infected agents' id and the scheduling timestamp is the current time. It will not schedule any events and stays *Infected*.

3. In case of any other event, the agent will not schedule any events and stays *Infected*.

This specification implicitly covers that an infected agent never goes back to the *Susceptible* state - it can only make the transition to *Recovered* or stay *Infected*. From the specification of the susceptible agent it becomes clear that a

susceptible agent who became infected, will always recover as the transition to *Infected* includes the scheduling of *Recovered* to itself.

The *recovered* agents specification is very simple. It stays *Recovered* forever and does not schedule any events.

The question is now how to put these into a property-test with QuickCheck. We focus on the susceptible agent, as it it the most complex one, which concepts can then be easily applied to the other two. Generally speaking, we create a random *susceptible* agent and a random event, feed it to the agent to get the output and check the invariants accordingly to input and output.

8.1.2 Encoding invariants

We start by encoding the invariants of the susceptible agent directly into Haskell, implementing a function which takes all necessary parameters and returns a *Bool* indicating whether the invariants hold or not. The encoding is straightforward when using pattern matching and it nearly reads like a formal specification due to the declarative nature of functional programming.

```
susceptibleProps :: SIREvent              -- ^ Random event sent to agent
                 -> SIRState              -- ^ Output state of the agent
                 -> [QueueItem SIREvent]  -- ^ Events the agent scheduled
                 -> AgentId               -- ^ Agent id of the agent
                 -> Bool
-- received Recover => stay Susceptible, no event scheduled
susceptibleProps Recover Susceptible es _ = null es
-- received MakeContact => stay Susceptible, check events
susceptibleProps MakeContact Susceptible es ai
  = checkMakeContactInvariants ai es cor
-- received Contact _ Recovered => stay Susceptible, no event scheduled
susceptibleProps (Contact _ Recovered) Susceptible es _ = null es
-- received Contact _ Susceptible => stay Susceptible, no event scheduled
susceptibleProps (Contact _ Susceptible) Susceptible es _  = null es
-- received Contact _ Infected, didn't get Infected, no event scheduled
susceptibleProps (Contact _ Infected) Susceptible es _ = null es
-- received Contact _ Infected AND got infected, check events
susceptibleProps (Contact _ Infected) Infected es ai
  = checkInfectedInvariants ai es
-- all other cases are invalid and result in a failed test case
susceptibleProps _ _ _ _ = False
```

Next, we give the implementation for the *checkMakeContactInvariants* and *checkInfectedInvariants* functions. The function *checkMakeContactInvariants* encodes the invariants which have to hold when the susceptible agent receives a *MakeContact* event. The *checkInfectedInvariants* function encodes the invariants which have to hold when the susceptible agent got *Infected*. Both implementations read like a formal specification, again thanks to the declarative nature of functional programming and pattern matching:

```
checkInfectedInvariants :: AgentId              -- ^ Agent id of the agent
                        -> [QueueItem SIREvent] -- ^ Events the agent scheduled
                        -> Bool
```

```
checkInfectedInvariants sender
  -- expect exactly one Recovery event
  [QueueItem receiver (Event Recover) t']
  -- receiver is sender (self) and scheduled into the future
  = sender == receiver && t' >= t
-- all other cases are invalid
checkInfectedInvariants _ _ = False
```

The *checkMakeContactInvariants* is a bit more complex but reads as a formal specification as well:

```
checkMakeContactInvariants :: AgentId              -- ^ Agent id of the agent
                           -> [QueueItem SIREvent] -- ^ Events the agent scheduled
                           -> Int                  -- ^ Contact Rate
                           -> Bool
checkMakeContactInvariants sender es contactRate
    -- make sure there has to be exactly one MakeContact event and
    -- exactly contactRate Contact events
    = invOK && hasMakeCont && numCont == contactRate
  where
    (invOK, hasMakeCont, numCont)
      = foldr checkMakeContactInvariantsAux (True, False, 0) es

    checkMakeContactInvariantsAux :: QueueItem SIREvent
                                  -> (Bool, Bool, Int)
                                  -> (Bool, Bool, Int)
    checkMakeContactInvariantsAux
        (QueueItem (Contact sender' Susceptible) receiver t') (b, mkb, n)
      = (b && sender == sender'    -- the sender in Contact must be the Susceptible agent
           && receiver `elem` ais  -- the receiver of Contact must be in the agent ids
           && t == t', mkb, n+1)   -- the Contact event is scheduled immediately
    checkMakeContactInvariantsAux
        (QueueItem MakeContact receiver t') (b, mkb, n)
      = (b && receiver == sender   -- the receiver of MakeContact is the Susceptible agent itself
           && t' == t + 1          -- the MakeContact event is scheduled 1 time unit into the future
           &&  not mkb, True, n)   -- there can only be one MakeContact event
    checkMakeContactInvariantsAux _ (_, _, _)
      = (False, False, 0)          -- other patterns are invalid
```

What is left is to actually write a property-test using QuickCheck. We are making heavy use of random parameters to express that the properties have to hold invariant of the model parameters. We make use of additional data generator modifiers: *Positive* ensures that the value generated is positive; *NonEmptyList* ensures that the randomly generated list is non-empty.

```
prop_susceptible_invariants :: Positive Int         -- ^ Contact rate
                            -> Probability          -- ^ Infectivity
                            -> Positive Double      -- ^ Illness duration
                            -> Positive Double      -- ^ Current simulation time
                            -> NonEmptyList AgentId -- ^ Agent ids of the population
                            -> Gen Property
prop_susceptible_invariants
  (Positive cor) (P inf) (Positive ild) (Positive t) (NonEmpty ais) = do
  -- generate random event, requires the population agent ids
  evt <- genEvent ais
  -- run susceptible random agent with given parameters
```

```
(ai, ao, es) <- genRunSusceptibleAgent cor inf ild t ais evt
-- check properties
return $ property $ susceptibleProps evt ao es ai
```

When running this property-test all 100 test cases pass. Due to the large random sampling space with 5 parameters, we increase the number of test cases to generate to 100,000 - still all test cases pass.

8.1.3 Encoding transition probabilities

In the specifications above there are probabilistic state-transitions, for example an infected agent *will* recover after a given time, which is randomly distributed with the exponential distribution. The susceptible agent *might* become infected, depending on the events it receives and the infectivity (γ) parameter. We look now into how we can encode these probabilistic properties using the powerful *cover* and *checkCoverage* feature of QuickCheck.

8.1.3.1 Susceptible agent

We follow the same approach as in encoding the invariants of the susceptible agent but instead of checking the invariants, we compute the probability for each case. Note that in this property-test we cannot randomise the model parameters because this would lead to random coverage. This might seem like a disadvantage but we do not really have a choice here - still, the model parameters can be adjusted arbitrarily and the property (must) still hold. We make use of the *cover* function together with *checkCoverage*, which ensures that we get a statistical robust estimate whether the expected percentages can be reached or not. Implementing this property-test is then simply a matter of computing the probabilities and of case analysis over the random input event and the agents output.

```
...
case evt of
  Recover ->
    cover recoverPerc True
     ("Susceptible receives Recover, expected " ++ show recoverPerc) True
...
```

Note the usage pattern of *cover*: we always include the test case into the coverage class and all test cases pass. The reason for this is that we are just interested in testing the coverage, which is in fact the property we want to test. We could have combined this test into the previous one but then we couldn't have use randomised model parameters. For this reason, and to keep the concerns separated we opted for two different tests, which makes them also much more readable.

When running the property-test we get the following output:

```
+++ OK, passed 819200 tests:
33.3582% Susceptible receives MakeContact, expected 33.33%
```

```
33.2578% Susceptible receives Recover, expected 33.33%
11.1643% Susceptible receives Contact * Recovered, expected 11.11%
11.1096% Susceptible receives Contact * Susceptible, expected 11.11%
10.5616% Susceptible receives Contact * Infected, stays Susceptible, expected 10.56%
 0.5485% Susceptible receives Contact * Infected, becomes Infected, expected 0.56%
```

After 819,200 (!) test cases QuickCheck comes to the conclusion that the distributions generated by the test cases reflect the expected distributions and passes the property-test. We see that the values do not match exactly in some cases but by using sequential statistical hypothesis testing QuickCheck is able to conclude that the coverage are statistically equal.

8.1.3.2 Infected agent

We want to write a property-test which checks whether the transition from *Infected* to *Recovered* actually follows the exponential distribution with a fixed δ (illness duration). The idea is to compute the expected probability for agents having an illness duration of less or equal δ. This probability is given by the cumulative density function (CDF) of the exponential distribution. The question is how to get the infected illness duration. This is simply achieved by infecting a susceptible agent and taking the scheduling time of the *Recover* event. We have written a custom data generator for this:

```haskell
getInfectedAgentDuration :: Double -> Gen (SIRState, Double)
getInfectedAgentDuration ild = do
  -- with these parameters the susceptible agent WILL become infected
  (_, ao, es) <- genRunSusceptibleAgent 1 1 ild 0 [0] (Contact 0 Infected)
  return (ao, recoveryTime es)
  where
    -- expect exactly one event: Recover
    recoveryTime :: [QueueItem SIREvent] -> Double
    recoveryTime [QueueItem Recover _ t]  = t
    recoveryTime _ = 0
```

Encoding the probability check into a property-test is straightforward:

```haskell
prop_infected_duration :: Property
prop_infected_duration = checkCoverage (do
  -- fixed model parameter, otherwise random coverage
  let ild  = 15
  -- compute probability drawing a random value less or equal
  -- ild from the exponential distribution (follows the CDF)
  let prob = 100 * expCDF (1 / ild) ild

  -- run random susceptible agent to become infected and
  -- return agents state and recovery time
  (ao, dur) <- getInfectedAgentDuration ild

  return (cover prob (dur <= ild)
            ("Infected agent recovery time is less or equals " ++ show ild ++
             ", expected at least " ++ show prob)
            (ao == Infected)) -- final state has to Infected
```

When running the property-test we get the following output.

```
+++ OK, passed 3200 tests
    (63.62% Infected agent recovery time is less or equals 15.0,
     expected at least 63.21%).
```

QuickCheck is able to determine after only 3,200 test cases that the expected coverage is met and passes the property-test.

8.2 Time-driven specification

The time-driven SIR agents have a very small interface: they only receive the agent population from the previous step and output their state in the current step. We can also assume an implicit forward flow of time, statically guaranteed by Yampas arrowized FRP. Thus a specification in time-driven approach is given in terms of probabilities and timeouts, rather than in events as in the event-driven testing as presented before.

- Susceptible agent - makes *on average* contact with β (contact rate) agents per time unit. The distribution follows the exponential distribution with $\lambda = \frac{1}{\beta}$. If a susceptible agent gets into contact with an infected agent, it will become infected with a uniform probability of γ (infectivity).
- Infected agent - *will* recover *on average* after δ (illness duration) time units. The distribution follows the exponential distribution with $\lambda = \delta$.
- Recovered agent - stays recovered *forever*.

8.2.1 Specifications of the susceptible agent

We cannot directly observe that a susceptible agent makes contact with other agents like we can in the event-driven approach but only indirectly through its change of state: a susceptible agent *might* become infected if there are infected agents in the population. Thus when we run a susceptible agent for some time, we have 3 possible outcomes of the agents output stream: 1. the agent did not get infected and thus all elements of the stream are *Susceptible*; 2. the agent got infected thus up to a given index in the stream all elements are *Susceptible* and change to *Infected* after; 3. the agent got *Infected* and then *Recovered* thus the stream is the same as in infected but there is a second index after which all elements change to *Recovered*. Encoding them in code is straightforward:

```
susceptibleInvariants :: [SIRState] -- ^ The output stream of the susceptible agent
                      -> Bool       -- ^ The population contains an infected agent
                      -> Bool       -- ^ True in case the invariant holds
susceptibleInvariants aos infInPop
    -- Susceptible became Infected and then Recovered
    | isJust recIdxMay
      = infIdx < recIdx &&  -- agent has to become infected before recovering
        all (==Susceptible) (take infIdx aos) &&
        all (==Infected) (take (recIdx - infIdx) (drop infIdx aos)) &&
        all (==Recovered) (drop recIdx aos) &&
```

```
        infInPop  -- can only happen if there are infected in the population

    -- Susceptible became Infected
    | isJust infIdxMay
      = all (==Susceptible) (take infIdx aos) &&
        all (==Infected) (drop infIdx aos) &&
        infInPop -- can only happen if there are infected in the population

    -- Susceptible stayed Susceptible
    | otherwise = all (==Susceptible) aos
  where
    -- look for the first element when agent became Infected
    infIdxMay = elemIndex Infected aos
    -- look for the first element when agent became Recovered
    recIdxMay = elemIndex Recovered aos

    infIdx = fromJust infIdxMay
    recIdx = fromJust recIdxMay
```

Putting this into a property-test is also straightforward. We generate a random population, run a random susceptible agent with a sampling rate of $\Delta t = 0.01$ and check the invariants on its output stream. These invariants all have to hold independently from the (positive) duration we run the random susceptible agent for, thus we run it for a random amount of time units. The invariants also have to hold for arbitrary positive beta (contact rate), gamma (infectivity) and delta (illness duration). At the same time, we want to get an idea of the percentage of agents which stayed susceptible, became infected or made the transition to recovered, thus we *label* all our test cases accordingly.

```
prop_susceptible_invariants :: Positive Double  -- ^ beta, contact rate
                            -> Probability      -- ^ gamma, infectivity within (0,1) range
                            -> Positive Double  -- ^ delta, illness duration
                            -> TimeRange        -- ^ simulation duration, within (0,50) range
                            -> [SIRState]       -- ^ random population
                            -> Property
prop_susceptible_invariants
      (Positive cor) (P inf) (Positive ild) (T t) as = property (do
    -- population contains an infected agent True/False
    let infInPop = Infected `elem` as

    -- run a random susceptible agent for random time units with
    -- sampling rate dt 0.01 and return its stream of output
    aos <- genSusceptible cor inf ild as t 0.01

    return
        -- label all test cases
        label (labelTestCase aos)
        -- check invariants on output stream
        (property (susceptibleInvariants aos infInPop))
  where
    labelTestCase :: [SIRState] -> String
    labelTestCase aos
      | Recovered `elem` aos = "Susceptible -> Infected -> Recovered"
      | Infected `elem` aos  = "Susceptible -> Infected"
      | otherwise            = "Susceptible"
```

Due to the high dimensionality of the random sampling space, we run 10,000 tests - all succeed as expected.

```
SIR Agent Specifications Tests
  Susceptible agents invariants: OK (12.72s)
    +++ OK, passed 10000 tests:
    55.78% Susceptible -> Infected -> Recovered
    37.19% Susceptible -> Infected
     7.03% Susceptible
```

This test so far did not state anything about the probability of a susceptible agent getting infected. The probability for it is bimodal (see next Chapter) due to the combined probabilities of the exponential distribution of the contact rate and the uniform distribution of the infectivity. Unfortunately, the bimodality makes it not possible to compute a coverage percentage of infected in this case, as we did in the event-driven test because the bimodal distribution can only be described in terms of a distribution and not a single probability. This was possible in the even-driven approach because we decoupled the production of the *Contact _ Infected* event from the infection: both were uniform distributed, thus we could compute a coverage percentage. Thus we see that different approaches also allow different explicitness of testing.

8.2.2 Probabilities of the infected agent

An infected agent *will* recover after *finite* time, thus we assume that there exists an index in the output stream, where the elements will change to *Recovered*. From the index we can compute the time of recovery, knowing the fixed sampling rate Δt.

```
infectedInvariant :: [SIRState]      -- ^ The stream of outputs from the infected agent
                  -> Double          -- ^ Sampling rate dt
                  -> Maybe Double    -- ^ Just recovery time, or Nothing if not recovered
infectedInvariant aos dt  = do
  -- search for the index of the first Recovery element
  recIdx <- elemIndex Recovered aos
  -- all elements up to the index need to be Infected,
  -- because the agent cannot go back to Susceptible
  if all (==Infected) (take recIdx aos)
    then Just (dt * recIdx)
    else Nothing
```

To put this into a property-test, we follow a similar approach as in the event-driven case of the infected agents invariants. We employ the CDF of the exponential distribution to get the probability of an agent recovering within δ (illness duration) time steps. We then run a random infected agent for an *unlimited* time with a sampling rate o f $\Delta t = 0.01$ and search in its potentially infinite output stream for the first occurrence of an *Infected* element to compute the recovery time, as shown in the invariant above. The code is conceptually exactly the same as in the event-driven case, so we do not repeat the property-test here.

When running the test we get the following output, indicating that QuickCheck finds the coverage satisfied after 3,200 test cases:

```
+++ OK, passed 3200 tests (62.28% infected agents have an illness
    duration of 15.0 or less, expected 63.21).
```

The fact that we run the random infected agent without time-limit explicitly expresses the invariant that an infected agent *will* recover in *finite* time steps: a correct implementation will produce a stream which contains an index after which all elements are *Infected*, thus resulting in *Just* recovery time. This is also a direct expression of the fact that the CDF of the exponential distribution reaches 1.0 at infinity. An approach which would guarantee the termination would be to limit the time to run the infected agent to *illnessDuration* and evaluate the property always to True. This approach guarantees termination but removes an important part of the specification - we decided to follow the initial approach to make the specification really clear, and in practice it has turned out to terminate within a very short time (see below).

8.2.3 The non-computability of the recovered agent test

The property-test for the recovered agent is trivial: we run a random recovered agent for a random number of time units with $\Delta t = 0.01$ and require that all elements in the output stream are *Recovered*. Of course this is no proof that the recovered agent stays recovered *forever* as this would take *forever* to test and is thus not computable. Here we are hitting the limits of what is possible with random black-box testing: without looking at the actual implementation it is not possible to prove that the recovered agent is really behaving as specified. We made this fact very clear at the beginning of this part: property-based testing is not a proof for the correctness but is only a support for raising the confidence in the correctness by constructing cases which show that the behaviour is not incorrect.

To be really sure that the recovered agent behaves as specified we need to employ white-box verification and look at the actual implementation. It is immediately obvious that the implementation follows the specification and actually *is* the specification, and we can even regard it as a very concise proof that it will stay recovered *forever*:

```
recoveredAgent :: SIRAgent
recoveredAgent = constant Recovered
```

The signal function *constant* is the *const* function lifted into an arrow: *constant b = arr (const b)*. This should be proof enough that a recovered agent will stay recovered *forever*. We discuss the topic of computability in pure functional ABS in a slightly different context in Appendix C.

8.3 Discussion

In this section we have shown how to express the specifications of both the event- and time-driven agent behaviour directly in code as properties and how to implement property-tests in QuickCheck for them. The approach to event-driven properties was to establish a correspondence between an input-event the current agent-state and the output events and the new agents state. In case of the time-driven agent, the properties are expressed in terms of a potentially infinite stream of agent output-states. Although both implementations follow the same underlying model, the technical details of the properties differ substantially. The reason for this is that although property-based testing is a black-box verification technique, the implementation often requires substantial knowledge of the internal details as can be seen especially in the event-driven case.

The resulting properties are highly expressive due to pattern matching and declarative programming and can be regarded as a kind of formal specification. Together with the properties which check the state-transition probabilities, we claim that the property-tests shown in this chapter fully specify both the event- and time-driven agent behaviour. This is a first example emphasising the usefulness of QuickCheck for testing ABS, providing a first strong evidence for the hypothesis that randomised property-testing is a good match for testing ABS.

Curiously, the implementation of all the specifications and property-tests has substantially more lines of code than the original implementations. However, this is not relevant here: we showed how to implement a full specification of an ABS model as a property-based test and we succeeded! This is definitely a strong indication that our hypothesis that randomised property-based testing is a suitable tool for testing ABS is valid. With unit tests we would be quite lost here: even for the SIR model, it is hard to enumerate all possible interactions and cases but by stating invariants as properties and generating random test-cases we make sure they are checked.

We have not looked into more complex testing patterns like the synchronous agent interactions of Sugarscape. We didn't look into testing full agent and interacting agent behaviour using property-tests due to its complexity would justify a thesis on its own. Due to its inherent stateful nature with complex dependencies between valid states and agents actions we need a more sophisticated approach as outlined in [38], where the authors show how to build a meta-model and commands which allow to specify properties and valid state-transitions which can be generated automatically. We leave this for further research.

By exploiting lazy evaluation in the time-driven tests we scratch on what is conveniently possible in established approaches to ABS: we can let the simulation run potentially forever as in the case of the infected agent and rely on the correctness of the implementation to terminate in finite step when consuming the potentially infinite stream.

We did not include an explicit environment in our agent specification tests and assumed a full connected network where all agents can make contact with

each other. We claim that property-based testing is highly useful there as well, especially when dealing with random environments like in Sugarscape or social and random networks [44, 85]. We leave this for further research but we hypothesise that for the SIR model all properties presented here should still hold under different environments.

Chapter 9

Testing model invariants

The tests of the event-driven implementation in the previous chapter were stateless: only one computational step of an agent was considered by feeding a single event and ignoring the agent continuation. Also the events didn't contain any notion of time as they would carry within the queue. Feeding follow-up events into the continuation would make testing inherently stateful as we introduce history into the system. Such tests would allow to test the full life-cycle of one agent or a full population.

In this chapter we will discuss how we can encode properties and specifications which require stateful testing. We define stateful testing here as: evolving a simulation state consisting of one or more agents over multiple events. Note that this also includes running the whole simulation. Note that we primarily focus on the event-driven implementation here unless noted otherwise.

We first show how we can encode actual laws of the underlying SIR model into properties and write property-tests in QuickCheck for them. We then employ random event-sampling to check whether these invariants also hold when ignoring the event-interdependencies between agents. Further, we compare both the event- and time-driven implementations with each other, giving an excellent use case for property-based testing in ABS. Finally we show how to verify both the time- and event-driven implementations against the original SD specification.

9.1 Invariants in simulation dynamics

By informally reasoning about the agent specification and by realising that they are in fact a state-machine with a one-directional flow of *Susceptible* $\rightarrow$ *Infected* $\rightarrow$ *Recovered*, we can come up with a few invariants which have to hold for any SIR simulation run, independent of the random-number stream and the population:

1. Simulation time is monotonic increasing. Each event carries a timestamp when it is scheduled. This timestamp may stay constant between multiple

events but will eventually increase and must never decrease. Obviously this invariant is a fundamental assumption in most simulations: time advances into the future and does not flow backwards.

2. The number of total agents N stays constant. The SIR model does not specify the dynamic creation or removal of agents during simulation. This is in contrast to the Sugarscape where, depending on the model parameters, this can be very well the case.

3. The number of *Susceptible* agents S is monotonic decreasing. Susceptile agents *might* become infected, reducing the total number of susceptible agents but they can never increase because neither an infected nor recovered agent can go back to *Susceptible*.

4. The number of *Recovered* agents R is monotonic increasing. This is because infected agents *will* recover, leading to an increase of recovered agents but once the recovered state is reached, there is no escape from it.

5. The number of *Infected* agents respects the invariant of the equation $I = N - (S + R)$ for every step. This follows directly from the first property which says $N = S + I + R$.

9.1.1 Encoding the invariants

All of those properties are easily expressed directly in code and read like a formal specification due to the declarative nature of functional programming:

```
sirInvariants :: Int                    -- ^ N total number of agents
              -> [(Time,(Int,Int,Int))] -- ^ simulation output for each step/event: (Time, (S,I,R))
              -> Bool
sirInvariants n aos = timeInc && aConst && susDec && recInc && infInv
  where
    (ts, sirs)  = unzip aos
    (ss, _, rs) = unzip3 sirs

    -- 1. time is monotonic increasing
    timeInc = allPairs (<=) ts
    -- 2. number of agents N stays constant in each step
    aConst = all agentCountInv sirs
    -- 3. number of susceptible S is monotonic decreasing
    susDec = allPairs (>=) ss
    -- 4. number of recovered R is monotonic increasing
    recInc = allPairs (<=) rs
    -- 5. number of infected I = N - (S + R)
    infInv = all infectedInv sirs

    agentCountInv :: (Int,Int,Int) -> Bool
    agentCountInv (s,i,r) = s + i + r == n

    infectedInv :: (Int,Int,Int) -> Bool
    infectedInv (s,i,r) = i == n - (s + r)
```

```
allPairs :: (Ord a, Num a) => (a -> a -> Bool) -> [a] -> Bool
allPairs f xs = all (uncurry f) (pairs xs)

pairs :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)
```

Putting this property into a QuickCheck test is straightforward. Note that we randomise the model parameters β (contact rate), γ (infectivity) and δ (illness duration) because the properties have to hold for all positive, finite model parameters.

```
prop_sir_invariants :: Positive Int     -- ^ beta, contact rate
                    -> Probability      -- ^ gamma, infectivity in range (0,1)
                    -> Positive Double  -- ^ delta, illness duration
                    -> TimeRange        -- ^ random duration in range (0, 50)
                    -> [SIRState]       -- ^ population
                    -> Property
prop_sir_invariants
    (Positive cor) (P inf) (Positive ild) (T t) as  = property (do
  -- total agent count
  let n = length ss
  -- run the SIR simulation with a new RNG
  ret <- genSimulationSIR ss cor inf ild t
  -- check invariants and return result
  return (sirInvariants n ret)
```

Due to the large sampling space, we increase the number of test cases to run to 10,000 and unsurprisingly all tests pass. Note that we put a random time-limit of (0,50) on the simulations to run, meaning that if a simulation does not terminate before that limit, it will be terminated at that random t. This is actually not necessary because we can reason that the SIR simulation *will always* reach an equilibrium in finite steps thus not requiring an actual time-limit - we discuss this more in-depth in Appendix C.

9.1.2 Random event sampling invariants

An interesting question is whether or not these properties depend on correct interdependencies of events the agents send to each other in reaction to events they receive. Put in other words: do these invariants also hold under *random event sampling*? To test this, instead of using the actual SIR implementation, which inserts the events generated by the agents into the event-queue, we wrote a new SIR kernel. It completely ignores the events generated by the agents and instead makes use of an infinite stream of random queue-elements from which it executes a given number, 100,000 in our case. Note that queue-elements contain a timestamp, the receiving agent id and the actual event: the timestamp is ensured to be increasing, to hold up the monotonic time property, the receiving agent id is drawn randomly from the constant list of all agents in the simulation and the actual event is randomly generated. As it turns out, the implementation of the agents ensure that the SIR properties are also invariant under *random event sampling* - all tests pass.

9.1.3 Time-driven invariants

We can expect that the invariants above also hold for the time-driven implementation. The property-test is exactly the same, with the time-driven implementation running instead of the even-driven one. A big difference is that is not necessary to check the property of monotonic increasing time, as it is an invariant statically guaranteed by arrowized FRP through the Yampa implementation. Due to the fact that the flow of time is always implicitly forward and no time-variable is explicitly made accessible within the code, it is not possible to violate the forward flow of time.

When we ran the property-test we got a big surprise though: after a few test cases the property-test failed due to a violation of the invariants! After a little bit of investigation it became clear that the invariant *(3) number of susceptible agents is monotonic decreasing* was violated: in the failing test case the number of susceptible agents is monotonic decreasing with the exception of one step where it *increases* by 1 just to decrease by 1 in the next step. A coverage test reveals that this happens in about 66% of 1,000 test cases.

The technicalities of the problem are highly involved and not provided in depth here. The source of the problem are the semantics of *switch* and *dpSwitch* which could lead to a delayed output of the agent state, leading to inconsistencies when feeding it back as environment in the next step. The solution is to delay the output of the susceptible agent by one step using *iPre* as already shown in the original time-driven implementation of 4.1. This solves the problem and the property-test passes.

9.2 Comparing time- and event-driven implementations

Having two conceptually different implementations of the same model, an obvious questions we want to answer is whether they are producing the same dynamics or not. To be more precise, we need to answer the question whether both simulations produce the same distributions under random model parameters and simulation time. This is a perfect use case for QuickCheck as well and easily encoded into a property-test.

We generate random values for β (contact rate), γ (infectivity) and δ (illness duration) as well as a random population and a random duration to run the simulations for. Note that we restrict γ to be drawn from the (0,1) range as it represents a probability; the random duration is drawn from the (0, 50) range to reduce the run-time duration to a reasonable amount without taking away the randomness.

Both simulation types are run with the same random parameters for 100 replications, collecting the output of the final step. The samples of these replications are then compared using a Mann-Whitney test with a 95% confidence (p-value of 0.05). The reason for choosing this statistical test over a two sample t-test is that the Mann-Whitney test does not require the samples to be normally

distributed. We know both from experimental observations and discussions in [100] that both implementations produce a bimodal distribution, thus we have to use a non-parametric test like Mann-Whitney to compare them.

We expect a high coverage of at least 90%, which makes our assumption explicit that we expect both simulations to produce highly similar distributions despite their different underlying implementations.

```
prop_event_time_equal :: Positive Int     -- ^ beta, contact rate
                      -> Probability      -- ^ gamma, infectivity, within (0,1) range
                      -> Positive Double  -- ^ delta, illness duration
                      -> TimeRange        -- ^ time to run within (0, 50) range
                      -> [SIRState]       -- ^ population
                      -> Property
prop_event_time_equal
    (Positive cor) (P inf) (Positive ild) (T t) as = checkCoverage (do
  -- run 100 replications for time- and event-driven simulation
  (ssTime, isTime, rsTime)    <- unzip3 <$> genTimeSIRRepls 100 as cor inf ild t
  (ssEvent, isEvent, rsEvent) <- unzip3 <$> genEventSIRRepls 100 as cor inf ild t

  -- confidence of 95 for Mann Whitney test
  let p = 0.05
  -- perform statistical tests
  let ssTest = mannWhitneyTwoSample ssTime ssEvent p
      isTest = mannWhitneyTwoSample isTime isEvent p
      rsTest = mannWhitneyTwoSample rsTime rsEvent p

  let allPass = ssTest && isTest && rsTest

  -- add the test to the coverage tests only if it passes.
  return
    (cover 90 allPass "SIR implementations produce equal distributions" True)
```

Indeed when running this test, enforcing QuickCheck to perform sequential statistical hypothesis testing with *checkCoverage*, after 800 tests QuickCheck passes the test.

```
+++ OK, passed 800 tests
    (90.4% SIR event- and time-driven produce equal distributions).
```

This result shows that both implementations produce highly similar distributions although they are not exactly the same as the 10% of failure shows. We will discuss this issue in a broader context in the next section.

9.3 Testing the SIR model specification

In the previous chapters and sections we have established the correctness of our event- and time-driven implementation up to our informal specification, we derived from the formal SD specification from Chapter 2.2.1. What we are lacking is a verification whether the implementations also match the formal SD specification or not. We aim at connecting the agent-based implementation to the SD specification, by formalising it into properties within a property-test.

The SD specification can be given through the differential equations shown in Chapter 2.2.1, which we repeat here:

$$\begin{aligned} \frac{\mathrm{d}S}{\mathrm{d}t} &= -infectionRate & infectionRate &= \frac{I\beta S\gamma}{N} \\ \frac{\mathrm{d}I}{\mathrm{d}t} &= infectionRate - recoveryRate & recoveryRate &= \frac{I}{\delta} \\ \frac{\mathrm{d}R}{\mathrm{d}t} &= recoveryRate \end{aligned} \tag{9.1}$$

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

$$S(t) = N - I(0) + \int_0^t -infectionRate\,\mathrm{d}t \tag{9.2}$$

$$I(0) = 1 \tag{9.3}$$

$$I(t) = \int_0^t infectionRate - recoveryRate\,\mathrm{d}t \tag{9.4}$$

$$R(t) = \int_0^t recoveryRate\,\mathrm{d}t \tag{9.5}$$

9.3.1 Deriving a property

The goal is now to derive a property which connects those equations with our implementation. We have to be careful and realise a fundamental difference between the SD and ABS implementations: SD is deterministic and continuous, ABS is stochastic and discrete. Thus we cannot compare single runs but we can only compare averages: stated informally, the property we want to implement is that the ABS dynamics matches the SD ones *on average*, independent of the finite population size, model parameters β (contact rate), γ (infectivity) and δ (illness duration) and duration of the simulation. To be able to compare averages, we run 100 replications of the ABS simulation with same parameters except a different random-number generator in each replication and collect the output of the final steps. We then run a two-sided t-test on the replication values with the expected values generated by an SD simulation.

```
compareSDToABS :: Int    -- ^ Initial number of susceptibles
               -> Int    -- ^ Initial number of infected
               -> Int    -- ^ Initial number of recovered
               -> [Int]  -- ^ Final number of susceptibles in replications
               -> [Int]  -- ^ Final number of infected in replications
```

```
                -> [Int]  -- ^ Final number of recovered in replications
                -> Int    -- ^ beta (contact rate)
                -> Double -- ^ gamma (infectivity)
                -> Double -- ^ delta (illness duration)
                -> Time   -- ^ duration of simulation
                -> Bool
compareSDToABS s0 r0 i0
               ss is rs
               beta gamma delta t = sTest && iTest && rTest
  where
    -- run SD simulation to get expected averages
    (s, i, r) = simulateSD s0 i0 r0 beta gamma delta t

    confidence = 0.95
    sTest = tTestSamples TwoTail s (1 - confidence) ss
    iTest = tTestSamples TwoTail i (1 - confidence) is
    rTest = tTestSamples TwoTail r (1 - confidence) rs
```

The implementation of *simulateSD* is discussed in-depth in Appendix A. We are very well aware that comparing the output against an SD simulation is dangerous: after all, why should be trust the SD implementation? As outlined in the Appendix A, great care has been taken to ensure the correctness: the formulas from the SIR specification are directly put into code, allowed by Yampas arrowized FRP which guarantees that at least that translation step is correct - we then only rely on a small enough sampling rate and the correctness of the Yampa library. The former one is very well in our reach and we pick a sufficiently small samply rate; the latter one is beyond our reach but we expect the library to me mature enough to be correct for our purposes.

9.3.2 Implementing the test

Implementing a property-test is straightforward. Here we give the implementation for the time-driven SIR implementation, the implementation for the event-driven SIR implementation is exactly the same with the exception of *genTimeSIRRepls*. We again make use of the *checkCoverage* feature of QuickCheck to get statistical robust results and expect that in 75% of all test cases the SD and ABS dynamics match *on average* - we discuss below why we chose to use a 75% coverage. QuickCheck will run as many tests as necessary to reach a statistically robust result which either allows to reject or accept this hypothesis.

```
prop_sir_time_spec :: Positive Int    -- ^ beta, contact rate
                   -> Propability     -- ^ gamma, infectivity, within (0,1) range
                   -> Positive Double -- ^ delta, illness duration
                   -> TimeRange       -- ^ time to run within (0, 50) range
                   -> [SIRState]      -- ^ population
                   -> Property
prop_sir_time_spec
    (Positive cor) (P inf) (Positive ild) (T t) as = checkCoverage (do
  -- get initial agent numbers
  let (s0,i0,r0) = aggregateSIRStates as
  -- run 100 replications of time-driven SIR implementation
  (ss, is, rs) <- unzip3 <$> genTimeSIRRepls 100 as cor inf ild t
```

```
let prop = compareSDToABS s0 i0 r0 ss is rs cor inf ild t
return $ cover 75 prop "SIR time-driven passes t-test with simulated SD" True
```

9.3.3 Running the test

When running the tests for the time- and event-driven implementation, QuickCheck reports the following:

```
+++ OK, passed 400 tests
    (85.2% SIR time-driven passes t-test with simulated SD).

+++ OK, passed 3200 tests
    (74.84% SIR event-driven passes t-test with simulated SD).
```

The results clearly show that in both cases we reach the expected 75% of coverage: the distributions of the time- and event-driven implementations match the simulated SD dynamics to at least 75%, in case of time-driven this is even substantially higher. Still, this result raises a few questions:

1. Why does the performance of the time-driven implementation surpasses the event-driven one by more than 10%?
2. Why are we not reaching a far higher coverage beyond 90% and why have we chosen 75% in the first place? After all our initial assumption was that the time- and event-driven implementations are simply agent-based implementations of the SD model and thus their dynamics should generate the same distributions as the SD ones.

First of all, the results are a very strong indication that although both implementation techniques try to implement the same underlying model, they generate different distributions and are thus not *statistically* equal. This was already established in Chapter 9, where we have compared the distributions of both simulations and found that although we reach 90% similarity this means that they are still different in some cases. The results of this property-test reflect that as well and we argue that this is also the reason why we see different performance of each when compared to SD.

An explanation why the time-driven approach seems to be closer to the SD dynamics is that in the event-driven approach we are dealing with discrete events, jumping from time to time instead of moving forward in time continuously as it happens conceptually in the time-driven approach. Time is also continuous in SD, thus it seems intuitively clear that a time-driven approach is closer to the SD implementation than the event-driven one - it seems valid to call our time-driven approach a continuous agent-based simulation approach. The implication is that depending on our intention, picking a time-driven or an event-driven implementation can and will make a statistical difference. If one is transferring an SD to an ABS model, one might consider to follow the time-driven approach as it seems to come much closer to the SD dynamics than the event-driven approach.

The reason that we are not reaching a coverage level up to and beyond 90% is rooted in the fundamental difference between SD and ABS: due to ABS' stochastic nature, its dynamics cannot match an SD exactly because it generates a *distribution* whereas the SD is deterministic. This enables ABS to explore and reveal paths which are not possible in deterministic SD. In the case of the SIR model, such an alternative path would be the immediate recovery of the single infected agent at the beginning without infecting any other agent. This is not possible in the SD case: in case there is 1 infected agent, the whole epidemic will unfold.

The difficulty of comparing dynamics between SD and ABS and the impracticality to compare them *exactly* was shown by [100] in the case of the SIR model, where the authors showed that it generates a bimodal distribution. Further, the authors report that a 70% envelope contains both the results of the SD and ABS implementation which is the reason why we chose a a 75% coverage as our initial guess, which has turned out to work well and is in accordance with the results of [100].

The question which remains is whether it actually even makes sense to compare the approaches to SD or even amongst each other - after all they can be seen as fundamentally different approaches. We can argue that they are qualitatively equal as [50] has already emphasised in a different study on comparing ABS and SD: although dynamics of ABS models are statistically different from SD ones, they look similar. The main difference is that ABS can contribute additional insight through revealing extra patterns due to its stochasticity, something not possible with SD. Thus in the end we simply have to accept that the respective coverage ratios are probably the closest we can get and that this is also the closest we can get in terms of validating our implementations against the original SD specification.

9.4 Discussion

In this chapter we have shown how to encode properties about simulation dynamics, generated by executing agents over time. This allowed to encode actual laws of the underlying SIR model in code and check them under random model parameters.

In the case of the time-driven implementation we saw that our initial assumption, that the invariants will hold for this implementation as well was wrong: QuickCheck revealed a *very* subtle bug in our implementation. Although the probability of this bug is very low, QuickCheck found it due to its random testing nature. This is another *strong* evidence, that random property-based testing is an *excellent* approach for testing Agent-Based Simulations. On the other hand, this bug revealed the difficulties in getting the subtle semantics of FRP right to implement pure functional ABS. This is a strong case that in general an event-driven approach should be preferred, which is also much faster and also not subject to the sampling issues discussed in Chapter 4.1.

Further, we showed that property-based testing also allows to compare two conceptually different implementations of the same underlying model with each other. This is indeed a perfect use case for property-based testing as it compares whole distributions and not only single runs using unit tests, making this another strong case for the use of property-based testing in ABS.

Finally, after having shown in previous chapters, that individual agent behaviour is correct up to some specification, in this chapter we also focused on validating the dynamics of the simulation with the original SD specification. By using QuickCheck, we showed how to connect both ABS implementations to the SD specification by deriving a property, based on the SD specification. This property is directly expressed in code and tested through generating random test cases with random agent populations and random model parameters.

Although our initial idea of matching the ABS implementation to the SD specifications has not worked out in an exact way, we still showed a way of formalizing and expressing these relations in code and testing them using QuickCheck. The results showed that the ABS implementation comes close to the original SD specification but does not match it exactly - it is indeed richer in its dynamics as [50, 100] have already shown. Our approach might work out better for a different model, which has a better behaved underlying specification than the bimodal SIR.

Discussion

In this part of the thesis we had an extensive look into the usefulness of randomised property-based testing in the development, specification and testing of pure functional ABS. We found property-based testing particularly well suited for ABS firstly due to ABS stochastic nature and second because we can formulate specifications, meaning we describe *what* to test instead of *how* to test. Also the deductive nature of falsification in property-based testing suits very well the constructive and often exploratory nature of ABS.

Indeed, we can see property-based testing not only as a post-implementation testing tool but as an extension to the development process where the developer engages in an interactive cycle of implementing agent and simulation behaviour and then immediately putting specifications into property-tests and running them. This approach of expressing specifications instead of special cases like in unit tests is arguably a much more natural approach to test-driven development in ABS development than relying only on unit tests.

In this part we only focused on the explanatory SIR model and ignored the exploratory Sugarcape. It is important to understand that testing of exploratory models is also possible through hypothesis testing. We discuss this approach in Appendix B in the context of validating our Sugarscape implementation.

We have only touched the tip of the iceberg and expect tremendous potential from applying property-based testing to different kind of models and in implementing ABS in general. Indeed, although property-based testing has its origins in Haskell, similar libraries have been developed for other languages e.g. Java, Python, C++ as well and we hope that our research will spark an interest in applying property-based testing to the established object-oriented languages in ABS as well.

For further insight into testing FRP we can directly leverage on the work of [123]. Another unique benefit of pure functional programming, *equational reasoning* was not investigated in this thesis as it was beyond the focus of this Ph.D. We expect that this technique is applicable to parts of our approach as well but leave this for further research.

Part V:

Discussion and Conclusion

Chapter 10

Discussion

This thesis started out by challenging the established views that *"[..] object-oriented programming to be a particularly natural development environment for Sugarscape specifically and artificial societies generally [..]"* [47] (p. 179) and that *agents map naturally to objects* [113]. As an alternative, a radical different approach to implementing ABS was proposed, using the pure functional programming paradigm. As language of choice, Haskell was motivated due to its pure functional features, matureness and increasing relevance to real-world applications.

The conjecture was that by using Haskell, one can directly transfer the promises made by pure functional programming to ABS as well, directly gaining a few highly important benefits. The relevance of each of these promises to ABS was already pointed out in the respective chapters and it is quite obvious that these benefits would clearly be of substantial value in ABS. The common baseline is that all those benefits support implementing ABS which are more likely to be correct, something of fundamental value in simulation.

1. The static strong type system allows to remove a substantial number and class of bugs at run time and if one programs careful one can even guarantee that no bugs as in crashes or exceptions will occur at run time. This is particularly the case for purely computational problems, without IO [1], as ABS almost always are.

2. Explicit handling and control of side effects delivers even more static guarantees at compile time and allows to deal with deterministic side effects (random-number streams, read-/write-only contexts, state) in a referential transparent way. In combination with strong static typing this allows to reduce logical bugs, subject to the domain of the problem, by dramatically reducing available, valid operations on data - after all stateful applications

[1] Obviously IO is involved in all ABS, otherwise the results are not observable. By purely computational, we mean the lack of IO in the agents, as fundamental part of the model specification, other than visualisation and exporting to an output file.

are a fact, the challenge is how to deal with state. As ABS is an inherently stateful problem due to agents and the environment, this should increase the correctness of an ABS implementation as well. Further, this should allow to produce an implementation which is guaranteed to be reproducible at compile time, where runs with same initial conditions *will* result in same dynamics.

3. Parallel and concurrent programming is claimed to be a lot easier, less painful and less error prone in functional programming in general and in Haskell in particular due to immutable data and the explicit handling of side effects. The concept of Software Transactional Memory, which allows to express a problem as a data-flow problem, is highly promising. Besides, data-parallel programming promises to speedup code without the need for changing any of the logic or types. This seemed to offer a straightforward way of speeding up ABS implementations either through data parallelism or concurrency. This has always been quite difficult to achieve in traditional object-oriented ABS and pure functional programming seems to offer a solution.

4. The data-centric declarative style, referential transparency and immutability of data makes testing substantially easier due to composability: functions can be easily tested in isolation from each other even if they involve side effects. This opens the door for randomised property-based testing, which intuitively seemed to be a perfect match to test ABS implementations which are almost always stochastic in nature.

The central question which needed to be answered first was *how* ABS could be done pure functionally, as there didn't exist any research which offered a systematic solution to that problem. More specifically, it was unclear how to represent agents, how to express agent identity, local agents state, changing behaviour and interactions amongst agents and the environment. After all, this is straightforward in object-oriented programming due to method calls and mutable shared state, encapsulated in objects. This thesis solution was to use arrowized FRP, both in the pure implementation of Yampa and the monadic version as in the library Dunai. Building on top of them allowed to implement pro-activity of agents, encapsulation of local agent state, an environment as shared mutable state and synchronous agent interactions based on an event-driven approach. The central concept behind these approaches are Signal Functions, generalised in Dunai to Monadic Stream Functions, which are implemented using closures and continuations, fundamental building blocks and concepts of pure functional programming. With theses techniques it became possible to implement time-driven models like the agent-based SIR as introduced in Chapter 2.2.1 and the highly complex event-driven Sugarscape model as introduced in Chapter 2.2.2. The fact that the developed concepts can manage the complexity of a full Sugarscape implementation is proof that they are suitable for most agent-based models, running on single machines.

Probably the biggest strength coming out of this implementation techniques is that we can guarantee reproducibility at compile time: given identical initial conditions, repeated runs of the simulation will lead to same outputs. This is of fundamental importance in simulation and addressed in the Sugarscape model: *"... when the sequence of random numbers is specified ex ante the model is deterministic. Stated yet another way, model output is invariant from run to run when all aspects of the model are kept constant including the stream of random numbers."* (page 28, footnote 16) - the pure functional approach can guarantee that *at compile time*. Further, we can enforce update semantics to a certain degree by dramatically restricting the state agents can manipulate. For example, in the time-driven approach the fact that the environment is provided as read-only and agents cannot interfere with other agents due to purity and complete lack of side effects, factually enforces the parallel update strategy: there is simply no way for agents to misbehave and violate semantics of updates by mutating state they are not allowed to.

The thesis then turned to the additional opportunities a pure functional approach offers, with focus on parallelism and concurrency and property-based testing. By applying compositional parallelism it was possible to speed up the non-monadic time-driven ABS by a substantial factor, without losing static guarantees about reproducibility. The same was not possible with a monadic approach, which motivated the transition to concurrency using Software Transactional Memory, where a substantial speedup was achieved in a monadic, event-driven implementation sacrificing purity. Still, by limiting the side effects to Software Transactional Memory guarantees that the differences of the dynamics of repeated runs come only from the side effects of concurrency and nothing else.

It was shown that adding data parallelism is easy and often requires simply swapping out a data structure or library function against its parallel version. Concurrency, although still hard, is less painful to address and add in a pure functional setting due to immutable data and explicit side effects. Further, the benefits of implementing concurrent ABS based on Software Transactional Memory has been shown at length in the respective chapter which underlines the strength of Haskell for concurrent ABS due to its strong guarantees about retry semantics.

The last step was to apply property-based testing to pure functional ABS, to test the hypothesis whether stochastic ABS and random property-based testing are a good match. Indeed, it has turned out to be a perfect match: it was possible to fully specify the agents behaviour of the time- and event-driven SIR model directly in code and verify them with QuickCheck. Further, other important techniques relevant for ABS like validation against specifications and comparisons of different implementations of the same model are easily done with property-based testing. Also, by showing how to perform hypothesis testing with property-based testing in the case of the Sugarscape model, we were able to show how to apply it to exploratory models which have no formal ground truth as well.

It was shown that functional programming in general gives much more control and checking of invariants due to the explicit handling of effects. Together with the strong static type system, testing code is in full, explicit control over the functionality it checks. Property-based testing in particular is a perfect match for testing ABS due to the stochastic nature of both and because it supports convenient expressing of specifications.

This concludes that the thesis successfully proved the usefulness of a pure functional approach, where the initial hypotheses and claims about the benefits as outlined in Chapter 1, being definitely fulfilled.

10.1 Drawbacks

Obviously nothing comes without drawbacks, and also our approach suffers from a few. We discuss them here and propose solutions where applicable.

10.1.1 Efficiency

As mentioned already in the discussions of the respective chapters, currently the performance of our approaches does not come close to imperative implementations. There are two main reasons for it: first, functional programming is known for being slower due to higher levels of abstraction, which are bought by slower code in general and second, updates are the main bottleneck due to immutable data requiring to copy the whole or subparts of a data structure in cases of a change. The first one is easily addressable through the use of data parallelism and concurrency as shown in Chapter 6 and 7. The second reason can be addressed by the use of linear types [15], which allow to annotate a variable with how often it is used within a function. From this a compiler could derive aggressive optimisations, potentially resulting in imperative-style performance but retaining the declarative nature of the code. We leave this for further research. Also, the use of Monad Transformer stacks has performance implications, which can be quite subtle. A possible optimisation we followed is the careful usage and re-ordering of lifts, using `lift (mapM ...)` instead of `mapM (lift ...)`, which potentially results in increased performance .

However, it was shown by various people [96, 142, 143] that Haskell does not necessarily have to be slow and that it is indeed possible to reach C speed in Haskell. The direction to do this is using the worker/wrapper transformation [57], clever combination of techniques with strict `foldl'` and data declaration with strictness annotation instead of lazy tuples and Stream Fusion [35, 103]. The problem is of course that to apply these techniques one needs to have deep knowledge of Haskell and its subtle details of lazy evaluation, making this a highly non-trivial task. Another problem is that those techniques seem only applicable in the context of a tight loop which crunches numbers of a list, thus it is not directly applicable in our case as we are clearly bound by the effectful computations: MSF and Monad Transformers are the limiting factor, not inner loops.

Concluding we can say that the current performance makes our approach not very attractive for real-world use *at the moment*. Also the fact that the sequential object-oriented implementation seems to outperform the concurrent and parallel implementations as well, seems to question our motivation as to why we are using pure functional programming and parallel computation at all. Still, the bad performance results do not invalidate our research as this thesis aim is not the development of high-performance pure functional implementations but rather exploring concepts of ABS in pure functional programming. Thus, this work is seen as a first step which needs to be developed further into something to be used in the real world. We think that our pure functional approach will not be able to reach Java performance but we hypothesise that it should be possible to come considerably closer, making it more applicable for real-world usage. We leave a deeper investigation of this problem for further research.

10.1.2 Space leaks

Haskell is notorious for its memory leaks due to lazy evaluation: data is only evaluated when required. Even for simple programs one can be hit hard by a serious space leak where unevaluated code pieces (thunks) build up in memory until they are needed, leading to dramatically increased memory usage. It is no surprise that our highly complex Sugarscape implementation initially suffered severely from space leaks, piling up about 40 MByte / second. In simulation this is a big issue, threatening the value of the whole implementation despite its other benefits. Due to the fact that simulations might run for a (very) long time or conceptually forever, one must make absolutely sure that the memory usage stays within reasonable bounds. As a remedy, Haskell allows to add so-called strictness pragmas to code modules, which force strict evaluation of all data even if it is not used.

Another memory leak was caused by selecting the wrong data structure for the environment, for which we initially used an immutable array. The problem is that in the case of an update the whole array is copied, causing memory leaks *and* a performance problem. We replaced it by an `IntMap` which uses integers as key and is internally implemented as a radix tree which allows for very fast lookups and inserts because whole sub-trees can be re-used.

10.1.3 Productivity and learning curve

A case study in [66] hints that simply by switching to a static type system alone does not gain anything and can even be detrimental. To be useful, it needs to have a certain level of abstractions like Haskells' type system has. Although such case studies have to be taken with care, there is also some truth in it: working in a statically strong type system prevents the developer from moving quickly and making quick and dirty changes. This can be both a benefit and a drawback: in general it prevents from breaking changes which show up at compile time but at the same time the whole program is much more rigid and a proper structure needs to be thought out and designed often up-front, slowing

down the process. However, this is a contribution of this thesis that it outlines exactly these structures within ABS so that implementers who want to use the same approach do not have to reinvent the wheel.

A more severe problem is that pure functional programming, especially Haskell, is seen as hard to learn with a steep learning curve, putting a high barrier to implementers picking up a pure functional approach to ABS. Thus, the lack of broad availability of Haskell expertise can be enough to pose a serious drawback even if the approach of this thesis seem to be desirable in a project.

10.1.4 Agent interactions

Synchronous, direct agent interactions *do* work in Haskell but they are cumbersome to get right when building from scratch. Further, as we pointed out in Chapter 7, it seems that our approach to synchronous direct agent interactions is not applicable to concurrency with Software Transactional Memory.

This leads to the fundamental conclusion that in models which require complex agent interactions in a potentially concurrent environment, we are hitting the limits of our pure functional approach. The reason for it is that we have a conceptual mismatch, as in such a setting, agents are more naturally represented using the Actor Model. The Actor Model, a model of concurrency, was initially conceived by Hewitt in 1973 [73] and refined later on [71], [72]. It was a major influence in designing the concept of agents and although there are important differences between actors and agents there are important similarities thus the idea to use actors to build ABS comes quite natural. An actor is a uniquely addressable entity which can do the following *in response to a message*:

- Send an arbitrary number of messages to other actors.
- Create an arbitrary number of actors.
- Define its own behaviour upon reception of the next message.

When comparing this definition to the one of agents we give in Chapter 2.2, it is clear that the Actor Model was quite influential to the development of the concept of agents in ABS, which borrowed it initially from Multi-Agent Systems [168]. Technically, it emphasises message-passing concurrency with share-nothing semantics (no implicitly shared state through side effects between agents), which maps nicely to functional programming concepts.

Indeed, the programming model of actors [1] was the inspiration for the functional programming language Erlang thus we argue that a true concurrent actor approach like Erlang is substantially more natural and much more performant especially in a concurrent setting. Further, we hypothesise that actor based ABS implementations might have a bright future as ABS tends to develop towards larger and larger, distributed, always-online simulations, for which Erlang is arguably perfectly suited. We have prototyped highly promising concurrent event-driven SIR and Sugarscape implementations in Erlang supporting our hypothesis. Unfortunately, an in-depth discussion is beyond the scope of this thesis and we leave this topic for further research.

10.2 Generalising research

We hypothesise that our research might be applicable to related fields as discussed below, which puts our contributions into a much broader perspective, giving it more impact.

10.2.1 Simulation in general

We showed at length in this thesis that purity in a simulation leads to reproducibility, which is of utmost importance in scientific computation. These insights are easily transferable to simulation software in general and might be of huge benefit there. Also, dependent types in ABS might be applicable to simulations in general due to the correspondence between equilibrium and totality, in use for hypotheses formulation and specifications formulation as pointed out in Appendix C. We think the use of Software Transactional Memory for implementing concurrent simulations is not only beneficial to ABS in particular but potentially applies to a wide range of simulation problems with a data-driven approach. Also, we demonstrated the usefulness of property-based testing in this thesis, which we believe can be a useful tool for simulation in general, to encode specifications, explore dynamics, test hypotheses and perform verification and validation directly in code.

10.2.2 System Dynamics

We have implemented a System Dynamics (SD) simulation in Appendix A. Although it is only an implementation of the SIR model, it shows clearly how an SD *specification* can directly be encoded in pure functional programming using arrowized FRP. The approach is simple enough to be generated automatically, is highly performant and could serve as language of choice for an SD simulation engine backend.

10.2.3 Discrete Event Simulation

We have already implemented basic mechanics of a Discrete Event Simulation (DES) in the case of the event-driven SIR. However, it would be very interesting to see whether we can express a DES model network directly in code, guaranteeing compatibility of elements statically at compile time. This should be possible due to the declarative nature of pure functional programming and the process-oriented nature of arrowized FRP. In the end both DES and FRP model data-flow networks which are fixed at compile time, so it should be a natural fit.

There also exists a parallel version of DES, called Parallel DES [54], which is concerned about the problem of running a DES in parallel, where the challenge is how to deal with sequential dependencies. Optimistic approaches run them in parallel, just like Software Transactional Memory does, and rolls back actions in case sequential orderings are violated. We hypothesise that in a pure

functional language with or without Software Transactional Memory it should be conceptually easier to implement such rollback semantics due to immutable persistent data structures and controlled side effects.

10.2.4 Recursive simulation

Due to the recursive nature of functional programming we believe that it is also a natural fit to implement recursive simulations as the one discussed in [58]. In recursive ABS agents are able to halt time and anticipate an arbitrary number of actions, compare their outcome, resume time and continue with a specifically chosen action with the best outcome. More precisely, an agent has the ability to run the simulation recursively a number of times where the number is not determined initially but can depend on the outcome of the recursive simulation. So recursive ABS gives each agent the ability to run the simulation locally from its point of view to project its actions into the future and change them in the present. Due to controlled side effects and referential transparency, combined with the recursive nature of pure functional programming, we think that implementing a recursive simulation in such a setting should be straightforward.

10.2.5 Multi-Agent Systems

The fields of Multi-Agent Systems (MAS) and ABS are closely related, where ABS has drawn much inspiration from MAS [164, 168]. In both fields, the concept of interacting agents is of fundamental importance, so we expect our research also to be applicable in parts to the field of MAS. Especially the work on static guarantees and property-based testing should be very useful there because MAS is very interested in correctness, verification and formally reasoning about a system and their agents, to show that a system follows a formal specification.

10.3 The Gintis case revisited

After having discussed all the benefits and drawbacks pure functional programming has to offer for ABS, we want to return to the Introduction chapter again, where we mentioned the work of Gintis [59]. To repeat, in this paper the author has claimed to have found a mechanism in bilateral decentralized exchange, which resulted in Walrasian General Equilibrium without the neo-classical approach of a tatonement process through a central auctioneer. Due to its high-impact result for economics, researchers [84] tried to reproduce the results independently but were not able to do so. After Gintis provided the code, it was found that there was a bug in his implementation which led to the unexpected results, which were seriously damaged through this error. The work of [48] investigated the specific nature of Gintis bugs and reported the following (Section *3.1.1 Deviations from the paper*, page 23):

1. An agent calculates the optimum inventory (or demand) according to a specific formula, which involves a factor λ which gets computed as $\lambda =$

$\frac{\sum_j p_{ij} x_{ij}}{\sum_j p_{ij} x_{ij}}$. What the specific computation denotes is not important here, what is important is that this computation is always 1. The authors [48] claim that the correct formula should have been: $\lambda = \frac{\sum_j p_{ij} x_{ij}}{\sum_j p_{ij} o_j}$ with o in the denominator instead of x.

2. There seems to be a discrepancy between the paper and the implementation describing the trading of agents. According to the paper the amount they exchange follows $x_{ig} \equiv \frac{p_{ig} x_{ig}}{p_{ih}}$. In the implementation however, Gintis uses $x_{ig} \equiv \frac{p_{ih} x_{ih}}{p_{ig}}$ but seems to do so inconsistently throughout his code, leading to wrong calculations.

3. Another bug was due to reversed ordering of events, where agents use old prices vectors due to missing recalculation which only happens in the next step.

4. There are a number of other subtle bugs, which, according to the authors [48] seemed to have no impact on the outcome of the simulation: slight miscalculation of the standard deviation for consumer and producer prices, crashing the program at runtime when dividing the agents unequally between different production goods due to a negative random number which in turn raises an exception in the Delphi random function.

After having hypothesised in the Introduction chapter that pure functional programming might be of help in implementing simulations which are more likely to be correct, the following questions arise:

Do the techniques introduced in this thesis transfer to this problem and model as well? The short answer is, yes they obviously do as Gintis models interacting individual agents which consume and produce and trade with each other. This could be implemented both with our time- and event-driven approach, with a better choice probably being the event-driven one due to agents directly trading with each other. However, for some models, our techniques introduced in Part II are too powerful and a much simpler approach would suffice to implement it. In general too much power should always be avoided (at least in programming and software engineering) because with much power comes much responsibility: more power requires to pay more attention to details and thus there is increased risk to make mistakes. Thus we should always look for the technique with minimal power but maximum abstraction, which solves our problem sufficiently. Gintis model is such an example: it resides in a very different domain of ABS, called Agent-Based Computational Economics (ACE).

ACE is a very important field, which picked up ABS as a method for research in recent years. The field of economics is an immensely vast and complex one with many facets to it, ranging from firms, to financial markets to whole economies of a country [22]. Today its very foundations rest on rational expectations, optimization and the efficient market hypothesis. The idea is that the

macroeconomics are explained by the micro foundations [32] defined through behaviour of individual agents. These agents are characterized by rational expectations, optimizing behaviour, having perfect information, equilibrium [51]. This approach to economics has come under heavy criticism in the last years for being not realistic, making impossible assumptions like perfect information, not being able to provide a process under which equilibrium is reached [94] and failing to predict crashes like the sub-prime mortgage crisis despite all the promises - the science of economics is perceived to be detached from reality [51]. ACE is a promise to repair the empirical deficit which (neo-classic) economics seem to exhibit by allowing to make more realistic, empirical assumptions about the agents which form the micro foundations. The ACE agents are characterized by bounded rationality, local information, restricted interactions over networks and out-of-equilibrium behaviour [49]. Works which investigate ACE as a discipline and discuss its methodology are [13, 17, 130, 147]. Tesfatsion [148] defines ACE as *[...] computational modelling of economic processes (including whole economies) as open-ended dynamic systems of interacting agents.*.

It is important to understand, that ACE utilises ABS different than the social sciences do. The latter one focuses more on agent interactions, as it is apparent in the Sugarscape model, where in ACE the rational and non-rational actions of individual agents are more important. Thus in many ACE models, the full power of the techniques introduced in Part II is not required. More specifically, agents of ACE models tend to have much simpler state, behave often in only one specific way, don't use synchronised direct agent interactions and are very rarely located in a spatial environment but focus more on network connections [60, 166] or avoid the notion of connectivity altogether. This is certainly the fact in the case of the Gintis model, as implemented by Gintis himself and the Java implementation of [48]:

- Even though it is an agent-based model and there is a clear notion of agents in the code, where they are represented as objects, the agents are very simple in terms of their structure. They are characterised by a few floating-point values with very simple behaviour that does not change over time.
- There are only very simple direct agent interactions, exchanging bids and asks, leading to an exchange.
- There is no environment whatsoever and a fully connected network is implicitly assumed because each agent can trade with all other agents.
- The only side effect necessary in this simulation is to draw random numbers.

Thus, a Haskell implementation in this domain can be achieved completely without the full force of our techniques: agents can be represented as simple data structures without SF or MSF, interactions handled by the kernel, no need for an environment, scheduling is handled directly by the kernel and side effects

are restricted to run in the `Rand` Monad only. Due to the substantial complexity of the Gintis model, we have refrained from attempting a full implementation of it in Haskell. However, to investigate this point more in depth we implemented a simulation [2] with so called Zero Intelligence traders [61], a well known and fundamental concept found in ACE. Our implementation was inspired by an implementation in Python [3] and is a satisfactory proof-of-concept underlining the fact that we can do pure functional ABS also in a robust way without the full power offered by our techniques. However, a full treatment of this is beyond the scope of this thesis and is left for further research.

Would the use of Haskell have prevented the bugs which Gintis made? The first and second bugs as reported above are an indication for a problematic use of indexed lists or arrays. Generally, one can say that independent of programming languages, indices into an array or list are *always* problematic because it is very easy to make very subtle mistakes by getting indices wrong. Pure functional programming would suffer the same problem *if* a similar technique would have been used. However, due to the fact that data in Haskell is immutable, an *idiomatic* implementation, following pure functional programming concepts, we would have probably not seen the use of lists but (Generalised) Algebraic Data Types, making this kind of bug much less likely. Further, due to the *declarative* nature of pure functional programming in Haskell, it is more likely that the implementer, reading through the existing code might have spotted the bug: there is much less noise in an idiomatic functional implementation than in imperative code, making code highly expressive and concise, thus less to read and more obvious.

The third bug is a very subtle logical error, regarding the semantics of the simulation. A pure functional implementation alone would not have helped avoiding this mistake. However, we think the declarative style and immutability of data in pure functional programming would have made the fact that an old version of data is used much more explicit thus we think it would have been easier to spot.

The last bugs are typical run-time errors, which would and do occur in Haskell as well, so a pure functional approach would not necessarily avoid these kind of bugs. However, such bugs can be avoided by using a dependently typed functional language like Idris [23] as such a type system allows to ensure in the types that only positive random numbers are drawn, and that the input ranges are strictly positive.

Thus, summarizing we hypothesise that with a clean and *idiomatic* pure functional implementation it would have been very likely that Gintis would have avoided or spot the bugs. Further, we claim that dependent types might have been of substantial benefit in the Gintis case, but we leave this for further research as it is beyond the scope of this thesis (see Appendix C).

[2] Available at `https://github.com/thalerjonathan/zerointelligence`

[3] `http://people.brandeis.edu/~blebaron/classes/agentfin/GodeSunder.html`

Would property-based tests have been of any help to prevent the bugs? We hypothesise that it is very likely that if Gintis would have applied rigorous unit and property-based testing to his model - which he should have, due to the high impact of his outcome - he would have found the inconsistencies and could have corrected them. The code of [48] contains numerous *checkInvariants* and assertions, which *are* properties expressed in code, thus immediately applicable to property-based testing. Further, due to the mathematical nature of the problem, many properties in the form of formulas can be found in the paper specification, which should be directly expressible using property-based and unit testing.

10.4 Do agents map naturally to objects?

One of the initial motivation of this thesis was the claim of North et al. [113] that *agents map naturally to objects*. At the very end of this thesis we want to re-visit this claim in the new light of our pure functional approach and finally answer the questions whether agents do map naturally to objects or not.

To give a satisfactory answer, we first need to re-examine the abstractions used in our pure functional approach, where the fundamental building blocks are *recursion* and *continuations*. In recursion a function is defined in terms of itself: in the process of computing the output it *might* call itself with changed input data. *Continuations* in turn, are functions which allow to encapsulate the execution state of a program by capturing local variables (known as closure) to pick up computation from that point later on by returning a new function.

As an explanatory example, we implement a continuation in Haskell which sums up integers and stores the sum locally as well as returning it as return value of the current step. First, we define the type of a general continuation, which takes a polymorphic type `i` as input and returns a polymorphic type `o` as output together with a new continuation

```
newtype Cont i o = C (i -> (o, Cont i o))
```

Then we implement an actual instance of a continuation with input and output types fixed to `Int`. It takes an initial value `x` and sums up the values passed to it. It returns `adder` with the new sum recursively as the new continuation.

```
adder :: Int -> Cont Int Int
adder x = C (\x' -> let s = x + x' in
                    (s, adder s))
```

To run a continuation, we implement a function which runs a given continuation for a given number of steps and always passes `x` as input and prints the continuations output.

```
runCont :: Int -> Int -> Cont Int Int -> IO ()
runCont 0 x _ = return ()
runCont n x (C cont) = do
```

```
  -- run the continuation with x as input, cont' is the new continuation
  let (x', cont') = cont x
  print x'
  runCont (n-1) x cont'
```

When actually running the continuation `adder` with an initial value of -1 for 10 steps and increments of 2, we get the following output:

```
> runCont 100 2 (adder (-1))
1
3
...
17
19
```

This explanatory example should make it clear that we can encapsulate arbitrary complex state, which is only visible and accessible from within the continuation. Further, with a continuation it becomes possible to switch behaviour dynamically, like switching from one mode of behaviour to another as in a state machine, simply by returning new functions which encapsulate the new behaviour. If no change in behaviour should occur, the continuation simply recursively returns itself with the new state captured as seen in the example above.

In fact, Yampas signal functions (SF) and Dunais Monadic Stream Functions (MSF) are nothing else than such continuations: SF are pure, without a monadic context, as can be seen in the implementation of the supersampling in Chapter 4.1.1; MSFs have an additional monadic context, which makes it possible to execute effectful computations within the continuation as can be seen in the implementation of the simulation stepping MSF in Chapter 4.3.1.

When looking closer at the example from above, it becomes clear that the continuation `adder` is non-terminating and is a potentially infinite structure, possible through lazy evaluation of Haskell, where the function `runCont` deconstructs / consumes / observes the output of this infinite structure step-by-step. This is related to the concepts of *corecursion* which are an even deeper underlying theory to continuations in general and our approach in particular. Technically speaking, corecursion is the dual to recursion where instead of starting with a data structure and reducing it stepwise until a base case is reached, corecursion starts with an initial value and iterates it ad infinity, producing an infinite data structure as output, enabled through lazy evaluation. Indeed, our agents produce infinite streams as output, potentially running for infinite time as it is implemented in the time-driven approach and to a lesser extent in the event-driven SIR. Now it is also easy to see why agents are not represented by pure functions: they have to change over time, which is precisely what pure functions cannot do as they can not rely on a context or history of a system.

The fact that we represented pure functional agents as SF and MSF is thus no coincidence and did not fall from the sky: they are in fact representations of *coalgebras*, which is the way to express dynamical systems in mathematics and

in pure functional programming: *"In general, dynamical systems with a hidden, black-box state space, to which a user only has limited access via specified (observer or mutator) operations, are coalgebras of various kinds"* [88]. Informally speaking a coalgebra is of the form $S \xrightarrow{c} \boxed{\text{... S ...}}$, with a state space S, a function c and a structured output (the box), which also contains the original domain S [86]. This is precisely what we see in the recursive type definition of `Cont` above.

This sounds very much like agents and indeed, coalgebras have been used (amongst others) in Process Theory to model communicating processes, a topic closely related to Actors and ABS; and objects in object-oriented programming [87]. It seems that we have found an underlying theory which connects both object-oriented programming and our pure functional ABS approach. This hints that it might be indeed the case that agents map naturally to objects, as we discuss below. We refer to [86, 88] for a proper, formal introduction and discussion of coalgebras as it is beyond the scope of this thesis.

When following the concepts of continuations and coalgebras from above and the viewpoint that *"... a closure is just a primitive form of object: the special case of an object with just one method."* [4] then one way to look at an SF and MSF is to see them as very simple immutable objects with a single method - the continuation - following a shared-nothing semantics.

Like in coalgebras and in continuations we have some internal state which can be altered through specified set of operations (events/inputs) and the effect can be observed through the output but not directly. This is particularly clear in the Sugarscape model, where agents have indeed a complex internal state, which changes only through events and is only observable through the output of data type *SugAgentObservable* as a result of sending an event. Further, we added a notion of agent identity, a clearly specified agent interface, local agent state and synchronous direct agent interactions through tagless final.

This interpretation and the fact that we seem to have achieved all the relevant concepts like encapsulation of local agent state and interactions purely functional, it seems that we indeed have to agree that agents do actually map naturally to objects. However, we argue we have to think objects in a much broader context than the one of existing object-oriented terminology as in the popular family of Java, C++ and Python. The fact that we can represent agents as objects also in a purely functional way, with sound underlying theories like coalgebras, leads us to the question, what actually constitutes objects and we have to be careful not to confuse the *concept* of objects with their *implementation* within a language.

There does not exist a commonly agreed upon definition of objects and object-oriented programming but rather a bunch of ideas and concepts [5]. It is agreed that the original ideas of objects and object-oriented programming were conceived by Kristen Nygaard, the inventor of Simula 67, the first object-

[4] https://www.tedinski.com/2018/11/20/message-oriented-programming.html
[5] `http://wiki.c2.com/?DefinitionsForOo`

oriented language [36] and Alan Kay, the inventor of SmallTalk, another pioneering object-oriented language in the early 70s [92].

Kristen Nygaard identified object-oriented programming by *"A program execution is regarded as a physical model, simulating the behaviour of either a real or imaginary part of the world."*, thus he puts the focus on the modelling aspect of the problem. Alan Kay claims to have coined the term *object-oriented* and defines it in more technical terms: everything is an object; every object is an instance of a class; the class holds the shared behaviour for its instances; objects communicate by sending and receiving messages. Alan Kay puts a strong emphasis on sending and receiving messages, with a shared-nothing interpretation. This becomes especially clear in a quote attributed to him: *"The big idea is "messaging" ... "I invented the term Object-Oriented and I can tell you I did not have C++ in mind."*.

So we see that the original *concepts* of objects and object-oriented programming vary considerably from how objects and object-oriented programming is *implemented* today in the family of popular object-oriented programming languages like Java, C++ and Python. Our approach is *one* answer to do that in a pure, strong statically typed language - Haskell. It can be seen as an object-centric approach, which *implements* a very simple *concept* of shared-nothing, immutable, pure functional objects.

Chapter 11

Conclusion

> A good Ph.D. thesis is asking more questions than it answers.
>
> J. G.

This thesis has shown at length *how* pure functional ABS is possible in a robust maintainable way and *why* it is of benefit. Each chapter gives strong evidence that our claim that pure functional programming has indeed its place in ABS is valid - especially a full implementation in Sugarscape is a highly complex and non-trivial matter and our highly structured and robust approach can be seen as proof of success. At the same time we conclude that it is yet too early to pick up this paradigm for ABS. We think that engineering a proper implementation of a complex ABS model takes substantial effort in pure functional programming due to different techniques required and due to its strong static type system. We believe that at the moment such an effort pays off only in cases of high-impact and large-scale simulations which results might have far-reaching consequences e.g. influence policy decisions. It is only there where the high requirements for reproducibility, robustness and correctness provided by functional programming are really needed. Still, we plan on distilling the developed techniques into a general purpose ABS library. This should allow implementing models much easier and quicker, making the pure functional programming approach an attractive alternative for prototyping, opening the direction for a broad use of functional programming in the field of ABS.

Further, this thesis deepened the understanding of the structure of agent based computation by expressing it in a pure functional context, leading to a deeper and more general understanding of agents, how they can be represented and whether they indeed map naturally to objects or not. This central question of whether they map naturally to objects or not, permeates the whole thesis and drives it forward. The last chapter gave an in-depth answer of that question and we see that objects indeed have their place in ABS but, as we have carefully explained, the question is what objects constitute. By approaching the problem from a pure functional perspective, we also arrived at a deeper and broader

understanding of objects, not only restricted to the traditional object-oriented definition of Java, C++ or Python. So we can say: *"Yes, agents map naturally to objects, but we have to be precise what constitutes objects."*. Whether they are newtypes, tuples, (G)ADT, pure functions, Monads, Comonads, Arrows, SF, MSF, Actors or object-oriented objects. They are all valid ways of representing agents with varying degree of abstraction, flexibility and power and they all come with their benefit and drawbacks, which have to be clearly understood together with the problem to solve. This thesis simply added a promising new tool to the family of existing ones and only time will tell whether this tool is indeed as valuable as hinted in this thesis.

11.1 Further Research

11.1.1 A general purpose library

For pure functional ABS to ever reach a larger audience and acceptance, it will need a lot of support, especially in the form of a well designed, easy to use, robust, correct, high quality Haskell library. Designing and developing such a library is research on its own as it needs to combine all the separate concepts introduced in this thesis into one code base. We hope that from this development further insights into ABS in general and pure functional ABS in particular will emerge, which can then be published to the community.

11.1.2 Dependent and types

We see this thesis 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 [3, 4, 154].

To the best of our knowledge, so far no research using dependent types in agent-based simulation exists at all. We aim 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 [23] 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 like Agda and Coq.

We hypothesise that dependent types could help ruling out even more classes of bugs at compile time and encode invariants and model specifications on the type level, which implies that we don't need to test them using property testing. This would allow the ABS community to reason about a model directly in code. We think that a promising approach is to follow the work of [24, 25, 52] in which the authors utilize Generalised Algebraic Data Types to implement an

indexed Monad, which allows to implement correct-by-construction software. We have already started to outline a few core principles in the Appendix C, but we conjecture that the true benefit is yet to be revealed.

11.1.3 Shallow encoding

Using arrowized FRP with pure signal functions as in Yampa and Monadic Stream Functions as in Dunai is only *one* approach to ABS. As we have shown at length in this thesis, although it is a robust and maintainable approach with highly valuable and promising benefits, there are other potential approaches left unexplored.

A standard approach to encode a problem in Haskell is by designing an embedded domain specific language (EDSL) for that (domain) problem and then implement the problem at hand in this specifically designed language. One has to distinguish between a *deep* and *shallow* encoding: in the former one, the EDSL is expressed "deeply" using direct data types and functions of the host language Haskell - our approach using SF and MSF is of this nature.

The shallow encoding approach on the other hand provides a language on top of the host language, often expressed using Algebraic Data Types or Generalised ADT which are used to encode a problem as a data structure where the execution of the problem is done by an interpreter which evaluates this data structure and translates it to the semantics of the host language. An example of a shallow embedded EDSL is a language for mathematical expressions consisting of constants, multiplication and subtractions, which are represented as syntax trees and then translated into Haskell multiplication and subtraction to arrive at the final result.

The main benefit of a shallow encoding is that, similar to tagless final, it allows to separate definition from implementation due to the separation of the EDSL and its interpretation. This allows to strictly separate implementation from specification, composes very well and thus should also be easy to test as mocking of parts is straightforward.

A popular and powerful approach to implement shallow EDSLs are so called *Freer Monads* [131]. One big advantage of them is that they free one from the ordering of effects imposed through Monad Transformers as already mentioned in Chapter 2.3. Unfortunately, Freer Monads seem to be somewhere around 30 times slower than the equivalent MTL code with a complexity of $O(n^2)$ [1]. For IO bound problems like business applications, this is not a big deal but as already mentioned, ABS are almost always never IO bound, so raw computation is all what counts and this is undoubtedly worse with Freer Monads. Even though there seem to be a better encoding possible, which is about 2x slower than MTL [2] the fact that we are already performance limited in our deep encoding makes the Free Monads approach not very appealing [3]. Still, we hypothesise that it

[1] `https://reasonablypolymorphic.com/blog/freer-monads/`
[2] `https://reasonablypolymorphic.com/blog/too-fast-too-free/`
[3] `https://reasonablypolymorphic.com/blog/freer-yet-too-costly/`

will be highly interesting and valuable research with a strong focus on formal reasoning about correctness.

11.1.4 Actor-based ABS

We hypothesise that the future of concurrent, agent interaction centric ABS lies in a functional, actor based concurrent approach. It required the full thesis to come to this conclusion thus this thesis only scratched the surface of the potential for actor based ABS as in Erlang or Cloud Haskell. We think that this topic deserves rigorous research as well as it is currently strongly neglected with only very few papers existing, which just scratch the surface. The idea seems compelling: functional programming with an actor language seems the way to do ABS in the future as it gives us almost all properties introduced in this thesis with the exception that performance is substantially better and agent interactions are a lot easier and naturally expressed.

There have been a few attempts on implementing the actor model in real programming languages where the most notable ones are Erlang and Scala. Both languages saw some use in implementing ABS with notable papers being [16, 39, 40, 138, 157]

Further, process calculi like the CSP [74] and π-Calculus [109] become applicable in the context of an Actor Model, which should allow to actually *prove* the correctness of a specification. Due to the direct mapping of the process calculi languages to Erlang we see this as highly promising.

11.1.5 A theory of pure functional agents

The work in this thesis is quite applied, with a software-engineering heavy approach and in general does not follow a grand underlying formal theory of (pure functional) agents. The reason for that is, that to the best knowledge of the author, such a theory does not (yet) exist and this thesis' rather practical and experimental approach of distilling out certain patterns and recurring concepts of ABS computation, shows that.

Still, after having investigated the underlying concepts behind SF and MSF, namely continuations in the discussions, it became clear, that the basic computational concepts of our pure functional approach is closely related to *coalgebras*. Another closely related concept here is *codata* [43]. Whereas in *data*, we construct a complex (opaque) data type from primitive types (e.g. data definitions in Haskell), in *codata* we extract components from a complex (opaque) data type through eliminators. Put shortly, data is concerned how values are constructed and codata is concerned how those values are used [43] - data types are directly observable, codata types are only indirectly observable through their interface. Again, this is conceptually closely related to object-oriented programming and the authors of [43] believe that codata is a common ground between functional and object-oriented programming. Yet another concept, which has close relations to objects are *comonads* [156]. They can be seen as a structure

with notions of context-dependent computation or streams, which ABS can be seen as of.

What is striking is that those concepts are conceptually closely related to object-oriented programming and have been used to model that paradigm, thus it seems that we have come around a full circle and found a potential link between our pure functional and the established object-oriented approach to ABS. This implies that there might lurk some programming paradigm agnostic, unified theory of agents as used in ABS somewhere behind these concepts. We believe that such a theory would help ABS to put itself on solid formal grounds, connect it to other areas of computation and other sciences and ultimately be a powerful tool to investigate certain aspects of models more rigorously and be of help in specification, validation and verification.

References

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

[2] Allen, C., and Moronuki, J. *Haskell Programming from First Principles*. Allen and Moronuki Publishing, July 2016. Google-Books-ID: 5FaXDAEACAAJ.

[3] Altenkirch, T., Danielsson, N. A., Loeh, A., and Oury, N. Pi Sigma: Dependent Types Without the Sugar. In *Proceedings of the 10th International Conference on Functional and Logic Programming* (Berlin, Heidelberg, 2010), FLOPS'10, Springer-Verlag, pp. 40–55.

[4] Altenkirch, T., Mcbride, C., and Mckinna, J. Why dependent types matter. In *http://www.e-pig.org/downloads/ydtm.pdf* (2005).

[5] Armstrong, J. Erlang. *Commun. ACM 53*, 9 (Sept. 2010), 68–75.

[6] Asta, S., Özcan, E., and Siebers, P.-O. An investigation on test driven discrete event simulation. In *Operational Research Society Simulation Workshop 2014 (SW14)* (Apr. 2014).

[7] Axelrod, R. The Convergence and Stability of Cultures: Local Convergence and Global Polarization. Working Paper, Santa Fe Institute, Mar. 1995.

[8] Axelrod, R. Advancing the Art of Simulation in the Social Sciences. In *Simulating Social Phenomena*. Springer, Berlin, Heidelberg, 1997, pp. 21–40.

[9] Axelrod, R. Chapter 33 Agent-based Modeling as a Bridge Between Disciplines. In *Handbook of Computational Economics*, L. T. a. K. L. Judd, Ed., vol. 2. Elsevier, 2006, pp. 1565–1584.

[10] Axelrod, R., and Tesfatsion, L. A Guide for Newcomers to Agent-Based Modeling in the Social Sciences. Staff General Research Papers Archive, Iowa State University, Department of Economics, Jan. 2006.

[11] Axtell, R., Axelrod, R., Epstein, J. M., and Cohen, M. D. Aligning simulation models: A case study and results. *Computational & Mathematical Organization Theory 1*, 2 (Feb. 1996), 123–141.

[12] Balci, O. Verification, Validation, and Testing. In *Handbook of Simulation*, J. Banks, Ed. John Wiley & Sons, Inc., 1998, pp. 335–393.

[13] Ballot, G., Mandel, A., and Vignes, A. Agent-based modeling and economic theory: where do we stand? *Journal of Economic Interaction and Coordination 10*, 2 (2015), 199–220.

[14] Beck, K. *Test Driven Development: By Example*, 01 edition ed. Addison-Wesley Professional, Boston, Nov. 2002.

[15] Bernardy, J.-P., Boespflug, M., Newton, R. R., Jones, S. P., and Spiwack, A. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages 2*, POPL (Dec. 2017), 1–29. arXiv: 1710.09756.

[16] Bezirgiannis, N. *Improving Performance of Simulation Software Using Haskells Concurrency & Parallelism*. PhD thesis, Utrecht University - Dept. of Information and Computing Sciences, 2013.

[17] Blume, L., Easley, D., Kleinberg, J., Kleinberg, R., and Tardos, E. Introduction to computer science and economic theory. *Journal of Economic Theory 156* (Mar. 2015), 1–13.

[18] Borshchev, A., and Filippov, A. From System Dynamics and Discrete Event to Practical Agent Based Modeling: Reasons, Techniques, Tools.

[19] Bostrom, N. Are We Living in a Computer Simulation? *The Philosophical Quarterly 53*, 211 (2003), 243–255.

[20] Botta, N., Mandel, A., and Ionescu, C. Time in discrete agent-based models of socio-economic systems. Documents de travail du Centre d'Economie de la Sorbonne 10076, Université Panthéon-Sorbonne (Paris 1), Centre d'Economie de la Sorbonne, 2010.

[21] Botta, N., Mandel, A., Ionescu, C., Hofmann, M., Lincke, D., Schupp, S., and Jaeger, C. A functional framework for agent-based models of exchange. *Applied Mathematics and Computation 218*, 8 (Dec. 2011), 4025–4040.

[22] Bowles, S., Edwards, R., and Roosevelt, F. *Understanding Capitalism: Competition, Command, and Change*, 3 edition ed. Oxford University Press, New York, Mar. 2005.

[23] Brady, E. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming 23*, 05 (2013), 552–593.

[24] Brady, E. Programming and Reasoning with Algebraic Effects and Dependent Types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2013), ICFP '13, ACM, pp. 133–144.

[25] Brady, E. State Machines All The Way Down - An Architecture for Dependently Typed Applications. Tech. rep., 2016.

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

[27] Burnstein, I. *Practical Software Testing: A Process-Oriented Approach*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[28] Church, A. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics 58*, 2 (Apr. 1936), 345–363.

[29] Claessen, K., and Hughes, J. QuickCheck - A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2000), ICFP '00, ACM, pp. 268–279.

[30] Claessen, K., and Hughes, J. Testing Monadic Code with QuickCheck. *SIGPLAN Not. 37*, 12 (Dec. 2002), 47–59.

[31] Clinger, W. D. Foundations of Actor Semantics. Tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA, 1981.

[32] Colell, A. M. *Microeconomic Theory*. Oxford University Press, 1995. Google-Books-ID: dFS2AQAACAAJ.

[33] Collier, N., and Ozik, J. Test-driven agent-based simulation development. In *2013 Winter Simulations Conference (WSC)* (Dec. 2013), pp. 1551–1559.

[34] Courtney, A., Nilsson, H., and Peterson, J. The Yampa Arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell* (New York, NY, USA, 2003), Haskell '03, ACM, pp. 7–18.

[35] Coutts, D., Leshchinskiy, R., and Stewart, D. Stream Fusion. From Lists to Streams to Nothing at All. In *Icfp'07* (2007).

[36] Dahl, O.-j. The birth of object orientation: the simula languages. In *Software Pioneers: Contributions to Software Engineering, Programming, Software Engineering and Operating Systems Series* (2002), Springer, pp. 79–90.

[37] De Jong, T. Suitability of Haskell for Multi-Agent Systems. Tech. rep., University of Twente, 2014.

[38] De Vries, E. An in-depth look at quickcheck-state-machine, Jan. 2019.

[39] Di Stefano, A., and Santoro, C. Using the Erlang Language for Multi-Agent Systems Implementation. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology* (Washington, DC, USA, 2005), IAT '05, IEEE Computer Society, pp. 679–685.

[40] Di Stefano, A., and Santoro, C. eXAT: an Experimental Tool for Programming Multi-Agent Systems in Erlang. Tech. rep., 2007.

[41] Diehl, S. WHAT I WISH I KNEW WHEN LEARNING HASKELL.

[42] Discolo, A., Harris, T., Marlow, S., Jones, S. P., and Singh, S. Lock Free Data Structures Using STM in Haskell. In *Proceedings of the 8th International Conference on Functional and Logic Programming* (Berlin, Heidelberg, 2006), FLOPS'06, Springer-Verlag, pp. 65–80.

[43] Downen, P., Sullivan, Z., Ariola, Z. M., and Peyton Jones, S. Codata in Action. In *Programming Languages and Systems* (2019), L. Caires, Ed., Lecture Notes in Computer Science, Springer International Publishing, pp. 119–146.

[44] Easley, D., and Kleinberg, J. *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge University Press, July 2010. Google-Books-ID: 8xf1nAEACAAJ.

[45] Epstein, J. M. Chapter 34 Remarks on the Foundations of Agent-Based Generative Social Science. In *Handbook of Computational Economics*, L. T. a. K. L. Judd, Ed., vol. 2. Elsevier, 2006, pp. 1585–1604.

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

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

[48] Evensen, P., and Märdin, M. An Extensible and Scalable Agent-Based Simulation of Barter Economics. Master's thesis, Chalmers University of Technology, Göteborg, 2010.

[49] Farmer, J. D., and Foley, D. The economy needs agent-based modelling. *Nature 460*, 7256 (Aug. 2009), 685–686.

[50] Figueredo, G. P., Siebers, P.-O., Owen, M. R., Reps, J., and Aickelin, U. Comparing Stochastic Differential Equations and Agent-Based Modelling and Simulation for Early-Stage Cancer. *PLOS ONE 9*, 4 (Apr. 2014), e95150.

[51] Focardi, S. Is economics an empirical science? If not, can it become one? *Frontiers in Applied Mathematics and Statistics 1* (2015), 7.

[52] Fowler, S., and Brady, E. Dependent Types for Safe and Secure Web Programming. In *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages* (New York, NY, USA, 2014), IFL '13, ACM, pp. 49:49–49:60.

[53] Fujimoto, R. M. Parallel Discrete Event Simulation. *Commun. ACM 33*, 10 (Oct. 1990), 30–53.

[54] Fujimoto, R. M., Bagrodia, R., Bryant, R. E., Chandy, K. M., Jefferson, D., Misra, J., Nicol, D., and Unger, B. Parallel discrete event simulation: The making of a field. In *2017 Winter Simulation Conference (WSC)* (Dec. 2017), pp. 262–291.

[55] Galán, J. M., Izquierdo, L. R., Izquierdo, S. S., Santos, J. I., del Olmo, R., López-Paredes, A., and Edmonds, B. Errors and Artefacts in Agent-Based Modelling. *Journal of Artificial Societies and Social Simulation 12*, 1 (2009), 1.

[56] Gamma, E., Helm, R., Johnson, R., Vlissides, J., and Booch, G. *Design Patterns: Elements of Reusable Object-Oriented Software*, 1 edition ed. Addison-Wesley Professional, Oct. 1994.

[57] Gill, A., and Hutton, G. The Worker/Wrapper Transformation. *J. Funct. Program. 19*, 2 (Mar. 2009), 227–251.

[58] Gilmer, Jr., J. B., and Sullivan, F. J. Recursive Simulation to Aid Models of Decision Making. In *Proceedings of the 32Nd Conference on Winter Simulation* (San Diego, CA, USA, 2000), WSC '00, Society for Computer Simulation International, pp. 958–963.

[59] Gintis, H. The Emergence of a Price System from Decentralized Bilateral Exchange. *Contributions in Theoretical Economics 6*, 1 (2006), 1–15.

[60] Glasserman, P., and Young, P. Contagion in Financial Networks. SSRN Scholarly Paper ID 2681392, Social Science Research Network, Rochester, NY, Oct. 2015.

[61] Gode, D. K., and Sunder, S. Allocative Efficiency of Markets with Zero-Intelligence Traders: Market as a Partial Substitute for Individual Rationality. *Journal of Political Economy 101*, 1 (1993), 119–137.

[62] Gregory, J. *Game Engine Architecture, Third Edition*. Taylor & Francis, Mar. 2018.

[63] Grief, I., and Greif, I. SEMANTICS OF COMMUNICATING PARALLEL PROCESSES. Tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA, 1975.

[64] Gurcan, O., Dikenelli, O., and Bernon, C. A generic testing framework for agent-based simulation models. *Journal of Simulation 7*, 3 (Aug. 2013), 183–201.

[65] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE 79*, 9 (Sept. 1991), 1305–1320.

[66] Hanenberg, S. An Experiment About Static and Dynamic Type Systems: Doubts About the Positive Impact of Static Type Systems on Development Time. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (New York, NY, USA, 2010), OOPSLA '10, ACM, pp. 22–35. event-place: Reno/Tahoe, Nevada, USA.

[67] Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2005), PPoPP '05, ACM, pp. 48–60.

[68] Harris, T., and Peyton Jones, S. Transactional memory with data invariants.

[69] Heindl, A., and Pokam, G. Modeling Software Transactional Memory with AnyLogic. In *Proceedings of the 2Nd International Conference on Simulation Tools and Techniques* (ICST, Brussels, Belgium, Belgium, 2009), Simutools '09, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 10:1–10:10.

[70] Henderson, P. Functional Geometry. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming* (New York, NY, USA, 1982), LFP '82, ACM, pp. 179–187.

[71] Hewitt, C. What Is Commitment? Physical, Organizational, and Social (Revised). In *Coordination, Organizations, Institutions, and Norms in Agent Systems II*, P. Noriega, J. Vázquez-Salceda, G. Boella, O. Boissier, V. Dignum, N. Fornara, and E. Matson, Eds., no. 4386 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 293–307.

[72] Hewitt, C. Actor Model of Computation: Scalable Robust Information Systems. *arXiv:1008.1459 [cs]* (Aug. 2010). arXiv: 1008.1459.

[73] Hewitt, C., Bishop, P., and Steiger, R. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (San Francisco, CA, USA, 1973), IJCAI'73, Morgan Kaufmann Publishers Inc., pp. 235–245.

[74] Hoare, C. A. R. *Communicating Sequential Processes*. Prentice-Hall, 1985. Google-Books-ID: tpZLQgAACAAJ.

[75] Hudak, P., Courtney, A., Nilsson, H., and Peterson, J. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming*, J. Jeuring and S. L. P. Jones, Eds., no. 2638 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 159–187.

[76] Hudak, P., Hughes, J., Peyton Jones, S., and Wadler, P. A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (New York, NY, USA, 2007), HOPL III, ACM, pp. 12–1–12–55.

[77] Hudak, P., and Jones, M. Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity. Research Report YALEU/DCS/RR-1049, Department of Computer Science, Yale University, New Haven, CT, Oct. 1994.

[78] Hughes, J. Why Functional Programming Matters. *Comput. J. 32*, 2 (Apr. 1989), 98–107.

[79] Hughes, J. Generalising Monads to Arrows. *Sci. Comput. Program. 37*, 1-3 (May 2000), 67–111.

[80] Hughes, J. Programming with Arrows. In *Proceedings of the 5th International Conference on Advanced Functional Programming* (Berlin, Heidelberg, 2005), AFP'04, Springer-Verlag, pp. 73–129.

[81] Hughes, J. QuickCheck Testing for Fun and Profit. In *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages* (Berlin, Heidelberg, 2007), PADL'07, Springer-Verlag, pp. 1–32.

[82] Hutton, G. A Tutorial on the Universality and Expressiveness of Fold. *J. Funct. Program. 9*, 4 (July 1999), 355–372.

[83] Hutton, G. *Programming in Haskell*. Cambridge University Press, Aug. 2016. Google-Books-ID: 1xHPDAAAQBAJ.

[84] Ionescu, C., and Jansson, P. Dependently-Typed Programming in Scientific Computing. In *Implementation and Application of Functional Languages* (Aug. 2012), R. Hinze, Ed., Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 140–156.

[85] Jackson, M. O. *Social and Economic Networks*. Princeton University Press, 2008. Google-Books-ID: IpA9LwEACAAJ.

[86] Jacobs, B. *Introduction to Coalgebra*. Cambridge University Press, 2017. Google-Books-ID: tApQDQAAQBAJ.

[87] Jacobs, B., and Poll, E. Coalgebras and Monads in the Semantics of Java. *Theor. Comput. Sci. 291*, 3 (Jan. 2003), 329–349.

[88] Jacobs, B., and Rutten, J. A Tutorial on (Co)Algebras and (Co)Induction. *EATCS Bulletin 62* (1997), 62–222.

[89] Jankovic, P., and Such, O. Functional Programming and Discrete Simulation. Tech. rep., 2007.

[90] Jones, M. P. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text* (Berlin, Heidelberg, 1995), Springer-Verlag, pp. 97–136.

[91] Jones, S. P. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction* (2002), Press, pp. 47–96.

[92] Kay, A. C. The Early History of Smalltalk. In *The Second ACM SIGPLAN Conference on History of Programming Languages* (New York, NY, USA, 1993), HOPL-II, ACM, pp. 69–95. event-place: Cambridge, Massachusetts, USA.

[93] Kermack, W. O., and McKendrick, A. G. 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.

[94] Kirman, A. *Complex Economics: Individual and Collective Rationality.* Routledge, London ; New York, NY, July 2010.

[95] Kiselyov, O. Typed Tagless Final Interpreters. In *Generic and Indexed Programming: International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, J. Gibbons, Ed., Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 130–174.

[96] kqr, C. On Competing with C Using Haskell, June 2017.

[97] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM 21*, 7 (July 1978), 558–565.

[98] Lipovaca, M. *Learn You a Haskell for Great Good!: A Beginner's Guide.* No Starch Press, Apr. 2011. Google-Books-ID: QesxXj_ecD0C.

[99] Lysenko, M., and D'Souza, R. M. A Framework for Megascale Agent Based Model Simulations on Graphics Processing Units. *Journal of Artificial Societies and Social Simulation 11*, 4 (2008), 10.

[100] Macal, C. M. To Agent-based Simulation from System Dynamics. In *Proceedings of the Winter Simulation Conference* (Baltimore, Maryland, 2010), WSC '10, Winter Simulation Conference, pp. 371–382.

[101] Macal, C. M. Everything you need to know about agent-based modelling and simulation. *Journal of Simulation 10*, 2 (May 2016), 144–156.

[102] MacLennan, B. J. *Functional Programming: Practice and Theory*. Addison-Wesley, Jan. 1990. Google-Books-ID: JqhQAAAAMAAJ.

[103] Mainland, G. Haskell Beats C Using Generalized Stream Fusion.

[104] Marlow, S. *Parallel and Concurrent Programming in Haskell*. O'Reilly, 2013. Google-Books-ID: k0W6AQAACAAJ.

[105] Marlow, S., Newton, R., and Peyton Jones, S. A Monad for Deterministic Parallelism. In *Proceedings of the 4th ACM Symposium on Haskell* (New York, NY, USA, 2011), Haskell '11, ACM, pp. 71–82. event-place: Tokyo, Japan.

[106] McMillan, K. L. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[107] Meyer, R. Event-Driven Multi-agent Simulation. In *Multi-Agent-Based Simulation XV* (May 2014), Lecture Notes in Computer Science, Springer, Cham, pp. 3–16.

[108] Michaelson, G. *An Introduction to Functional Programming Through Lambda Calculus*. Courier Corporation, 2011. Google-Books-ID: gKvwPtvsSjsC.

[109] Milner, R. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, May 1999. Google-Books-ID: k2tfQgAACAAJ.

[110] Moggi, E. Computational Lambda-calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science* (Piscataway, NJ, USA, 1989), IEEE Press, pp. 14–23.

[111] Nilsson, H., Courtney, A., and Peterson, J. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (New York, NY, USA, 2002), Haskell '02, ACM, pp. 51–64.

[112] North, M. J., Collier, N. T., Ozik, J., Tatara, E. R., Macal, C. M., Bragen, M., and Sydelko, P. Complex adaptive systems modeling with Repast Simphony. *Complex Adaptive Systems Modeling 1*, 1 (Mar. 2013), 3.

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

[114] Odell, J. Objects and Agents Compared. *Journal of Object Technology 1*, 1 (May 2002), 41–53.

[115] Okasaki, C. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1999.

[116] Onggo, B. S. S., and Karatas, M. Test-driven simulation modelling: A case study using agent-based maritime search-operation simulation. *European Journal of Operational Research 254* (2016), 517–531.

[117] Ormerod, P., and Rosewell, B. Validation and Verification of Agent-Based Models in the Social Sciences. In *Epistemological Aspects of Computer Simulation in the Social Sciences*, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Oct. 2006, pp. 130–140.

[118] O'Sullivan, B., Goerzen, J., and Stewart, D. *Real World Haskell*, 1st ed. O'Reilly Media, Inc., 2008.

[119] Paterson, R. A New Notation for Arrows. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2001), ICFP '01, ACM, pp. 229–240.

[120] Perez, I. Back to the Future: Time Travel in FRP. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell* (New York, NY, USA, 2017), Haskell 2017, ACM, pp. 105–116.

[121] Perez, I. *Extensible and Robust Functional Reactive Programming*. Doctoral Thesis, University Of Nottingham, Nottingham, Oct. 2017.

[122] Perez, I., Baerenz, M., and Nilsson, H. Functional Reactive Programming, Refactored. In *Proceedings of the 9th International Symposium on Haskell* (New York, NY, USA, 2016), Haskell 2016, ACM, pp. 33–44.

[123] Perez, I., and Nilsson, H. Testing and Debugging Functional Reactive Programming. *Proc. ACM Program. Lang. 1*, ICFP (Aug. 2017), 2:1–2:27.

[124] Perfumo, C., Sönmez, N., Stipic, S., Unsal, O., Cristal, A., Harris, T., and Valero, M. The Limits of Software Transactional Memory (STM): Dissecting Haskell STM Applications on a Many-core Environment. In *Proceedings of the 5th Conference on Computing Frontiers* (New York, NY, USA, 2008), CF '08, ACM, pp. 67–78.

[125] Pierce, B. C., Amorim, A. A. d., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., Tolmach, A., and Yorgey, B. *Programming Language Foundations*. Software Foundations series, volume 2. Electronic textbook, May 2018.

[126] Polhill, J. G., Izquierdo, L. R., and Gotts, N. M. The Ghost in the Model (and Other Effects of Floating Point Arithmetic). *Journal of Artificial Societies and Social Simulation 8*, 1 (2005), 1.

[127] Popper, K. R. *The Logic of Scientific Discovery*. Psychology Press, 2002. Google-Books-ID: Yq6xeupNStMC.

[128] Porter, D. E. 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.

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

[130] Richiardi, M. Agent-based Computational Economics. A Short Introduction. LABORatorio R. Revelli Working Papers Series 69, LABORatorio R. Revelli, Centre for Employment Studies, 2007.

[131] Rivas, E., and Jaskelioff, M. Notions of Computation as Monoids. *arXiv:1406.4823 [cs, math]* (May 2014). arXiv: 1406.4823.

[132] Robinson, S. *Simulation: The Practice of Model Development and Use*. Macmillan Education UK, Sept. 2014. Google-Books-ID: Dtn0oAEACAAJ.

[133] Runciman, C., Naylor, M., and Lindblad, F. Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (New York, NY, USA, 2008), Haskell '08, ACM, pp. 37–48.

[134] Schelling, T. Dynamic models of segregation. *Journal of Mathematical Sociology 1* (1971).

[135] Schneider, O., Dutchyn, C., and Osgood, N. Towards Frabjous: A Two-level System for Functional Reactive Agent-based Epidemic Simulation. In *Proceedings of the 2Nd ACM SIGHIT International Health Informatics Symposium* (New York, NY, USA, 2012), IHI '12, ACM, pp. 785–790.

[136] Sculthorpe, N., and Nilsson, H. Safe Functional Reactive Programming Through Dependent Types. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2009), ICFP '09, ACM, pp. 23–34.

[137] Shavit, N., and Touitou, D. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1995), PODC '95, ACM, pp. 204–213.

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

[139] Siebers, P.-O., and Aickelin, U. Introduction to Multi-Agent Simulation. *arXiv:0803.3905 [cs]* (Mar. 2008). arXiv: 0803.3905.

[140] Sorokin, D. Aivika 3: Creating a Simulation Library based on Functional Programming, 2015.

[141] Steinhart, E. Theological Implications of the Simulation Argument. *Ars Disputandi: The Online Journal for Philosophy of Religion 10* (2010), 23–37.

[142] Stewart, D. Haskell as fast as C: working at a high altitude for low level performance, Apr. 2008.

[143] Stolarek, J. Haskell as fast as C: A case study, Apr. 2013.

[144] Stump, A. *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA, 2016.

[145] Sulzmann, M., and Lam, E. Specifying and Controlling Agents in Haskell. Tech. rep., 2007.

[146] Sweeney, T. The Next Mainstream Programming Language: A Game Developer's Perspective. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2006), POPL '06, ACM, pp. 269–269.

[147] Tesfatsion, L. Agent-Based Computational Economics: A Constructive Approach to Economic Theory. Handbook of Computational Economics, Elsevier, 2006.

[148] Tesfatsion, L. Agent-based Computational Economics (ACE) - Growing Economies from the Bottom Up. Tech. rep., May 2017.

[149] Thaler, J., Altenkirch, T., and Siebers, P.-O. Pure Functional Epidemics: An Agent-Based Approach. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages* (New York, NY, USA, 2018), IFL 2018, ACM, pp. 1–12. event-place: Lowell, MA, USA.

[150] Thaler, J., and Siebers, P.-O. The Art Of Iterating: Update-Strategies in Agent-Based Simulation.

[151] Thaler, J., and Siebers, P.-O. A Tale Of Lock-Free Agents - The potential of Software Transactional Memory in parallel Agent-Based Simulation. *ACM Trans. Model. Comput. Simul. Under Review*, Under Review (Oct. 2018), 21.

[152] Thaler, J., and Siebers, P.-O. The agents new cloth? Towards Pure Functional Programming in ABS.

[153] Thaler, J., and Siebers, P.-O. Show Me Your Properties! The Potential Of Property-Based Testing In Agent-Based Simulation.

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

[155] Turing, A. M. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society s2-42*, 1 (1937), 230–265.

[156] Uustalu, T., and Vene, V. The Essence of Dataflow Programming. In *Central European Functional Programming School* (2006), Z. Horváth, Ed., Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 135–167.

[157] Varela, C., Abalde, C., Castro, L., and Gulías, J. On Modelling Agent Systems with Erlang. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Erlang* (New York, NY, USA, 2004), ERLANG '04, ACM, pp. 65–70.

[158] Vendrov, I., Dutchyn, C., and Osgood, N. D. Frabjous A Declarative Domain-Specific Language for Agent-Based Modeling. In *Social Computing, Behavioral-Cultural Modeling and Prediction*, W. G. Kennedy, N. Agarwal, and S. J. Yang, Eds., no. 8393 in Lecture Notes in Computer Science. Springer International Publishing, Apr. 2014, pp. 385–392.

[159] Wadler, P. The Essence of Functional Programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1992), POPL '92, ACM, pp. 1–14.

[160] Wadler, P. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text* (London, UK, UK, 1995), Springer-Verlag, pp. 24–52.

[161] Wadler, P. How to Declare an Imperative. *ACM Comput. Surv. 29*, 3 (Sept. 1997), 240–263.

[162] Wald, A. Sequential Tests of Statistical Hypotheses. In *Breakthroughs in Statistics: Foundations and Basic Theory*, S. Kotz and N. L. Johnson, Eds., Springer Series in Statistics. Springer New York, New York, NY, 1992, pp. 256–298.

[163] Weaver, I. Replicating Sugarscape in NetLogo, Oct. 2009.

[164] Weiss, G. *Multiagent Systems*. MIT Press, Mar. 2013. Google-Books-ID: WY36AQAAQBAJ.

[165] Wilensky, U., and Rand, W. *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NETLogo*. MIT Press, 2015.

[166] Wilhite, A. Economic Activity on Fixed Networks. Handbook of Computational Economics, Elsevier, 2006.

[167] Winograd-Cort, D., and Hudak, P. Wormholes: Introducing Effects to FRP. In *Proceedings of the 2012 Haskell Symposium* (New York, NY, USA, 2012), Haskell '12, ACM, pp. 91–104.

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

[169] Zeigler, B. P., Praehofer, H., and Kim, T. G. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, Jan. 2000. Google-Books-ID: REzmY-OQmHuQC.

Appendices

Appendix A

Correct-by-construction System Dynamics

In this appendix we develop a correct-by-construction implementation of the SIR System Dynamics simulation. Note that the constant parameters *populationSize, infectedCount, contactRate, infectivity, illnessDuration* are defined globally and omitted for clarity.

A.1 Deriving the implementation

Computing the dynamics of a SD model happens by taking the integrals as shown in Chapter 9.3 and integrating over time. So conceptually we treat our SD model as a continuous function which is defined over time between 0 and infinity which outputs the values of each stock at each point in time. In the case of the SIR model we have 3 stocks: Susceptible, Infected and Recovered. Thus we start our implementation by defining the output of the SD function: for each time step we have the values of the 3 stocks:

```
type SIRStep = (Time, Double, Double, Double)
```

Next, we use a signal function to define the top-level function of the SIR SD simulation. It has no input, because an SD system is only defined by its own terms and parameters without external input and has as output the *SIRStep*.

```
sir :: SF () SIRStep
```

An SD model is fundamentally built on feedback: the values at time t depend on the previous step. Thus, we introduce feedback in which we feed the last step into the next step. Yampa provides the *loopPre* :: $c \rightarrow SF\ (a, c)\ (b, c) \rightarrow SF\ a\ b$ function for that. It takes an initial value c and a feedback signal function which receives the input a and the previous (or initial) value of the feedback c and has to return the output b and the new feedback value c. *loopPre* then returns

simply a signal function from a to b with the feedback happening transparent in the feedback signal function. The initial feedback value is the initial state of the SD model at $t = 0$. Further, we define the type of the feedback signal function.

```
sir = loopPre (0, initSus, initInf, initRec) sirFeedback
  where
    initSus = populationSize - infectedCount
    initInf = infectedCount
    initRec = 0

    sirFeedback :: SF ((), SIRStep) (SIRStep, SIRStep)
```

The next step is to implement the feedback signal function. As input we get *(a, c)* where a is the empty tuple () because an SD simulation has no input, and c is the fed back *SIRStep* from the previous (initial) step. With this we have all relevant data so we can implement the feedback function. We first match on the tuple inputs and construct a signal function using *proc*:

```
    sirFeedback = proc (_, (_, s, i, _)) -> do
```

Now we define the flows which are *infection rate* and *recovery rate*. The formulas for both of them can be seen in the differential equations of Chapter 9.3. This directly translates into Haskell code:

```
infectionRate = (i * contactRate * s * infectivity) / populationSize
recoveryRate  = i / illnessDuration
```

Next we need to compute the values of the three stocks, following the integral formulas of Chapter 9.3. For this we need the *integral* function of Yampa which integrates over a numerical input using the rectangle rule. Adding initial values can be achieved with the `^<<` operator of arrowized programming. This directly translates into Haskell code:

```
s' <- (initSus+) ^<< integral -< (-infectionRate)
i' <- (initInf+) ^<< integral -< (infectionRate - recoveryRate)
r' <- (initRec+) ^<< integral -< recoveryRate
```

We also need the current time of the simulation. For this we use Yampas *time* function:

```
      t <- time -< ()
```

Now we only need to return the output and the feedback value. Both types are the same thus we simply duplicate the tuple:

```
      returnA -< dupe (t, s', i', r')

    dupe :: a -> (a, a)
    dupe a = (a, a)
```

Δt	Susceptibles	Infected	Recovered	Max Infected
1.0	17.52	26.87	955.61	419.07 @ t = 51
0.5	23.24	25.63	951.12	399.53 @ t = 47.5
0.1	27.56	24.27	948.17	384.71 @ t = 44.7
$1e-2$	28.52	24.11	947.36	381.48 @ t = 43.97
$1e-3$	28.62	24.08	947.30	381.16 @ t = 43.9
AnyLogic	28.625	24.081	947.294	381.132 @ t = 44

Table A.1: Results running the simulation with varying Δt until $t = 100$ with a population Size $N = 1{,}000$, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ and initially 1 infected agent.

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

```
runSD :: Time -> DTime -> [SIRStep]
runSD t dt = embed sir ((), steps)
  where
    steps = replicate (floor (t / dt)) (dt, Nothing)
```

A.2 Results

Although we have translated our model specifications directly into code we still need to validate the dynamics and test the system for its numerical behaviour under varying Δt. This is necessary because numerical integration, which happens in the *integral* function, can be susceptible to instability and errors. Yampa implements the simple rectangle-rule of numerical integration which requires very small Δt to keep the errors minimal and arrive at sufficiently good results.

We have run the simulation with varying Δt to show what difference varying Δt can have on the simulation dynamics. We ran the simulations until $t = 100$ with a population Size $N = 1{,}000$, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ and initially 1 infected agent. For comparison we looked at the final values at $t = 100$ of the susceptible, infected and recovered stocks. Also, we compare the time and the value when the infected stock reaches its maximum. The values are reported in the Table A.1.

As additional validation we added the results of a System Dynamics simulation in AnyLogic Personal Learning Edition 8.3.1, which is reported in the last row in the Table A.1. Also we provided a visualisation of the AnyLogics simulation dynamics in Figure A.1a. By comparing the results in Table A.1 and

the dynamics in Figure A.1a to A.1b we can conclude that we arrive at the same dynamics, validating the correctness of our simulation also against an existing, well-known and established System Dynamics software package.

A.3 Discussion

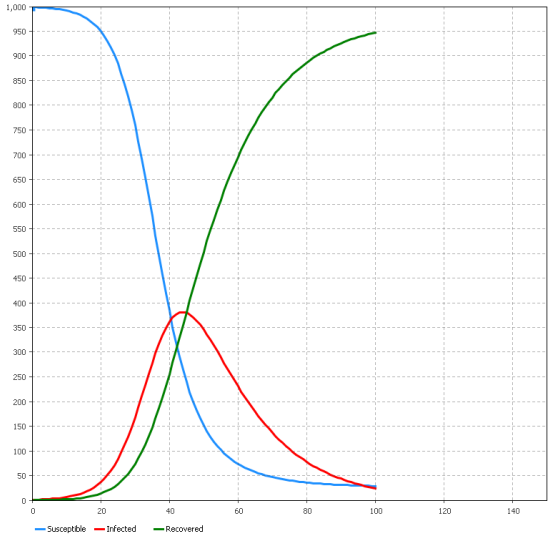
We claim that our implementation is correct-by-construction because the code *is* the model specification - we have closed the gap between the specification and its implementation. Also we can guarantee that no non-deterministic influences can happen in this implementation due to the strong static type system of Haskell. This guarantees that repeated runs of the simulation will always result in the exact same dynamics given the same initial parameters, something of fundamental importance in System Dynamics.

Further, we showed the influence of different Δt and validated our implementation against the industry-strength System Dynamics simulation package AnyLogic Personal Learning Edition 8.3.1 where we could match our results with the one of AnyLogic, proving the correctness of our system also on the dynamics level.

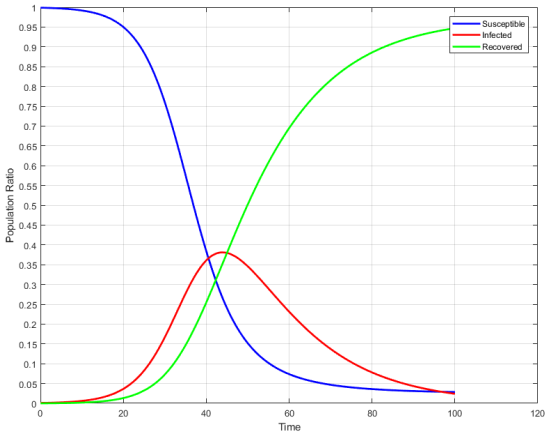
Obviously the numerical well behaviour depends on the integral function which uses the rectangle rule. We showed that for the SIR model and small enough Δt, the rectangle rule works well enough. Still it might be of benefit if we provide more sophisticated numerical integration like Runge-Kutta methods. We leave this for further research.

The key strength of System Dynamic simulation packages is their visual representation which allows non-programmers to "draw" System Dynamics models and simulate them. We believe that one can auto-generate Haskell code using our approach to implement System Dynamics from such diagrams but leave this for further research.

Also we are very well aware that due to the vast amount of visual simulation packages available for System Dynamics, there is no big need for implementing such simulations directly in code. Still we hope that our pure functional approach with Functional Reactive Programming might spark an interest in approaching the implementation of System Dynamics from a new perspective, which might lead to pure functional back-ends of visual simulation packages, giving them more confidence in their correctness.



(a) System Dynamics simulation of SIR compartment model in AnyLogic Personal Learning Edition 8.3.1. 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 until $t = 100$.



(b) Dynamics of the SIR compartment model following this implementation. 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 until $t = 150$. Plot generated from data by this Haskell implementation using Octave.

Figure A.1: Visual comparison of the SIR SD dynamics generated by AnyLogic and our implementation in Haskell.

A.4 Full Implementation

```
populationSize :: Double
populationSize = 1000

infectedCount :: Double
infectedCount = 1

contactRate :: Double
contactRate = 5

infectivity :: Double
infectivity = 0.05

illnessDuration :: Double
illnessDuration = 15

type SIRStep = (Time, Double, Double, Double)

sir :: SF () SIRStep
sir = loopPre (0, initSus, initInf, initRec) sirFeedback
  where
    initSus = populationSize - infectedCount
    initInf = infectedCount
    initRec = 0

    sirFeedback :: SF ((), SIRStep) (SIRStep, SIRStep)
    sirFeedback = proc (_, (_, s, i, _)) -> do
      let infectionRate = (i * contactRate * s * infectivity) / populationSize
          recoveryRate  = i / illnessDuration

      t <- time -< ()

      s' <- (initSus+) ^<< integral -< (-infectionRate)
      i' <- (initInf+) ^<< integral -< (infectionRate - recoveryRate)
      r' <- (initRec+) ^<< integral -< recoveryRate

      returnA -< dupe (t, s', i', r')

    dupe :: a -> (a, a)
    dupe a = (a, a)

runSD :: Time -> DTime -> [SIRStep]
runSD t dt = embed sir ((), steps)
  where
    steps = replicate (floor (t / dt)) (dt, Nothing)
```

Appendix B

Validating Sugarscape in Haskell

In this chapter we look at how property-based testing can be made of use to verify the *exploratory* Sugarscape model [47] as already introduced in Chapter 2.2.2. Whereas in the chapters on testing the explanatory SIR model we had an analytical solution, the fundamental difference in the exploratory Sugarscape model is that none such analytical solutions exist. This raises the question, which properties we can actually test in such a mode.

The answer lies in the very nature of exploratory models: they exist to explore and understand phenomena of the real world. Researchers come up with a model to explain the phenomena and (hopefully) with a few questions and *hypotheses* about the emergent properties. The actual simulation is then used to test and refine the hypotheses. Indeed, descriptions, assumptions and hypotheses of varying formal degree abound in the Sugarscape model. Examples are: *the carrying capacity becomes stable after 100 steps; when agents trade with each other, after 1000 steps the standard deviation of trading prices is less than 0.05; when there are cultures, after 2700 steps either one culture dominates the other or both are equally present.*

We show how to use property-testing to formalise and check such hypotheses. For this purpose we undertook a full *verification* of our implementation [1] from Chapter 2.2.2. We validated it against the book [47] and a NetLogo implementation [163] [2].

[1] The code can be accessed freely from `https://github.com/thalerjonathan/phd/tree/master/public/towards/SugarScape/sequential`

[2] `https://www2.le.ac.uk/departments/interdisciplinary-science/research/replicating-sugarscape`, Note that lending didn't properly work in their NetLogo code and that they didn't implement Combat

B.1 Property-based hypothesis testing

The property we test for is whether *the emergent property / hypothesis under test is stable under replicated runs* or not. To put it more technical, we use QuickCheck to run multiple replications with the same configuration but with different random-number streams and require that all tests pass. During the verification process we have derived and implemented property-tests for the following hypotheses:

1. Disease Dynamics where all recover - When disease are turned on, if the number of initial diseases is 10, then the population is able to rid itself completely from all disease within 100 ticks.

2. Disease Dynamics where a minority recovers - When disease are turned on, if the number of initial diseases is 25, the population is not able to rid itself completely from all diseases within 1,000 ticks.

3. Trading Dynamics - When trading is enabled, the trading prices stabilise after 1,000 ticks with the standard deviation of the prices having dropped below 0.05.

4. Cultural Dynamics - When having two cultures, red and blue, after 2,700 ticks, either the red or the blue culture dominates or both are equally strong. If they dominate they make up 95% of all agents, if they are equally strong they are both within 45% - 55%.

5. Inheritance Gini Coefficient - According to the book, when agents reproduce and can die of age then inheritance of their wealth leads to an unequal wealth distribution measured using the Gini Coefficient *averaging* at 0.7.

6. Carrying Capacity - When agents don't mate nor can die from age, due to the environment, there is an *average* maximum carrying capacity of agents the environment can sustain. The capacity should be reached after 100 ticks and should be stable from then on.

7. Terracing - When resources regrow immediately, after a few steps the simulation becomes static. Agents will stay on their terraces and will not move any more because they have found the best spot due to their behaviour. About 45% will be on terraces and 95% - 100% are static, not moving any more.

The hypotheses and their validation is described more in-depth in the section B.2 below.

B.1.1 Implementation

To start with, we implement a custom data generator to produce output from a Sugarscape simulation. The generator takes the number of ticks and the

scenario with which to run the simulation and returns a list of outputs, one for each tick.

```
sugarscapeUntil :: Int                -- ^ Number of ticks to run
                -> SugarScapeScenario -- ^ Scenario to run
                -> Gen [SimStepOut]   -- ^ Output of each step
sugarscapeUntil ticks params = do
  -- create a random-number generator
  g <- genStdGen
  -- initialise the simulation state with the given random-number generator
  -- and the scenario
  let simState = initSimulationRng g params
  -- run the simulation with the given state for number of ticks
  return (simulateUntil ticks simState)
```

Using this generator, we can very conveniently produce Sugarscape data within a QuickCheck *Property*. Depending on the problem, we can generate only a single run or multiple replications, in case the hypothesis is assuming *averages*. To see its use, we show the implementation of the *Disease Dynamics (1)* hypothesis.

```
prop_disease_allrecover :: Property
prop_disease_allrecover = property (do
  -- after 100 ticks...
  let ticks = 100
  -- ... given Animation V-1 parameter configuration ...
  let params = mkParamsAnimationV_1
  -- ... from 1 sugarscape simulation ...
  aos <- sugarscapeLast ticks params
  -- ... counting all infected agents ...
  let infected = length (filter (==False)) map (null . sugObsDiseases . snd) aos
  -- ... should result in all agents to be recovered
  return (cover 100 (infected == 0) "Diseases all recover" True)
```

From the implementation it becomes clear, that this hypothesis states that the property has to hold *for all* replications. The *Inheritance Gini Coefficient (5)* hypothesis on the other hand assumes that the Gini Coefficient *averages* at 0.7. We cannot average over replicated runs of the same property thus we generate multiple replications of the Sugarscape data within the property and employ a two-sided t-test with a 95% confidence to test the hypothesis:

```
prop_gini :: Int      -- ^ Number of replications
          -> Double   -- ^ Confidence of the t-test
          -> Property
prop_gini repls confidence = property (do
  -- after 1000 ticks...
  let ticks = 1000
  -- ... the gini coefficient should average at 0.7 ...
  let expGini = 0.7
  -- ... given the Figure III-7 parameter configuration ...
  let params = mkParamsFigureIII_7
  -- ... from 100 replications ...
  gini <- vectorOf repls (genGiniCoeff ticks params)
  -- on a two-tailed t-test with given confidence
  return (tTestSamples TwoTail expGini (1 - confidence) gini)
```

B.1.2 Running the tests

As already pointed out in Part IV, QuickCheck by default runs up to 100 test cases of a property and if all evaluate to *True* the property-test succeeds. On the other hand, QuickCheck will stop at the first test case which evaluates to *False* and marks the whole property-test as failed, no matter how many test cases got through already. For this reason we have used *cover* with an expected percentage of 100, meaning that we expect all tests to fall into the coverage class. This allows us to emulate failure with QuickCheck reporting the actual percentage of passed test cases.

Due to the duration even 1,000 ticks can take to compute, to get a first estimate of our hypotheses tests within reasonable time, we reduce the number of maximum successful replications required to 10 and when doing t-tests 10 replications are run there as well.

```
SugarScape Tests
  Disease Dynamics All Recover:      OK (29.25s)
    +++ OK, passed 10 tests (100% Diseases all recover).

  Disease Dynamics Minority Recover: OK (536.00s)
    +++ OK, passed 10 tests (100% Diseases no recover).

  Trading Dynamics:                  OK (149.33s)
    +++ OK, passed 10 tests (70% Prices std less than 5.0e-2).
    Only 70% Prices std less than 5.0e-2, but expected 100%

  Cultural Dynamics:                 OK (996.84s)
    +++ OK, passed 10 tests (50% Cultures dominate or equal).
    Only 50% Cultures dominate or equal, but expected 100%

  Carrying Capacity:   OK (988.20s)
    +++ OK, passed 10 tests (90% Carrying capacity averages at 204.0).
    Only 90% Carrying capacity averages at 204.0, but expected 100%

  Terracing:           OK (280.59s)
    +++ OK, passed 10 tests (80% Terracing is happening).
    Only 80% Terracing is happening, but expected 100%

  Inheritance Gini:    OK (7232.59s)
    +++ OK, passed 0 tests (0% Gini coefficient averages at 0.7).
    Only 0% Gini coefficient averages at 0.7, but expected 100%
```

How to deal with the failure of hypotheses is obviously highly model specific. A first approach is to increase the number of replications to run to 100 to get a more robust estimate of the failure rate. If the failure rate stays within reasonable ranges then one can arguably assume that the hypothesis is valid for sufficiently enough cases. On the other hand, if the failure rate escalates, then

it is reasonable to deem the hypothesis invalid and refine it or even abandon it altogether.

With the exception of the Gini Coefficient, we accept the failure rate of the hypotheses we presented here and deem them sufficiently valid for the task at hand. In case of the Gini Coefficient, none of the replication was successful, which makes it obvious that it does *not* average at 0.7. Thus the hypothesis as stated in the book does not hold and is invalid. One way to deal with it would be to simply delete it. Another, more constructive approach, is to keep it but require all replications to fail by marking it with *expectFailure* instead of *property*. In this way an invalid hypothesis is marked explicitly and acts as documentation and also as regression test.

B.2 Hypotheses and test cases

In this section we briefly describe the process of validating our Sugarscape implementation against the specification of the Sugarscape book [47] and the work of [163].

B.2.1 Terracing

Our implementation reproduces the terracing phenomenon as described in the book and as can be seen in the NetLogo implementation as well. We implemented a property-test in which we measure the closeness of agents to the ridge: counting the number of same-level sugars cells around them and if there is at least one lower then they are at the edge. If a certain percentage is at the edge then we accept terracing. The question is just how much, which we estimated from tests and resulted in 45%. Also, in the terracing animation the agents actually never move which is because sugar immediately grows back thus there is no incentive for an agent to actually move after it has moved to the nearest largest cite in can see. Therefore we test that the coordinates of the agents after 50 steps are the same for the remaining steps.

B.2.2 Carrying capacity

Our simulation reached a steady state (variance < 4 after 100 steps) with a mean around 182. Epstein reported a carrying capacity of 224 (page 30) and the NetLogo implementations' [163] carrying capacity fluctuates around 205 which both are significantly higher than ours. Something was definitely wrong - the carrying capacity has to be around 200 (we trust in this case the NetLogo implementation and deem 224 an outlier).

After inspection of the NetLogo model we realised that we implicitly assumed that the metabolism range is *continuously* uniformly randomized between 1 and 4 but this seemed not what the original authors intended: in the NetLogo model there were a few agents surviving on sugar level 1 which was never the case in ours as the probability of drawing a metabolism of exactly 1 is practically zero

when drawing from a continuous range. We thus changed our implementation to draw a discrete value as the metabolism.

This partly solved the problem, the carrying capacity was now around 204 which is much better than 182 but still a far cry from 210 or even 224. After adjusting the order in which agents apply the Sugarscape rules, by looking at the code of the NetLogo implementation, we arrived at a comparable carrying capacity of the NetLogo implementation: agents first make their move and harvest sugar and only after this the agents metabolism is applied (and ageing in subsequent experiments).

For regression-tests we implemented a property-test which tests that the carrying capacity of 100 simulation runs lies within a 95% confidence interval of a 210 mean. These values are quite reasonable to assume, when looking at the NetLogo implementation - again we deem the reported Carrying Capacity of 224 in the Book to be an outlier / part of other details we don't know.

One lesson learned is that even such seemingly minor things like continuous vs. discrete or order of actions an agent makes, can have substantial impact on the dynamics of a simulation.

B.2.3 Wealth distribution

By visual comparison we validated that the wealth distribution (page 32-37) becomes strongly skewed with a histogram showing a fat tail, power-law distribution where very few agents are very rich and most of the agents are quite poor. We compute the skewness and kurtosis of the distribution which is around a skewness of 1.5, clearly indicating a right skewed distribution and a kurtosis which is around 2.0 which clearly indicates the 1st histogram of Animation II-3 on page 34. Also we compute the Gini coefficient and it varies between 0.47 and 0.5 - this is accordance with Animation II-4 on page 38 which shows a gini-coefficient which stabilises around 0.5 after. We implemented a regression-test testing skewness, kurtosis and gini-coefficients of 100 runs to be within a 95% confidence interval of a two-sided t-test using an expected skewness of 1.5, kurtosis of 2.0 and gini-coefficient of 0.48.

B.2.4 Migration

With the information provided by [163] we could replicate the waves as visible in the NetLogo implementation as well. Also we propose that a vision of 10 is not enough yet and shall be increased to 15 which makes the waves very prominent and keeps them up for much longer - agent waves are travelling back and forth between both Sugarscape peaks. We have not implemented a regression-test for this property as we couldn't come up with a reasonable straightforward approach to implement it.

B.2.5 Pollution and diffusion

With the information provided by [163] we could replicate the pollution behaviour as visible in the NetLogo implementation as well. We have not implemented a regression-test for this property as we couldn't come up with a reasonable straightforward approach to implement it.

B.2.6 Mating

We could not replicate Figure III-1 - our dynamics first raised and then plunged to about 100 agents and go then on to recover and fluctuate around 300. This findings are in accordance with [163], where they report similar findings - also when running their NetLogo code we find the dynamics to be qualitatively the same.

Also at first we weren't able to reproduce the cycles of population sizes. Then we realised that our agent-behaviour was not correct: agents which died from age or metabolism could still engage in mating before actually dying - fixing this to the behaviour, that agents which died from age or metabolism will not engage in mating solved that and produces the same swings as in [163]. Although our bug might be obvious, the lack of specification of the order of the application of the rules is an issue in the SugarScape book.

B.2.7 Inheritance

We couldn't replicate the findings of the Sugarscape book regarding the Gini coefficient with inheritance. The authors report that they reach a gini coefficient of 0.7 and above in Animation III-4. Our Gini coefficient fluctuated around 0.35. Compared to the same configuration but without inheritance (Animation III-1) which reached a Gini coefficient of about 0.21, this is indeed a substantial increase - also with inheritance we reach a larger number of agents of around 1,000 as compared to around 300 without inheritance. The Sugarscape book compares this to chapter II, Animation II-4 for which they report a Gini coefficient of around 0.5 which we could reproduce as well. The question remains, why it is lower (lower inequality) with inheritance?

The baseline is that this shows that inheritance indeed has an influence on the inequality in a population. Thus we deemed that our results are qualitatively the same as the make the same point. Still there must be some mechanisms going on behind the scenes which are unspecified in the original Sugarscape.

B.2.8 Cultural dynamics

We could replicate the cultural dynamics of AnimationIII-6 / Figure III-8: after 2700 steps either one culture (red / blue) dominates both hills or each hill is dominated by a different ulture. We wrote a test for it in which we run the simulation for 2.700 steps and then check if either culture dominates with a ratio of 95% or if they are equal dominant with 45%. Because always a few

agents stay stationary on sugarlevel 1 (they have a metabolism of 1 and cant see far enough to move towards the hills, thus stay always on same spot because no improvement and grow back to 1 after 1 step), there are a few agents which never participate in the cultural process and thus no complete convergence can happen. This is accordance with [163].

B.2.9 Combat

Unfortunately [163] didn't implement combat, so we couldn't compare it to their dynamics. Also, we weren't able to replicate the dynamics found in the Sugarscape book: the two tribes always formed a clear battlefront where some agents engage in combat e.g. when one single agent strays too far from its tribe and comes into vision of the other tribe it will be killed almost always immediately. This is because crossing the sugar valley is costly: this agent wont harvest as much as the agents staying on their hill thus will be less wealthy and thus easier killed off. Also retaliation is not possible without any of its own tribe anywhere near.

We didn't see a single run where an agent of an opposite tribe "invaded" the other tribes hill and ran havoc killing off the entire tribe. We don't see how this can happen: the two tribes start in opposite corners and quickly occupy the respective sugar hills. So both tribes are acting on average the same and also because of the number of agents no single agent can gather extreme amounts of wealth - the wealth should rise in both tribes equally on average. Thus it is very unlikely that a super-wealthy agent emerges, which makes the transition to the other side and starts killing off agents at large. First: a super-wealthy agent is unlikely to emerge, second making the transition to the other side is costly and also low probability, third the other tribe is quite wealthy as well having harvested for the same time the sugar hill, thus it might be that the agent might kill a few but the closer it gets to the center of the tribe the less like is a kill due to retaliation avoidance - the agent will simply get killed by others.

Also it is unclear in case of AnimationIII-11 if the R rule also applies to agents which get killed in combat. Nothing in the book makes this clear and we left it untouched so that agents who only die from age (original R rule) are replaced. This will lead to a near-extinction of the whole population quite quickly as agents kill each other off until 1 single agent is left which will never get killed in combat because there are no other agents who could kill it - instead it will enter an infinite die and reborn cycle thanks to the R rule.

B.2.10 Spice

The book specifies for AnimationIV-1 a vision between 1-10 and a metabolism between 1-5. The last one seems to be quite strange because the maximum sugar / spice an agent can find is 4 which means that agents with metabolism of either 5 will die no matter what they do because the can never harvest enough to satisfy their metabolism. When running our implementation with this configuration the number of agents quickly drops from 400 to 105 and continues to slowly

degrade below 90 after around 1000 steps. The implementation of [163] used a slightly different configuration for AnimationIV-1, where they set vision to 1-6 and metabolism to 1-4. Their dynamics stabilise to 97 agents after around 500+ steps. When we use the same configuration as theirs, we produce the same dynamics. Also it is worth nothing that our visual output is strikingly similar to both the book AnimationIV-1 and [163].

B.2.11 Trading

For trading we had a look at the NetLogo implementation of [163]: there an agent engages in trading with its neighbours *over multiple rounds* until either MRSs cross over or no trade has happened anymore. Because [163] were able to exactly replicate the dynamics of the trading time-series we assume that their implementation is correct. We think that the fact that an agent interact with its neighbours over multiple rounds is made not very clear in the book. The only hint is found on page 102: *"This process is repeated until no further gains from trades are possible."* which is not very clear and does not specify exactly what is going on: does the agent engage with all neighbours again? is the ordering random? Another hint is found on page 105 where trading is to be stopped after MRS cross-over to prevent an infinite loop. Unfortunately this is missing in the Agent trade rule T on page 105. Additional information on this is found in footnote 23 on page 107. Further on page 107: *"If exchange of the commodities will not cause the agents' MRSs to cross over then the transaction occurs, the agents recompute their MRSs, and bargaining begins anew."*. This is probably the clearest hint that trading could occur over multiple rounds.

We still managed to exactly replicate the trading-dynamics as shown in the book in Figure IV-3, Figure IV-4 and Figure IV-5. The book is also pretty specific on the dynamics of the trading-prices standard-deviation: on page 109 the authors specify that at t=1000 the standard deviation will have always fallen below 0.05 (Figure IV-5), thus we implemented a property-test which tests for exactly that property. Unfortunately we didn't reach the same magnitude of the trading volume where ours is much lower around 50 but it is equally erratic, so we attribute these differences to other missing specifications or different measurements because the price-dynamics match that well already so we can safely assume that our trading implementation is correct.

According to the book, Carrying Capacity (Animation II-2) is increased by Trade (page 111/112). To check this it is important to compare it not against AnimationII-2 but a variation of the configuration for it where spice is enabled, otherwise the results are not comparable because carrying capacity changes substantially when spice is on the environment and trade turned off. We could replicate the findings of the book: the carrying capacity increases slightly when trading is turned on. Also does the average vision decrease and the average metabolism increase. This makes perfect sense: trading allows genetically weaker agents to survive which results in a slightly higher carrying capacity but shows a weaker genetic performance of the population.

According to the book, increasing the agent vision leads to a faster conver-

gence towards the (near) equilibrium price (page 117/118/119, Figure IV-8 and Figure IV-9). We could replicate this behaviour as well.

According to the book, when enabling R rule and giving agents a finite life span between 60 and 100 this will lead to price dispersion: the trading prices will not converge around the equilibrium and the standard deviation will fluctuate wildly (page 120, Figure IV-10 and Figure IV-11). We could replicate this behaviour as well.

The Gini coefficient should be higher when trading is enabled (page 122, Figure IV-13) - We could replicate this behaviour.

Finite lives with sexual reproduction lead to prices which don't converge (page 123, Figure IV-14). We could reproduce this as well but it was important to re-set the parameters to reasonable values: increasing number of agents from 200 to 400, metabolism to 1-4 and vision to 1-6, most important the initial endowments back to 5-25 (both sugar and spice) otherwise hardly any mating would happen because the agents need too much wealth to engage (only fertile when have gathered more than initial endowment). What was kind of interesting is that in this scenario the trading volume of sugar is substantially higher than the spice volume - about 3 times as high.

From this part, we didn't implement: Effect of Culturally Varying Preferences, page 124 - 126, Externalities and Price Disequilibrium: The effect of Pollution, page 126 - 118, On The Evolution of Foresight page 129 / 130.

B.2.12 Diseases

We were able to exactly replicate the behaviour of Animation V-1 and Animation V-2: in the first case the population rids itself of all diseases (maximum 10) which happens pretty quickly, in less than 100 ticks. In the second case the population fails to do so because of the much larger number of diseases (25) in circulation. We used the same parameters as in the book. The authors of [163] could only replicate the first animation exactly and the second was only deemed "good". Their implementation differs slightly from ours: In their case a disease can be passed to an agent who is immune to it - this is not possible in ours. In their case if an agent has already the disease, the transmitting agent selects a new disease, the other agent has not yet - this is not the case in our implementation and we think this is unreasonable to follow: it would require too much information and is also unrealistic. We wrote regression tests which check for animation V-1 that after 100 ticks there are no more infected agents and for animation V-2 that after 1000 ticks there are still infected agents left and they dominate: there are more infected than recovered agents.

B.3 Discussion

In this appendix we showed how to use QuickCheck to formalise and check hypotheses about an *exploratory* agent-based model, in which no ground truth exists. Due to ABS stochastic nature in general it became obvious that to get

a good measure of a hypotheses validity we need to emulate failure using the *cover* function of QuickCheck. This allowed us to show that the hypotheses we have presented are sufficiently valid for the task at hand and can indeed be used for expressing and formalising emergent properties of the model and also as regression tests within a TDD cycle.

Appendix C

The equilibrium-totality correspondence

In the property-tests of Chapter 9.1 and 8.2 we limited the time an individual simulation is run to a random range between 0 and 50. In this context, the decision to do so was for practical reasons to guarantee termination as both the event- and time-driven implementations would run forever if no time- and/or event-limit is specified [1].

However, restricting the simulation to a time- and/or event-limit is not necessary in a correct SIR implementation because it *will* reach an equilibrium *within finite time* at which point the simulation can be terminated. This is the case as soon as there are no more infected agents: intuitively this is clear because only infected agents can lead to infections of susceptible agents. They make the transition to recovered after having gone through the infection phase and will never go back to susceptible, thus also the pool of susceptible agents is finite. The infected agents themselves *will* recover within finite time. Thus we can conclude that a correct implementation of the SIR model must enter an equilibrium, a steady state in finite time.

Using this informal reasoning, we change the property-test from Chapter 9.1 to encode this property implicitly.

```
prop_sir_invariants :: Positive Int    -- ^ beta, contact rate
                    -> Property        -- ^ gamma, infectivity in range (0,1)
                    -> Positive Double -- ^ delta, illness duration
                    -> [SIRState]      -- ^ population
                    -> Property
prop_sir_invariants
    (Positive cor) (P inf) (Positive ild) as  = property (do
  -- CHANGED: run the SIR simulation with UNRESTRICTED time
  ret <- genSimulationSIR ss cor inf ild 0
  -- CHANGED: take data as long as not in equilibrium
```

[1] Note that the event-driven implementation would terminate if the event-queue is empty but in the case of the SIR this will never be the case due to susceptible agents keep scheduling *MakeContact*, resulting in an infinite stream of events.

```
let ret' = takeWhile ((>0).snd3.snd) ret
-- check invariants and return result
return (sirInvariants (length as) ret')
```

Unfortunately this code is dangerous: generally, we cannot distinguish between a very long or infinitely running simulation. It might be the case that there is a bug in our implementation, violating the property that all infected agents eventually recover, in which case *takeWhile* might run forever. This means that we cannot write a property-test, which could tell us whether this property holds or not for both our time- and event-driven implementations - it is in general not computable. Obviously this is nothing new and was established in the 1930s through the work of Turing [155]. The question is now: what can we do about it?

One solution is to abandon the power of general recursion and Turing-completeness and switch to a different kind of pure functional programming language in which programs can be checked for totality by the compiler. These languages have a different kind of type system, called dependent types. Generally, dependent types add the following concepts to pure functional programming:

1. Totality and termination - A total function terminates with a well-typed result or produces a non-empty finite prefix of a well-typed infinite result in finite time [26]. In dependently typed languages, which abandon Turing completeness, this can be checked at compile time under certain circumstances.

2. Types are first-class citizen - In dependently typed languages, types can depend on any *values*, and can be *computed* at compile time, which makes them first-class citizen. This allows to compute the return type of a function depending on its input values. Note that this requires totality, otherwise type checking would be not decidable and potentially non-terminating.

3. Types as *constructive* proofs - Because types can depend on any values and can be computed at compile time, they can be used as constructive proofs, which must terminate. This means a well-typed program, which in itself is a proof, is always terminating, which in turn means that it must consist out of total functions.

Thus it should become clear that there exists a strong relation between property-based tests and dependent types: in property-based testing we express specifications / properties / laws in code and test their invariance at run time by random sampling the space. In dependent-types it is possible to express such properties already statically in types. So far no research using dependent types in agent-based simulation exists at all. We hypothesise that dependent types allow to push the correctness of ABS to a new, unprecedented level by narrowing the gap between model specification and implementation even further.

C.1 Constructivism

The main theoretical and philosophical foundations of dependent types are Martin-Löf intuitionistic type theory. The view of dependently typed programs to be proofs is rooted in a deep philosophical discussion on the foundations of mathematics, which evolve around the existence of mathematical objects, with two conflicting positions known as classic vs. constructive [2]. In general, the constructive position has been identified with realism and empirical computational content, where the classical one with idealism and pragmatism.

In the classical view, the position is that to prove $\exists x.P(x)$ it is sufficient to prove that $\forall x.\neg P(x)$ leads to a contradiction. The constructive view would claim that only the contradiction is established but that a proof of existence has to supply an evidence of an x and show that $P(x)$ is provable. In the end this boils down whether to use proof by contradiction or not, which is sanctioned by the law of the excluded middle which says that $A \vee \neg A$ must hold. The classic position accepts that it does and such proofs of existential statements as above, which follow directly out of the law of the excluded middle, abound in mathematics (Polynomial of degree n has n complex roots; continuous functions which change sign over a compact real interval have a zero in that interval,...). The constructive view rejects the law of the excluded middle and thus the position that every statement is seen as true or false, independently of any evidence either way. [154] (p. 61): *The constructive view of logic concentrates on what it means to prove or to demonstrate convincingly the validity of a statement, rather than concentrating on the abstract truth conditions which constitute the semantic foundation of classical logic.*

To prove a conjunction $A \wedge B$ we need prove both A and B, to prove $A \vee B$ we need to prove one of A, B and know which we have proved. This shows that the law of the excluded middle can not hold in a constructive approach because we have no means of going from a proof to its negation. Implication $A \Rightarrow B$ in constructive position is a transformation of a proof A into a proof B: it is a function which transforms proofs of A into proofs of B. The constructive approach also forces us to rethink negation, which is now an implication from some proof to an absurd proposition (bottom): $A \Rightarrow \bot$. Thus a negated formula has no computational context and the classical tautology $\neg\neg A \Rightarrow A$ is then obviously no longer valid. Constructively solving this would require us to be able to effectively compute / decide whether a proposition is true or false - which amounts to solving the halting problem, which is not possible in general.

A very important concept in constructivism is that of finitary representation / description. Objects which are infinite e.g. infinite sets as in classic mathematics, fail to have computational representation, they are not computable. This leads to a fundamental tenet in constructive mathematics: [154] (p. 62): *Every object in constructive mathematics is either finite [..] or has a finitary description*

Concluding, we can say that constructive mathematics is based on principles

[2] We follow the excellent introduction on constructive mathematics [154], chapter 3.

quite different from classical mathematics, with the idealistic aspects of the latter replaced by a finitary system with computational content. Objects like functions are given by rules, and the validity of an assertion is guaranteed by a proof from which we can extract relevant computational information, rather than on idealist semantic principles.

This short section only touched on the very basics and omitted dependently typed programming concepts altogether, For an in-depth introduction to dependent types and programming languages with dependent type systems we refer to: [26, 125, 129, 144, 154].

C.2 Dependent types in ABS

ABS as *constructive and exploratory* science [45, 46], follows the Popperian approach of falsification: the aim is to construct a model which explains a real-world (empirical) phenomenon. If validation shows that the generated dynamics match the ones of the real-world sufficiently enough, we can say that we have found *a* hypothesis, the model, with emergent properties explaining the real-world phenomenon sufficiently enough. Epstein describes this process with [45]: *"If you can't grow it, you can't explain it"*. Thus, an agent-based model and the simulated dynamics can be understood as a constructive proof which explain a real-world phenomenon sufficiently enough. Although Epstein certainly was not talking about a constructive proof in any mathematical sense in this context (he was using the word *generative*), dependent types *might* be a perfect match and correspondence between the constructive nature of ABS and programs as proofs.

When we talk about dependently typed programs to be proofs, then we also must attribute the same to dependently typed ABS, which are then constructive proofs as well. The question is then: a constructive proof of what? It is not entirely clear *what we are proving* when we are constructing dependently typed ABS. Probably the answer might be that a dependently typed ABS is then indeed a constructive proof in a mathematical sense, explaining a real-world phenomenon sufficiently well - we have closed the gap between a rather informal constructivism as mentioned above when citing Epstein who certainly didn't mean it in a constructive mathematical sense, and a formal constructivism, made possible by the use of dependent types.

Models like Sugarscape (Chapter 2.2.2) are exploratory in nature and don't have a formal ground truth where one could derive equilibria or dynamics from and validate with. In such models the researchers work with informal hypotheses which they express before running the model and then compare them informally against the resulting dynamics or more formally with property-based tests (see Appendix B). It would be of interest if dependent types could be made of use in encoding hypotheses on a more constructive and formal level directly into the implementation code. So far we have no idea how this could be done but it might be a very interesting application as it allows for a more formal and

automatic testable approach to hypothesis checking.

Often, Agent-Based Models define their agents in terms of state-machines. It is easy to make wrong state-transitions e.g. in the SIR model when an infected agent should recover, the compiler does not prevent one from making the transition back to susceptible. Using dependent types it might be possible to encode invariants and state-machines on the type level which can prevent such invalid transitions already at compile time. This would be a huge benefit for ABS because of the popularity of state-machines in ABS.

State-machines often have timed transitions e.g. in the SIR model, an infected agent recovers after a given time. Nothing prevents us from introducing a bug and *never* doing the transition at all. With dependent types we might be able to encode the passing of time in the types and guarantee on a type level that an infected agent has to recover after a finite number of time steps. Also, it would be interesting to use dependent types to encode the strong monotonic increasing flow of time.

In more sophisticated models agents interact in more complex ways with each other e.g. through message exchange using agent ids to identify target agents. The existence of an agent is not guaranteed and depends on the simulation time because agents can be created or terminated at any point during simulation. Dependent types could be used to implement agent ids as a proof that an agent with the given id exists *at the current time step*. This also implies that such a proof cannot be used in the future, which is prevented by the type system as it is not safe to assume that the agent will still exist in the next step.

Coming back to the original problem of the totality of the SIR model, dependent types in general and Idris' ability for totality- and termination-checking in particular, should theoretically allow us to prove that an agent-based SIR implementation terminates after finite time. If an implementation of the agent-based SIR model in Idris is total it is a formal proof by construction. Such an implementation should not run for a limited virtual time but run unrestricted of the time and the simulation should terminate as soon as there are no more infected agents, returning the termination time as an output. Also if we find a total implementation of the SIR model and extend it to the SIR+S model, which adds a cycle from recovered back to susceptible, then the simulation should become again as the pool of susceptible agents becomes potentially infinite.

The HOTT book [129] states that lists, trees,... are inductive types/inductively defined structures where each of them is characterized by a corresponding *induction principle*. Thus, for a constructive proof of the totality of the agent-based SIR model we need to find the induction principle of it. This leaves us with the question of what the inductive, defining structure of the agent-based SIR model is? Is it a tree where a path through the tree is one way through the simulation or is it something else? It seems that such a tree would grow and then shrink again e.g. infected agents. Can we then apply this further to (agent-based) simulation in general?

A central question in tackling this is whether to follow a model- or an agent-centric approach. The former one looks at the model and its specifications as a whole and encodes them e.g. one tries to directly find a total implementation of

an agent-based model. The latter one looks only at the agent level and encodes that as dependently typed as possible and hopes that model guarantees emerge on a meta-level - put otherwise: does the totality of an implementation emerge when we follow an agent-centric approach?

C.3 Discussion

In this appendix we briefly discussed the *correspondence between the equilibrium of a simulation and the totality of its implementation.* We have shown by informal reasoning that the SIR implementation will reach an equilibrium within finite time, thus it is not necessary to limit the simulation to a finite number of steps but terminate it when this equilibrium is reached. Unfortunately, using property-based tests we cannot prove that an implementation actually will terminate, which is in general not computable. However, when abandoning Turing completeness, by using pure functional programming languages with dependent types this actually becomes possible where one can write *total* programs, which are guaranteed to terminate, checked by the compiler.

We briefly introduced the concept of dependent types, their underlying philosophical foundations and discussed their potential use in ABS. As language of choice, for conducting research on dependent types in ABS, we propose Idris [23] 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.

Concluding, we see this thesis on pure functional programming without dependent types as a first step towards a more structural understanding of ABS implementations, where dependent types should allow us to develop this even further. A full treatment of dependent types in ABS are beyond the scope of this thesis and would justify a thesis in itself, thus we leave it for further research.