PhD Thesis

The Pure Functional Programming Paradigm In Agent-Based Simulation

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

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

May 3, 2019

Abstract

This thesis shows how to implement Agent-Based Simulations (ABS) using the *pure* functional programming paradigm and what the benefits and drawbacks are when doing so. As language of choice, Haskell is used due to its modern nature, increasing use in real-world applications and *pure* nature. The thesis presents various implementation techniques to ABS and then discusses concurrency and parallelism and verification and validation in ABS in a pure functional setting. Additionally the thesis briefly discusses the use of dependent types in ABS, to close the gap between specification and implementation - something the presented implementation techniques don't focus on. Finally a case-study is presented which tries to bring together the insights of the previous chapters by replicating an agent-based model both in pure and dependently typed functional programming. The agent-based model which was selected was much discussed in ABS communities as it claimed to have solved a fundamental problem of economics but it was then found that the implementation had a number of bugs which shed doubt on the validity and correctness of the results. The thesis' case study investigates whether this failure could have happened in pure and dependent functional programming and is a further test to see of how much value functional programming is to ABS.

TODO "finalise research" milestone until end of June 2019

1. finish property-base testing work in Haskell: make sense of the SIR model verification which does not work: either give an in-depth explanation why not and why our implementation is still valid or find a way of solving it.
2. implement SIR equilibrium-totality correspondence in Idris, closely related to the property-based testing
3. implement Gintis Case-Study and focus on property-tests and reasoning and potential directions for a dependently-typed implementation. this will be highly important for my internal assessor

TODO "first draft" milestone until end of July 2019

1. Revise and finalise methodology.
2. Write short monad transformer section in methodology
3. Draft all chapters of discussion part.
4. Write a first draft of conclusions and future research

TODO "reflecting and filling in holes" milestone until end of August 2019

1. Reading: Ionescu thesis
2. Understand Ionescu: check what ionescou wants from his students and what literaure he cites, watch YouTube video

3. Generalising ABS: extract the foundational properties: agents as monad / comonad / arrow and discuss their implications from an abstract point of view

4. Refine Presentation: Has to come across as a thesis, not a technical report! Need to discuss concepts more generally and derive more concepts from the explanations, otherwise it looks like an advanced technical report.

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

1. consistency of words / expressions
    - avoid WE because I have written this thesis, so remove all personal references e.g. not We or I
    - s vs 's vs. s' e.g. Agents vs. Agents's vs. Agents'
    - case study vs. case-study (no -)
    - Monad vs monad (lower/upper case?)
    - speedup vs speed up vs speed-up
    - types / function names / monad names ITALIC!?
    - main-thread vs main thread
    - avoid don't, won't, haven't, e.g.,... and write them all out
2. proof reading
3. refining expressions
4. reconcile all code into thesis code folder?
5. extract all references of the thesis in separate file and manually fine-tune them so that they are perfect: including DOI, include link to webpage in case of a blog,...
6. PERFECT introduction chapter
7. PERFECT conclusion chapter

TODO prepare for viva:

1. read transformers chapters
2. read exception chapter
3. read basic libraries chapter
4. completely understand monad-transformers: what is executed in which order? what about commutativity? e.g. STM is innermost monad, will be evaluated last to get an IO but this in fact leads then to the evaluation of all the monads before because STM is able to re-run

Contents

To my parents for their unconditional love and support throughout all my life.

Acknowledgements

Peer-Olaf Siebers Supervisor Thorsten Altenkirch Supervisor FP Group, for always being open to discussions, keen to help. Martin Handley and James Hey for working through my thesis and their feedback. Ivan Perez for always having an open ear for questions, valuable discussions with him and his contributions. Julie Greensmith for the valuable discussions and pointing me into right directions at important stages of the phd.

Part I:

Preliminiaries

Chapter 1

Introduction

TODO ARGUMENT REFINEMENT: make it clear, that the aim of the thesis and the way it approaches this topic is NOT through a direct comparison with oop languages / implementations / Frameworks. they are only layed out to understand the current state of the field but not as a comparison. argue precisely why not. the basic aim is rather on a conceptual level where we sometimes draw comparison in both directions e.g. the lack of method calls&open recursion.

TODO SCIENTIFIC WORK: WARNING: i must be careful not to show some evidence of benefits / drawbacks. otherwise its not scientific. so i need a very careful argument / narrative which avoids FP vs OOP but still allows some judgement over both in terms of Advantages and drawbacks

TODO: one thing we are lucky in ABS is that most models at their core are pure computations without IO like user-interactions. this fact makes it a lot more promising to approach it from a pure functional approach! if this would not be the case, we would have a harder time selling pure functional programming. there are ways of dealing with IO in a pure functional way e.g. constructing IO actions and executing them outside of the agents but this only complicates things and we might as well make the agents run in IO (really? don't we gain something?)

The traditional approach to Agent-Based Simulation (ABS) has so far always been object-oriented techniques, due to the influence of the seminal work of Epstein et al [38] 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* [90], which still holds up today.

This thesis challenges the metaphor and explores ways of approaching ABS with the *pure* functional programming paradigm using the languages Haskell and Idris. It is the first one to do so on a *systematical* level and develops a foundation by presenting fundamental concepts and advanced features to show how to leverage the benefits of both languages [61, 19] to become available when implementing ABS functionally. By doing this, the thesis both 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.

TODO: also i see the functional approach as a way to think and explore ABS in a deeper way - especially to develop a deeper and more complete understanding on the computational structure underlying ABS. It is established, that FP helps in structuring computation in a very clear and precise way, leading to a deeper understanding about problems TODO this is also what Ionescu says in his thesis. So we also see FP as a tool for a deeper understanding of the computational structures involved in ABS. By implementing use-cases, reflecting on them and generalising we extract implicit knowledge and make it explicit. We hope that this undertaking is to the whole benefit of the ABS discipline and will also feed back into the traditional implementation techniques of OOP.

this Thesis is about abstracting the software concept of abs so we arrive at deeper understanding of abs Implementation in general, and its computatuonal structure. the tool used is pure FP because it sees structures as essential to programming. to quote Ted Kaminski in "Abstractions at the boundaries": Through the use of the right abstractions, we humans were able to reduce our attention to some partof the problem, something small enough that the human brain is capable of actually thinking deeply about it. Something we could study, and eventually emerge with a deep understand of. Then we could go back, armed with our new understanding, to re-interpret the larger problem. Eventually, with the right luck and/or genius, the real problem gets made small enough that *click*. abstractions were also a tool for human thought. dependent types should allow this to push even further with more Focus on structure than on application

This thesis claims that the agent-based simulation 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.

Thus this thesis' general research question is *how to implement ABS purely functional and what the benefits and drawbacks are of doing so.* Further, it hypothesises that by using pure functional programming for implementing ABS makes it is easy to add parallelism and concurrency, the resulting simulations are easy to test and verify, applicable to property-based testing, 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.

1.1 Publications

Throughout the course of the Ph.D. four (4) papers were published:

1. The Art Of Iterating - Update Strategies in Agent-Based Simulation [121] - This paper derives the 4 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-driven ABS purely functional.

2. Pure Functional Epidemics [**?**] - 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 the Methodology chapter of this thesis.

3. A Tale Of Lock-Free Agents (TODO cite) - 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 famous 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. It does not go into technical details of functional programming as it is written for a simulation Journal.

4. The Agents' New Cloths? Towards Pure Functional Agent-Based Simulation (TODO cite) - 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.

5. Show Me Your Properties! The Potential Of Property-Based Testing In Agent-Based Simulation - 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.

1.2 Contributions

1. 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. Due to the increased interested in functional concepts which were added to object-oriented languages in recent years, because of its established benefits in concurrent programming, testing and software-development in general, presenting such foundational research gives this thesis significant impact. Also it opens the way for the benefits of FP to incorporate into scientific computing, which are explored in the contributions below.

2. 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. STM is particularly strong in pure FP because of retry-semantics can be guaranteed to exclude non-repeatable persistent side-effects already at compile time. 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. Due to the increasing need for massively large-scale ABS in recent years [78], making this possible within a purely functional approach as well, gives this thesis substantial impact.

3. This thesis is the first to present the use of property-based testing in ABS which allows a declarative specification- testing of the implemented ABS directly in code with *automated* 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. This should lead to simulation software which is more likely to be correct, thus making this a significant contribution with valuable impact.

4. This thesis is the first to outline the potential use of *dependent types* to Agent-Based Simulation on a *conceptual level* to investigate its usefulness for increasing the correctness of a simulation. Dependent types can help to narrow the gap between the model specification and its implementation, reducing the potential for conceptual errors in model-to-code translation. This immediately leads to fewer number of tests required due to guarantees being expressed already at compile time. Ultimately dependent types lead to higher confidence in correctness due to formal guarantees in code, making this a unique contribution with high impact.

1.3 Thesis structure

This thesis focuses on a strong narrative which tells the story of *how* to do ABS with pure functional programming, *why* one would do so and when one should *avoid* this paradigm in ABS.

TODO: write when all is finished

Chapter 2

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 [30, 116, 69].

A multi-method simulation library in Haskell called *Aivika 3* is described in the technical report [114]. 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 effect type for realising parallel and distributed simulation but also discusses generalising their approach to avoid running in IO.

In his master thesis [13] the author investigates Haskells' parallel and concurrency features to implement (amongst others) *HLogo*, a Haskell clone of the NetLogo [136] simulation package, focusing on using STM for a limited form of agent-interactions. *HLogo* is basically a re-implementation of NetLogos API in Haskell where agents run within an unrestricted context (known as *IO*) and thus can also make use of STM functionality. The benchmarks show that this approach does indeed result in a speed-up especially under larger agent-populations. The authors' thesis can be seen as one of the first works on ABS using Haskell. Despite the concurrency and parallel aspect our work share, our approach is rather different: we avoid IO within the agents under all costs and explore the use of STM more on a conceptual level rather than implementing a ABS library and compare our case-studies with lock-based and imperative implementations.

There exists some research [32, 127, 112] 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 Multi Agent Systems [139]. 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, which focus on message-passing and their implementation. Further they show that using this kind of concurrency allows to overcome some problems of low level concurrent programming as well. Also [13] 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 this model worked, 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, which despite same initial starting conditions, might result in different dynamics each time due to concurrency.

The work [78] discusses a framework, which allows to map Agent-Based Simulations to Graphics Processing Units (GPU). Amongst others they use the SugarScape model [38] and scale it up to millions of agents on very large environment grids. They reported an impressive speed-up of a factor of 9,000. Although their work is conceptually very different we can draw inspiration from their work in terms of performance measurement and comparison of the SugarScape model.

Using functional programming for DES was discussed in [69] 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 agent-based simulations was presented in [109, 128]. 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 [29] which was created for simulation purposes. Simula 67 already supported Discrete Event Simulation and was highly influential for today's object-oriented languages. Although the language was important and influential, in our research we look into different approaches, orthogonal to the existing object-oriented concepts.

Lustre is a formally defined, declarative and synchronous dataflow programming language for programming reactive systems [52]. While it has solved some issues related to implementing ABS in Haskell it still lacks a few important features necessary for ABS. We don't see any way of implementing an environment in Lustre as we do in Chapters 5 and 6. 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.

The authors of [16] 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. Our architecture for synchronous agent-interaction as discussed in Chapter TODO was not directly inspired by their work but has some similarities: the use of messages and the problem of when to advance time in models with arbitrary number synchronised agent-interactions.

The authors of [17] 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. This paper is the closest usage of dependent types in agent-based simulation we could find in the existing literature and to our best knowledge there exists no work on general concepts of implementing pure functional agent-based simulations with dependent types. As a remedy to having no related work to build on, we looked into works which apply dependent types to solve real world problems from which we then can draw inspiration from.

In his talk [117], 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 [49] defines computer-games as *"[..] soft real-time interactive agent-based computer simulations"* (p. 9) and in the end they have also very important 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. 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 parallelized easily in a pure functional language and updating game-objects concurrently using software transactional memory (STM).

Chapter 3

Methodology

This chapter introduces the background and methodology used in the following chapters.

3.1 Agent-Based Simulation

TODO RESTRUCTURING: maybe move SIR and Sugarscape descriptions into this section as 2 examples.

History, methodology (what is the purpose of ABS: 3rd way of doing science: exploratory, helps understand real-world phenomena), classification according to [80], ABS vs. MAS, event- vs. time-driven [85], examples: agent-based SIR, SugarScape, Gintis Bartering

We understand ABS as a method of modelling and simulating a system where the global behaviour may be unknown but the behaviour and interactions of the parts making up the system is known. Those parts, called agents, are modelled and simulated out of which then the aggregate global behaviour of the whole system emerges. So the central aspect of ABS is the concept of an agent which can be understood as a metaphor for a pro-active unit, situated in an environment, able to spawn new agents and interacting with other agents in a network of neighbours by exchange of messages [139]. It is important to note that we focus our understanding of ABS on a very specific kind of agents where the focus is on communicating entities with individual, localized behaviour from out of which the global behaviour of the system emerges. We informally assume the following about our agents:

- They are uniquely addressable entities with some internal state.
- They can initiate actions on their own e.g. change their internal state, send messages, create new agents, kill themselves.
- They can react to messages they receive with actions as above.
- They can interact with an environment they are situated in.

An implementation of an ABS must solve two fundamental problems:

1. **Source of pro-activity** How can an agent initiate actions without the external stimuli of messages?

2. **Semantics of Messaging** When is a message m, sent by agent A to agent B, visible and processed by B?

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-system, 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 multi-processing in the former.

The semantics of messaging define when sent messages are visible to the receivers and when the receivers process them. Message-processing could happen either immediately or delayed, depending on how message-delivery works. There are two ways of message-delivery: immediate or queued. In the case of immediate message-deliver the message is sent directly to the agent without any queuing in between e.g. a direct method-call. This would allow an agent to immediately react to this message as this call of the method transfers the thread-of-execution to the agent. This is not the case in the queued message-delivery where messages are posted to the message-box of an agent and the agent pro-actively processes the message-box at regular points in time.

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

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

We informally assume the following about our agents [113, 139, 80, 91]:

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

Epstein [37] identifies ABS to be especially applicable for analysing *"spatially distributed systems of heterogeneous autonomous actors with bounded information and computing capacity"*. They exhibit the following properties:

- Linearity & Non-Linearity - actions of agents can lead to non-linear behaviour of the system.
- Time - agents act over time, which is also the source of their pro-activity.
- States - agents encapsulate some state, which can be accessed and changed during the simulation.
- Feedback-Loops - because agents act continuously and their actions influence each other and themselves in subsequent time-steps, feedback-loops are the common in ABS.
- Heterogeneity - agents can have properties (age, height, sex,...) where the actual values can vary arbitrarily between agents.
- Interactions - agents can be modelled after interactions with an environment or other agents.
- Spatiality & Networks - agents can be situated within e.g. a spatial (discrete 2D, continuous 3D,...) or complex network environment.

Note that there doesn't exist a commonly agreed technical definition of ABS but the field draws inspiration from the closely related field of Multi-Agent Systems (MAS) [139], [135]. 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.

macal paper [80]: very good survey/review paper on ABMS in General. fp can help with challenges h2, h4 and h5. also fp can help macals added transparency challenge, my thesis in general also adresses the knowledge challenge of macal "lack of abms educational...", note that we do NOT address ease-of-use as our approach is not easy to use. also the yampa approach can be seen as a hybrid approach of ABS/SD as posed as Research Challenge by macal. further

STM might be one way of tackling large-scale ABS as identified as Research Challenge by macal. also this paper supports that ABS is a fundamentally new technique that offers the Potential to solve problems that are not robustly addressed by other methods

3.1.1 Traditional approaches

Introduce established implementation approaches to ABS. Frameworks: NetLogo, Anylogic, Libraries: RePast, DesmoJ. Programming: Java, Python, C++. Correctness: ad-hoc, manual testing, test-driven development.

TODO: we need citiations here to support our claims!

TODO: this is a nice blog: https://drewdevault.com/2018/07/09/Simple-correct-fast.html

The established approach to implement ABS falls into three categories:

1. Programming from scratch using object-oriented languages where Java and Python are 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 one.
3. Using a high-level ABS tool-kit for non-programmers, which allow customization 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.

In general one can say that these approaches, especially the 3rd one, support fast prototyping of simulations which allow quick iteration times to explore the dynamics of a model. Unfortunately, all of them suffer the same problems when it comes to verifying and guaranteeing the correctness of the simulation.

The established way to test software in established object-oriented approaches is writing unit-tests which cover all possible cases. This is possible in approach 1 and 2 but very hard or even impossible when using an ABS tool-kit, as in 3, which is why this approach basically employs manual testing. In general, writing those tests or conducting manual tests is necessary because one cannot guarantee the correct working at compile-time which means testing ultimately tests the correct behaviour of code at run-time. The reason why this is not possible is due to the very different type-systems and paradigm of those approaches. Java has a strong but very dynamic type-system whereas Python is completely dynamic not requiring the programmer to put types on data or variables at all. This means that due to type-errors and data-dependencies run-time errors can occur which origins might be difficult to track down.

It is no coincidence that JavaScript, the most widely used language for programming client-side web-applications, originally a completely dynamically typed language like Python, got additions for type-checking developed by the industry through TypeScript. This is an indicator that the industry acknowledges types as something important as they allow to rule out certain classes of

bugs at run-time and express guarantees already at compile-time. We expect similar things to happen with Python as its popularity is surging and more and more people become aware of that problem. Summarizing, due to the highly dynamic nature of the type-system and imperative nature, run-time errors and bugs are possible both in Python and Java which absence must be guaranteed by exhaustive testing.

The problem of correctness in agent-based simulations became more apparent in the work of Ionescu et al [68] which tried to replicate the work of Gintis [46]. 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 break-through for economics as the theory of walrasian general equilibrium is non-constructive as it only postulates the properties of the equilibrium [26] but does not explain the process and dynamics through which this equilibrium can be reached or constructed - Gintis seemed to have found just this process. Ionescu et al. [68] failed and were only able to solve the problem by directly contacting Gintis which provided the code - the definitive formal reference. It was found that there was a bug in the code which led to the "revolutionary" results which were seriously damaged through this error. They also reported ambiguity between the informal model description in Gintis paper and the actual implementation. TODO: it is still not clear what this bug was, find out! look at the master thesis

This is supported by a talk [117], in which Tim Sweeney, CEO of Epic Games, discusses the use of main-stream imperative object-oriented programming languages (C++) in the context of Game Programming. Although the fields of games and ABS seem to be very different, in the end they have also very important similarities: both are simulations which perform numerical computations and update objects in a loop either concurrently or sequential [49]. Sweeney reports that reliability suffers from dynamic failure in such languages e.g. random memory overwrites, memory leaks, accessing arrays out-of-bounds, 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.

TODO: general introduction

TODO: list common bugs in object-oriented / imperative programming TODO: java solved many problems TODO: still object-oriented / imperative ultimately struggle when it comes to concurrency / parallelism due to their mutable nature.

TODO: [129]

TODO: software errors can be costly TODO: bugs per loc

3.2 Pure functional programming

Functional programming (FP) is called *functional* because it makes functions the main concept of programming, promoting them to first-class citizens: functions can be assigned to variables, they can be passed as arguments to other functions and they can be returned as values from functions. The roots of FP lie in the Lambda Calculus which was first described by Alonzo Church [22]. This is a fundamentally different approach to computing than imperative programming (includeing established object-orientation) which roots lie in the Turing Machine [125]. 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.

MacLennan [81] defines Functional Programming as a methodology and identifies it with the following properties (amongst others):

1. It is programming without the assignment-operator.
2. It allows for higher levels of abstraction.
3. It allows to develop executable specifications and prototype implementations.
4. It is connected to computer science theory.
5. Suitable for Parallel Programming.
6. Algebraic reasoning.

[3] defines Functional Programming as "a computer programming paradigm that relies on functions modelled on mathematical functions." Further they explicate that it is

- in Functional programming programs are combinations of expressions
- Functions are *first-class* which means the can be treated like values, passed as arguments and returned from functions.

[81] 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:

- Value is independent of the evaluation order.
- Expressions can be evaluated in parallel.
- Referential transparency.
- No side effects.
- Inputs to an operation are obvious from the written form.

- 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 [81] 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 the *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 the map, fold, zip, operators. So functional programming is not really possible in this way in classic imperative languages e.g. C as you cannot construct new functions and return them as results from functions [1].

The equivalence in functional programming to 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 which are 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. Although it might seem strange wanting to have 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 with 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 e.g. a map, zip, fold over monads which results in polymorphic behaviour - this is the meaning when one says that a language is polymorphic in its side-effects.

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 [92] 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.

[1] Object-Oriented languages like Java let you to partially work around this limitation but are still far from *pure* functional programming.

For an excellent and widely used introduction to programming in Haskell we refer to [67]. Other, more exhaustive books on learning Haskell are [77, 3]. For an introduction to programming with the Lambda-Calculus we refer to [86]. For more general discussion of functional programming we refer to [63, 81, 61].

3.2.1 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 history and that it loses its referential transparency character, which makes understanding and debugging much harder. Examples for side-effects are (amongst others): modifying state, 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 we need 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 then ensured by the compiler which means that a program in which one tries to e.g. read a file in a function which only allows drawing random-numbers will fail to compile. Haskell also provides mechanisms to combine multiple effects e.g. one can define a function which can draw random-numbers and modify some state. The most common side-effect types are: *IO* allows all kind of I/O related side-effects: reading/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 / Writer / State* allows to read / write / both from / to an environment.

A function without any side-effect type is called *pure*, and the *factorial* function is indeed pure. Below we give an example of a function which is not pure. The *queryUser* function *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 - this is no accident and 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: [87, 130, 131, 132, 71].

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 which implies that we don't need test them with e.g. unit-tests. Second, because running effects themselves is *pure*, we can execute effectful functions in a very controlled way by making the effect-context 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. TODO: need maybe more explanation on how effects are executed

Further, this type system allows Haskell to make a very clear distinction between parallelism and concurrency. Parallelism is always deterministic and thus pure without side-effects because although parallel code runs concurrently, it does by definition not interact with data of other threads. This can be indicated through types: we can run pure functions in parallel because for them it doesn't matter in which order they are executed, the result will always be the same due to the concept of referential transparency. Concurrency is potentially non-deterministic because of non-deterministic interactions of concurrently running threads through shared data. For a technical, in-depth discussion on Parallelism and Concurrency in Haskell we refer to the following books and papers: [82, 95, 54, 83].

TODO: explain monad transformers because I am using them heavily throughout the thesis

3.2.2 Theoretical Foundation

The theoretical foundation of Functional Programming is the Lambda Calculus, which was introduced by Alonzo Church in the 1930s. After some revision due to logical inconsistencies which were shown by Kleene and Rosser, Church

published the untyped Lambda Calculus in 1936 which, together with a type-system (e.g. Hindler-Milner like in Haskell) on top is taken as the foundation of functional programming today.

[81] defines a calculus to be "... a notation that can be manipulated mechanically to achieve some end;...". The Lambda Calculus can thus be understood to be a notation for expressing computation based on the concepts of *function abstraction*, *function application*, *variable binding* and *variable substitution*. It is fundamentally different from the notation of a Turing Machine in the way it is applicative whereas the Turing Machine is imperative / operative. To give a complete definition is out of the scope of this text, thus we will only give a basic overview of the concepts and how the Lambda Calculus works. For an exhaustive discussion of the Lambda Calculus we refer to [81] and [10].

Function Abstraction Function abstraction allows to define functions in the Lambda Calculus. If we take for example the function $f(x) = x^2 - 3x + a$ we can translate this into the Lambda Calculus where it denotes: $\lambda x.x^2 - 3x + a$. The λ symbol denotes an expression of a function which takes exactly one argument which is used in the body-expression of the function to calculate something which is then the result. Functions with more than one argument are defined by using nested λ expressions. The function $f(x, y) = x^2 + y^2$ is written in the Lambda Calculus as $\lambda x.\lambda y.x^2 + y^2$.

Function Application When wants to get the result of a function then one applies arguments to the function e.g. applying $x = 3, y = 4$ to $f(x, y) = x^2 + y^2$ results in $f(3, 4) = 25$. Function application works the same in Lambda Calculus: $((\lambda x.\lambda y.x^2 + y^2)3)4 = 25$ - the question is how the result is actually computed - this brings us to the next step of variable binding and substitution.

Variable Binding In the function $f(x) = x^2 - 3x + a$ the variable x is *bound* in the body of the function whereas a is said to be *free*. The same applies to the lambda expression of $\lambda x.x^2 - 3x + a$. An important property is that bound variables can be renamed within their scope without changing the meaning of the expression: $\lambda y.y^2 - 3y + a$ has the same meaning as the expression $\lambda x.x^2 - 3x + a$. Note that free variable *must not be renamed* as this would change the meaning of the expression. This process is called α-conversion and it becomes sometimes necessary to avoid name-conflicts in variable substitution.

Variable Substitution To compute the result of a Lambda Expression - also called evaluating the expression - it is necessary to substitute the bound variable by the argument to the function. This process is called β-reduction and works as follows. When we want to evaluate the expression $((\lambda x.\lambda y.x^2 + y^2)3)4$ we first substitute 4 for x, rendering $(\lambda y.4^2 + y^2)3$ and then 3 for y, resulting in $(4^2 + 3^2)$ which then ultimately evaluates to 25. Sometimes α-conversion becomes necessary e.g. in the case of the expression $((\lambda x.\lambda y.x^2 + y^2)3)y$ we must not substitute y directly for x. The result would be $(\lambda y.y^2 + y^2)3 = 3^2 + 3^2 = 18$

- clearly a different meaning than intended (the first y value is simply thrown away). Here we have to perform α-conversion before substituting y for x. $((\lambda x.\lambda y.x^2 + y^2)3)y = ((\lambda x.\lambda z.x^2 + z^2)3)y$ and now we can substitute safely without risking a name-clash: $((\lambda x.\lambda z.x^2 + z^2)3)y = (\lambda z.y^2 + z^2)3) = (y^2 + 3^2)3) = y^2 + 9$ where y occurs free.

Examples

$(\lambda x.x)$ denotes the identity function - it simply evaluates to the argument.

$(\lambda x.y)$ denotes the constant function - it throws away the argument and evaluates to the free variable y.

$(\lambda x.xx)(\lambda x.xx)$ applies the function to itself (note that functions can be passed as arguments to functions - they are *first class* in the Lambda Calculus) - this results in the same expression again and is thus a non-terminating expression.

We can formulate simple arithmetic operations like addition of natural numbers using the Lambda Calculus. For this we need to find a way how to express natural numbers [2]. This problem was already solved by Alonzo Church by introducing the Church numerals: a natural number is a function of an n-fold composition of an arbitrary function f. The number 0 would be encoded as $0 = \lambda f.\lambda x.x$, 1 would be encoded as $1 = \lambda f.\lambda x.fx$ and so on. This is a way of *unary notation*: the natural number n is represented by n function compositions - n things denote the natural number of n. When we want to add two such encoded numbers we make use of the identity $f^{(m+n)}(x) = f^m(f^n(x))$. Adding 2 to 3 gives us the following lambda expressions (note that we are using a sugared version allowing multiple arguments to a function abstraction) and reduces after 7 steps to the final result:

$2 = \lambda fx.f(fx)$
$3 = \lambda fx.f(f(fx))$
$ADD = \lambda mnfx.mf(nfx)$

ADD 2 3
$1 : (\lambda mnfx.mf(nfx))(\lambda fx.f(f(fx)))(\lambda fx.f(fx))$
$2 : (\lambda nfx.(\lambda fx.f(f(fx)))f(nfx))(\lambda fx.f(fx))$
$3 : (\lambda fx.(\lambda fx.f(f(fx)))f((\lambda fx.f(fx))fx))$
$4 : (\lambda fx.(\lambda x.f(f(fx)))((\lambda fx.f(fx))fx))$
$5 : (\lambda fx.f(f(f(\lambda fx.f(fx))fx)))$
$6 : (\lambda fx.f(f(f(\lambda x.f(fx))x)))$
$7 : (\lambda fx.f(f(f(f(fx)))))$

[2] In the short introduction for sake of simplicity we assumed the existence of natural numbers and the operations on them but in a pure lambda calculus they are not available. In programming languages which build on the Lambda Calculus e.g. Haskell, (natural) numbers and operations on them are built into the language and map to machine-instructions, primarily for performance reasons.

3.2.2.1 Types

The Lambda Calculus as initially introduced by Church and presented above is *untyped*. This means that the data one passes around and upon one operates has no type: there are no restriction on the operations on the data, one can apply all data to all function abstractions. This allows for example to add a string to a number which behaviour may be undefined thus leading to a non-reducible expression. This led to the introduction of the simply typed Lambda Calculus which can be understood to add tags to a lambda-expression which identifies its type. One can then only perform function application on data which matches the given type thus ensuring that one can only operate in a defined way on data e.g. adding a string to a number is then not possible any-more because it is a semantically wrong expression. The simply typed lambda calculus is but only one type-system and there are much more evolved and more powerful type-system e.g. *System F* and *Hindley-Milner Type System* which is the type-system used in Haskell. It is completely out of the scope of this text to discuss type systems in depth but we give a short overview of the most important properties.

Generally speaking, a type system defines types on data and functions. Raw data can be interpreted in arbitrary ways but a type system associates raw data with a type which tells the compiler (and the programmer) how this raw data is to be interpreted e.g. as a number, a character,... Functions have also types on their arguments and their return values which defines upon which types the function can operate. Thus ultimately the main purpose of a type system is to reduce bugs in a program. Very roughly one can distinguish between static / dynamic and strong / weak typing.

Static and dynamic typing A statically typed language performs all type checking at compile time and no type checking at runtime, thus the data has no type-information attached at all. Dynamic typing on the other hand performs type checking during run-time using type-information attached to values. Some languages use a mix of both e.g. Java performs some static type checking at compile time but also supports dynamic typing during run-time for downcasting, dynamic dispatch, late binding and reflection to implement object-orientation. Haskell on the other hand is strictly statically typed with no type checks at runtime.

Strong and weak typing A strong type system guarantees that one cannot bypass the type system in any way and can thus completely rule out type errors at runtime. Pointers as available in C are considered to be weakly typed because they can be used to completely bypass the type system e.g. by casting to and from a (void*) pointer. Other indications of weak typing are implicit type conversions and untagged unions which allow values of a given typed to be viewed as being a different type. There is not a general accepted definition of strong and weak typing but it is agreed that programming languages vary across the strength of their typing: e.g. Haskell is seen as very strongly typed, C very

weakly, Java more strongly typed than C whereas Assembly is considered to be untyped.

3.2.3 Language of choice

In our research we are using the *pure* functional programming language Haskell. The paper of [61] 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 main points why we decided to go for Haskell are:

- Rich Feature-Set - it has all fundamental concepts of the pure functional programming paradigm included. 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 [62, 61], is applicable to a number of real-world problems [95] 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.
- Purity - 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 (exceptions are when doing concurrency but there we restrict most of the concepts to STM).
- It is as closest to pure functional programming, as in the lambda-calculus, as we want to get. Other languages are often a mix of paradigms and soften some criteria / are not strictly functional and have different purposes. Also Haskell is very strong rooted in Academia and lots of knowledge is available, especially at Nottingham, Lisp / Scheme was considered because it was the very first functional programming language but deemed to be not modern enough with lack of sufficient libraries. Also it would have given the Erlang was considered in prototyping and allows to map the messaging concept of ABS nicely to a concurrent language but was ultimately rejected due to its main focus on concurrency and not being purely functional. Scala was considered as well and has been used in the research on the Art Of Iterating paper but is not purely functional and can be also impure.

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

3.2.4 Functional Reactive Programming

Short introduction to FRP (yampa), based on my pure functional epidemics paper.

Functional Reactive Programming is a way to implement systems with continuous and discrete time-semantics in pure functional languages. There are many different approaches and implementations but in our approach we use *Arrowized* FRP [64, 65] as implemented in the library Yampa [60, 28, 88].

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.

$$Signal\,\alpha \approx Time \to \alpha$$
$$SF\,\alpha\,\beta \approx Signal\,\alpha \to Signal\,\beta$$

Yampa provides a number of combinators for expressing time-semantics, events and state-changes of the system. They allow to change system behaviour in case of events, run signal functions and generate stochastic events and random-number streams. We shortly discuss the relevant combinators and concepts we use throughout the paper. For a more in-depth discussion we refer to [60, 28, 88].

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

Dynamic behaviour To change the behaviour of a signal function at an occurrence of an event during run-time, (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 e.g. an exponential distribution. Yampa provides the combinator *occasionally :: RandomGen g ⇒ g → Time → b → SF a (Event b)* for this. It takes a random-number generator, a rate and a value the stochastic event will carry. It generates events on average with the given rate. Note that at most one event will be generated and no 'backlog' is kept. This means that

when this function is not sampled with a sufficiently high frequency, depending on the rate, it will lose events.

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

Running signal functions To *purely* run a signal function Yampa provides the function *embed :: SF a b → (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.

3.2.5 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 [64, 65].

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

There exists a number of arrow combinators, which allow arrowized programing in a point-free style but due to lack of space we will not discuss them here. Instead we make use of Paterson's do-notation for arrows [96], 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 [100].

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

3.2.6 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 [99, 98]:

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

MSFs are also arrows, which means we can apply arrowized programming with Patersons do-notation as well. MSFs are implemented in Dunai, which is available on Hackage. Dunai allows us to apply monadic transformations to every sample by means of combinators like $arrM :: Monad\ m \Rightarrow (a \rightarrow m\ b) \rightarrow MSF\ m\ a\ b$ and $arrM\_ :: Monad\ m \Rightarrow m\ b \rightarrow MSF\ m\ a\ b$. A part of the library Dunai is BearRiver, a wrapper, which re-implements Yampa on top of Dunai, which enables one to run arbitrary monadic computations in a signal function. BearRiver simply adds a monadic parameter m to each SF, which indicates the monadic context this signal function runs in.

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

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

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

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

Chapter 4

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 can we represent an agent, its local state and its interface?
2. How can we represent agent-to-agent interactions and what are their semantics?
3. How can we represent an environment?
4. How can we represent agent-to-environment interactions and what are their semantics?
5. How can agents and an environment initiate actions without external stimuli?
6. How can we 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-system, 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 multi-processing in the former.

Generally, there exist time- and event-driven approaches to ABS [85]. 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 discussed in the section on time-driven ABS, we use the agent-based SIR model [79, **?**]. 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, this is the very heart of Simulation: modelling a virtual system over some (virtual) time. 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 discussed in the section on event-driven ABS, we use the Sugarscape model. In this model time is discrete and represented by the natural numbers where agents act in every tick - time is not modelled explicitly as in the agent-based SIR case. In such a model, the underlying semantics map more naturally to a DES core, extended by ABS features. Although the Sugarscape model does not semantically map to a DES core in a strict sense, our implementation approach is very close to such it and can be easily extended to a true DES core - thus it serves as a good example for the discussion of the event-driven approach, and we also show how to extend it to a pure DES core, allowing to implement models with more explicit event-driven semantics as discussed in [85].

According to the (informal) definition of ABS (see Chapter 3.1), 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 OOP 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. Also 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.

The semantics of messaging define when sent messages are visible to the receivers and when the receivers process them. Message-processing could happen either immediately or delayed, depending on how message-delivery works. There are two ways of message-delivery: immediate or queued. In the case of

immediate message-deliver the message is sent directly to the agent without any queuing in between e.g. a direct method-call. This would allow an agent to immediately react to this message as this call of the method transfers the thread-of-execution to the agent. This is not the case in the queued message-delivery where messages are posted to the message-box of an agent and the agent pro-actively processes the message-box at regular points in time. With established OOP approaches we can have both: either a direct method-call or a message-box approach - in pure FP this is a much more subtle problem and it turns out that the problem of messaging / interacting of agents and of agents with the environment is the most subtle problem when approaching ABS from a pure functional perspective.

4.1 Update Strategies

Generally there are four strategies to approach time-driven ABS [121], where the differences deal with how the simulation is stepped / the agents are executed and the interaction-semantics.

4.1.1 Sequential Strategy

In this strategy there exists a globally synchronized time-flow and in each time-step 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 makes 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. A variation of this strategy is used See Figure 4.1 for a visualisation of the control flow in this strategy.

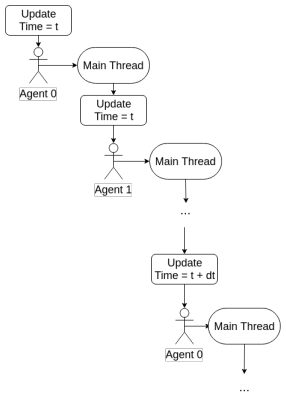


Figure 4.1: Control flow in the Sequential Strategy.

4.1.2 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 systematic error in this strategy. First it is not logical because all actions are meant to happen at the same time and also it would implicitly induce an ordering, violating 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 is the strategy used for the implementation of the agent-based SIR model, see below. See Figure 4.2 for a visualisation of the control flow in this strategy.

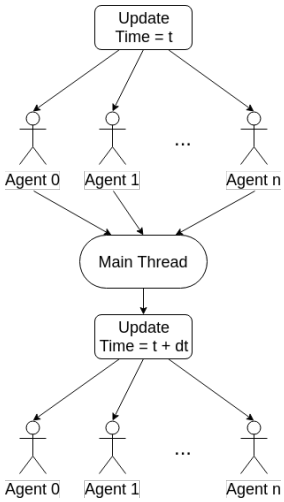


Figure 4.2: Control flow in the Parallel Strategy.

4.1.3 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 in parallel, which are able to see actions by others immediately, this is the very definition of concurrency: parallel execution with mutual read/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 first to a shared resource 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 will become important in the subsequent chapter on concurrency in ABS where we explain the use of Software Transactional Memory. See Figure 4.3 for a visualisation of the control flow in this strategy.

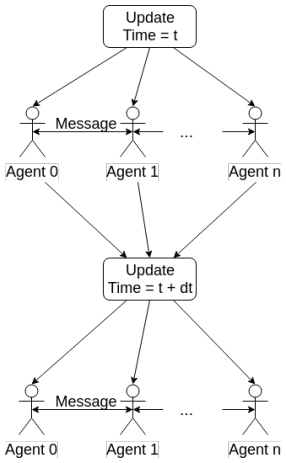


Figure 4.3: Control flow in the Concurrent Strategy.

4.1.4 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 multi-processor 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 e.g. visualize it. 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 [59] for which [50] and [25] 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. See Figure 4.4 for a visualisation of the control flow in this strategy.

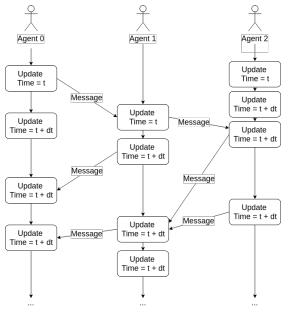


Figure 4.4: Control flow in the Actor Strategy.

4.2 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 using pure functional programming (FP). The *sequential* strategy will be covered in-depth in Chapter 6 on event-driven ABS, the *parallel* one in Chapter 5 on time-driven ABS and the *concurrent* strategy is discussed in-depth in Chapter III on parallel ABS. The *actor* strategy is not used in this thesis but its implementation follows directly from the Chapters 5 and III: instead of globally synchronising in the main-thread, a closed feedback-loop is run in every agent thread.

The established approaches to implement ABS follow the object-oriented paradigm (OOP) and solve these problems from this perspective, which is quite well understood by now, as high quality ABS frameworks like RePast [89] prove. In OOP an agent is mapped directly onto an object, encapsulating the agents state and providing methods, which implement the agents' actions. OOP 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 OOP, agents have global access to an environment e.g. through a Singleton or a simple global variable, and can mutate the environments data by direct method calls.

All these language features are not available in FP and we face seemingly severely restrictions like immutable state, recursion, a static type-system. Further we restrict ourselves deliberately to *pure* FP and avoid running in *IO* under all costs. The question is then to solve these problems in FP *and* use the restrictions to our advantage. Depending on the type and model of the ABS we approach these problems slightly different. In the next two chapters we show how to implement both a time-driven ABS using the agent-based SIR model as example and an event-driven ABS using the Sugarscape model as example. In both sections we present fundamental concepts of how to engineer an ABS from a pure FP perspective. This will then be used in subequent chapters to discuss

why one would follow an FP approach, identifying its benefits and advantages over OOP approaches.

Part II:

Towards pure functional ABS

Chapter 5

Pure Functional Time-Driven ABS

In this chapter, we pose solutions to the previously mentioned problems by derive a pure functional approach for time-driven ABS through the example of the agent-based SIR model. 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].

5.1 The SIR model

The *explanatory* SIR model is a very well studied and understood compartment model from epidemiology [73], which allows to simulate the dynamics of an infectious disease like influenza, tuberculosis, chicken pox, rubella and measles spreading through a population.

In this model, people in a population of size N can be in either one of three states *Susceptible*, *Infected* or *Recovered* at a particular time, where it is assumed that initially there is at least one infected person in the population. People interact *on average* with a given rate of β other people per time-unit and become infected with a given probability γ when interacting with an infected person. When infected, a person recovers *on average* after δ time-units and is then immune to further infections. An interaction between infected persons does not lead to re-infection, thus these interactions are ignored in this model. This definition gives rise to three compartments with the transitions seen in Figure 5.1.

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

Figure 5.1: States and transitions in the SIR compartment model.

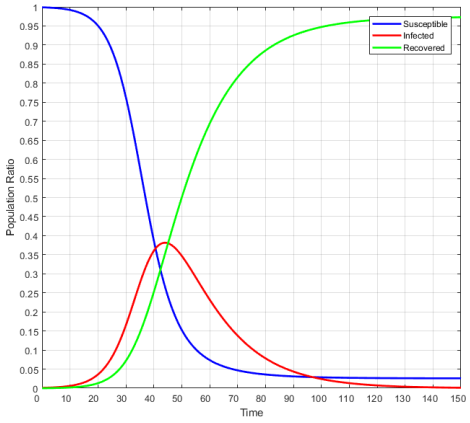


Figure 5.2: Dynamics of the SIR compartment model using the System Dynamics approach. Population Size $N = 1{,}000$, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time-steps. Generated using our pure functional SD approach (see Chapter 18.2).

This model was also formalized using System Dynamics (SD) [104]. In SD one models a system through differential equations, allowing to conveniently express continuous systems, which change over time, solving them by numerically integrating over time, which gives then rise to the dynamics. The SIR model is modelled using the following equation, with the dynamics shown in Figure 5.2 .

$$\begin{aligned} \frac{\mathrm{d}S}{\mathrm{d}t} &= -infectionRate \\ \frac{\mathrm{d}I}{\mathrm{d}t} &= infectionRate - recoveryRate \\ \frac{\mathrm{d}R}{\mathrm{d}t} &= recoveryRate \end{aligned} \tag{5.1}$$

$$\begin{aligned} infectionRate &= \frac{I\beta S\gamma}{N} \\ recoveryRate &= \frac{I}{\delta} \end{aligned} \tag{5.2}$$

The approach of mapping the SIR model to an ABS is to discretize the population and model each person in the population as an individual agent. The transitions between the states are happening due to discrete events caused both by interactions amongst the agents and time-outs. The major advantage of ABS is that it allows to incorporate spatiality as shown in Section 5.4 and

simulate heterogenity of population e.g. different sex, age. This is not possible with other simulation methods e.g. SD or Discrete Event Simulation [140].

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 [15]. 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 4.1.2, in the parallel update-strategy, the agents act conceptually all at the same time in lock-step. This implies that they observe the same environment 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, FP can be used to enforce the correct application of this strategy already on the compile-time level.

In the ABS classification of [80], this model can be seen as an *Interactive 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 and behaviours of other agents and the state of the environment.

5.2 First step: pure computation

As described in Chapter 3.2.4, Arrowized FRP [64] is a way to implement systems with continuous and discrete time-semantics where the central concept is the signal function, which can be understood as a process over time, mapping an input- to an output-signal. Technically speaking, a signal function is a continuation which allows to capture state using closures and hides away the Δt, which means that it is never exposed explicitly to the programmer, meaning it cannot be manipulated. As already pointed out, agents need to perceive time, which means that the concept of processes over time is an ideal match for our agents and our system as a whole, thus we will implement them and the whole system as signal functions.

We start by defining the SIR states as 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 Random Monad because signal functions as implemented in Yampa are not capable of being monadic.

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

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

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 - this is one, very naive way of implementing the interactions between agents.

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

Note the use of $iPre :: a \rightarrow SF\ a\ a$, which delays the input signal by one sample, taking an initial value for the output at time zero. The reason for it is that we need to delay the transition from susceptible to infected by one step due to the semantics of the *switch* combinator: whenever the switching event occurs, the signal function into which is switched will be run at the time of the event 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
      makeContact <- occasionally g (1 / contactRate) () -< ()
      if isEvent makeContact
        then (do
          -- draw random element from the list
          a <- drawRandomElemSF g -< as
          case a of
            Infected -> do
              -- returns True with given probability
              i <- randomBoolSF g infectivity -< ()
              if i
                then returnA -< (Infected, Event ())
                else returnA -< (Susceptible, NoEvent)
            _        -> returnA -< (Susceptible, NoEvent))
        else returnA -< (Susceptible, NoEvent)
```

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

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

An infected agent recovers *on average* after δ time units. This is implemented by drawing the duration from an exponential distribution [15] with $\lambda = \frac{1}{\delta}$ and making the transition to the *Recovered* state after this duration. Thus the infected agent behaves as infected until it recovers, on average after the illness duration, after which it behaves as a recovered agent by switching into *recoveredAgent*. As in the case of the susceptible agent, we use the *occasionally* function to generate the event when the agent recovers. Note that the infected agent ignores the states of the other agents as its behaviour is completely independent of them.

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

```
      let a = event Infected (const Recovered) recEvt
      returnA -< (a, recEvt)
```

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

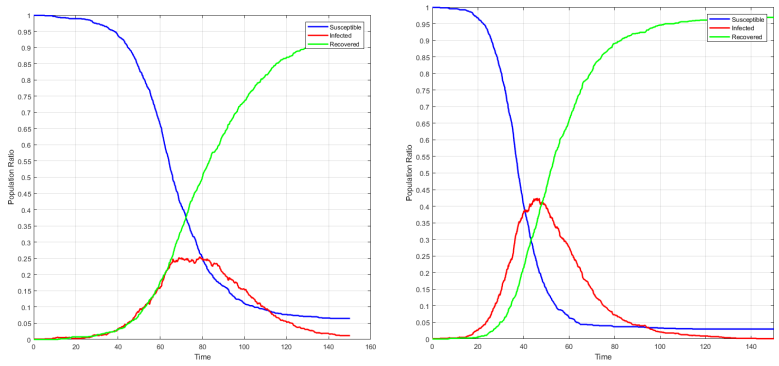
```
runSimulation :: RandomGen g => g -> Time -> DTime -> [SIRState] -> [[SIRState]]
runSimulation g t dt as
    = embed (stepSimulation sfs as) ((), dts)
  where
    steps     = floor (t / dt)
    dts       = replicate steps (dt, Nothing)
    n         = length as
    (rngs, _) = rngSplits g n [] --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

What we need to implement next is a closed feedback-loop - the heart of every agent-based simulation. Fortunately, [88, 28] 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)
```



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

Figure 5.3: 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.

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.

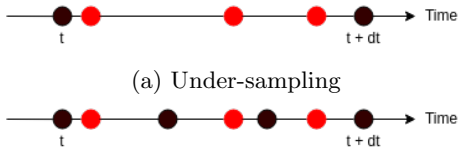
Conceptually, *dpSwitch* allows us to recursively switch back into the *step-Simulation* with the continuations and new states of all the agents after they were run in parallel.

5.2.1 Results

The dynamics generated by this step can be seen in Figure 5.3.

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 which matches the frequency of events generated by *contact rate*. If we choose a too large Δt, we loose events, which will result in wrong dynamics as can be seen in Figure 5.3a. This issue is known as under-sampling and is described in Figure 5.4.

For tackling this issue we have three options. The first one is to use a smaller



(a) Under-sampling

(b) Super-sampling

Figure 5.4: 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.

Δt as can be seen in 5.3b, 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 [138]. We will come back to this approach in the even-driven approach to ABS in Chapter 6.3. 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.

In Yampa there exists a function *embed* which allows to run a given signal-function with provided Δt but the problem is that this function does not really help because it does not return a signal-function. What we need is a signal-function which takes the number of super-samples n, the 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:

```
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 a SF
    tfCont    = superSamplingAux sf'

superSamplingAux :: SF' a [b]
superSamplingAux sf' = SF' tf
  where
          tf0 :: 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 a 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. We could then wrap the occasionally function which would then generate a list of events.

5.2.2 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 - we implement each agent 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 won't 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 the instance of Yampa results in a clear, expressive and robust implementation. State is implicitly encoded, depending on which signal function is active. By using explicit time-semantics with *occasionally* we can achieve extremely fine grained stochastics by sampling the system with small Δt: we are treating it as a truly continuous time-driven agent-based system.

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

So far we have an acceptable implementation of an agent-based SIR approach. What we are lacking at the moment is a general treatment of an environment and an elegant solution to the random number correlation. In the next step we make the transition to Monadic Stream Functions as introduced in Dunai [99], which allows FRP within a monadic context and gives us a way for an elegant solution to the random number correlation.

5.3 Second Step: Going Monadic

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

5.3.1 Identity Monad

We start by making the transition to BearRiver by simply replacing Yampas signal function by BearRivers', which is the same but takes an additional type parameter m, indicating the monadic context. If we replace this type-parameter with the Identity Monad, we should be able to keep the code exactly the same, 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
```

5.3.2 Random Monad

Using the Identity Monad does not gain us anything but it is a first step towards a more general solution. Our next step is to replace the Identity Monad by the Random Monad, which will allow us to run the whole simulation within the Random Monad with the full features of FRP, finally solving the problem of correlated random numbers in an elegant way. We start by re-defining the SIRMonad and SIRAgent:

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

The question is now how to access this Random Monad functionality within the MSF context. For the function *occasionally*, there exists a monadic pendant *occasionallyM* which requires a MonadRandom type-class. Because we are now running within a MonadRandom instance we simply replace *occasionally* with *occasionallyM*.

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

5.3.3 Discussion

Running in the Random Monad solved the problem of correlated random numbers and elegantly guarantees us that we won't have correlated stochastics as

(a) von Neumann (b) Moore

Figure 5.5: Common neighbourhoods in discrete 2D environments of Agent-Based Simulation.

discussed in the previous section. In the next step we introduce the concept of an explicit discrete 2D environment.

5.4 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 [38] which is simply a finite NxM grid with either a Moore or von Neumann neighbourhood (Figure 5.5). Agents are either static or can move freely around with cells allowing either single or multiple occupants.

We can directly map the SIR model to a discrete 2D environment by placing the agents on a corresponding 2D grid with an unrestricted neighbourhood. The behaviour of the agents is the same but they select their interactions directly from the 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 Dynamics approach and for purposes of a more interesting approach, we restrict the neighbourhood to Moore (Figure 5.5b).

We also implemented this spatial approach in Java using the well known ABS library RePast [89], to have a comparison with a state of the art approach and came to the same results as shown in Figure 5.6. 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.

5.4.1 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 Random 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 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 Random 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.

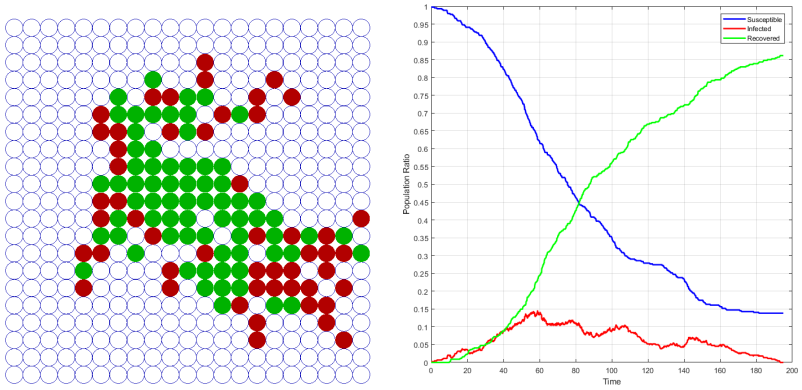
```
simulationStep :: RandomGen g => [(SIRAgent g, Disc2dCoord)]
               -> SIREnv -> SF (Rand g) () SIREnv
simulationStep sfsCoords env = MSF (\_ -> do
  let (sfs, coords) = unzip sfsCoords
  -- run agents sequentially but with shared, read-only environment
  ret <- mapM (`unMSF` env) sfs
  -- 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
```

5.4.2 Results

We implemented rendering of the environments using the gloss library which allows us to cycle arbitrarily through the steps and inspect the spreading of the disease over time visually as seen in Figure 5.6.



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

Figure 5.6: Simulating the agent-based SIR model on a 21x21 2D grid with Moore neighbourhood (Figure 5.5b), a single infected agent at the center and same SIR parameters as in Figure 5.2. 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.

Note that the dynamics of the spatial SIR simulation, which are seen in Figure 5.6b look quite different from the reference dynamics of Figure 5.2. This is due to a much more restricted neighbourhood which results in far fewer infected agents at a time and a lower number of recovered agents at the end of the epidemic, meaning that fewer agents got infected overall.

5.4.3 Discussion

By introducing a structured environment with a Moore neighbourhood, we showed the ABS ability 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.

5.5 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 how small Δt should be. Third it requires one to think about agent interactions in a new way instead of being just method-calls.

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

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

Determinism is also ensured by fixing the Δt and not making it dependent on the performance of e.g. a rendering-loop or other system-dependent sources of non-determinism as described by [100]. Also by using FRP we gain all the benefits from it and can use research on testing, debugging and exploring FRP systems [100, 97].

Also we showed how to implement the *parallel* update-strategy [121] 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.

Our approach can guarantee reproducibility already at compile time, which means that repeated runs of the simulation with the same initial conditions will always result in the same dynamics, something highly desirable in simulation in general. This can only be achieved through purity, which guarantees the absence of implicit side-effects, which allows to rule out non-deterministic influences at compile time through the strong static type system, something not possible with traditional object-oriented approaches. Further, through purity and the strong static type system, we can rule out important classes of run-time bugs e.g. related to dynamic typing, and the lack of implicit data-dependencies which are common in traditional imperative object-oriented approaches.

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.

Currently, the performance of the system does not come close to imperative implementations. We compared the performance of our pure functional approach as presented in Section 5.4 to an implementation in Java using the ABS library RePast [89]. 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 per-

formance 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 [85] than a time-driven one as ours, so the comparison is not completely valid. Still, we have already started investigating speeding up performance through the use of Software Transactional Memory [54, 55], which is quite straight forward when using MSFs. It shows very good results but we have to leave the investigation and optimization of the performance aspect of our approach for further research as it is beyond the scope of this paper.

Despite the strengths and benefits we get by leveraging on FRP, there are errors that are not raised at compile time, e.g. we can still have infinite loops and run-time errors. This was for example investigated in [110] 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.

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 have added further mechanisms of agent interaction which we had to omit due to lack of space.

Chapter 6

Pure Functional Event-Driven ABS

In this chapter we build on the previous discussion of update-strategies in Chapter 4 and the implementation techniques presented in the time-driven approach of Chapter 5 to develop concepts for event-driven ABS in a pure functional way.

In event-driven ABS [85], 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 time-stamp, which indicates the virtual simulation time when it is scheduled. This implies that time could stay constant e.g. when an event is scheduled with a time-delay of 0 the virtual simulation time does not advance. 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 5. Further, we exploit this mechanism to implement direct agent-interactions in pure functional ABS as discussed in the Sugarscape use-case below.

The event-driven approach makes the simulation kernel technically closely related to a Discrete Event Simulation (DES) [140]. Due to the necessity of imposing a correct ordering of events in this type of ABS, we need to step it event by event, with the *sequential* update-strategy being the only feasible one for this type of ABS. Note that there exists also Parallel DES (PDES) [43], which processes events in parallel and deals with inconsistencies by reverting to consistent states - we hypothesize 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 use the Sugarscape model to develop pure functional concepts for event-

driven ABS [1]. We chose this model for the following reasons: 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 they used Object Pascal for programming the agents and C for low-level graphics [7]; the authors explicitly advocate OOP as a good fit to ABS which begged the question whether and how well a pure functional implementation is possible. The Sugarscape model is not a classic event-driven model: in it the agents do schedule events but they don't do this into the future - events in Sugarscape don't have associated time-stamps. Still the underlying concepts are the same as in event-driven ABS and it is trivial to add time-stamps as we will show in an additional section where we implement an event-driven implementation of the previously introduced agent-based SIR model, moving towards a real event-driven ABS with DES character.

6.1 Sugarscape

The seminal Sugarscape Model was one of the first models in Agent-Based Simulation, developed by Epstein and Axtell in 1996 [38]. Their aim was to *grow* an artificial society by simulation and connect observations in their simulation to phenomenon observed in real-world societies. In the model a population of agents move around in a discrete 2D environment, where sugar 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, making it an *exploratory* model.

In the ABS classification of [80], 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 and behaviours of other agents and the state of the environment; and agents can change their behaviour during the simulation through observing their own state, learning and populations can adjust their composition.

The full specification of the Sugarscape model itself fills a small book [38] 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, where in each step 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

[1] The code of all steps can be accessed freely from: `https://github.com/thalerjonathan/phd/tree/master/public/towards/SugarScape/sequential`

position - it is equally probable of being scheduled in any position. The semantics of the model 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 but makes their behaviour appear statistically in parallel.* Every agent executes the following actions, where agents executed after in the same tick, can already see the changes and interactions with preceding agents:

(a) The agent ages by 1 tick. An agent might have a maximum age and when reached will result in the removal of the agent (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 applying the parallel update-strategy.

(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 next step).

(d) Apply pollution of the environment through the agent. Depending on how much the agent has harvested during its movement and consumed 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) TODO why?. 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.

(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.

(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 or immediately. Depending on whether seasons are enabled, 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.

TODO: show 2 screenshots of sugarscape: mating and pollution

Note that depending on the configuration of the simulation, some actions of an agent might be skipped as nearly all actions can be turned on or off e.g. pollution, dying from age, mating, trading, lending, diseases.

6.2 Implementation Concepts

TODO REFINE CONTENT: we can write combinators which encapsulate all the continuation plumbing so it looks in the end very similar to a method call: shortly discuss and show that in Sugarscape. implement sync-interaction combinator: it becomes clear in the type of the function what is going on

In the next sections we derive implementation concepts of pure functional event-driven ABS. Due to the complexity of the Sugarscape model we don't provide a full implementation but only present concepts derived from our implementation and present small parts of the Sugarscape implementation when necessary.

6.2.1 Agent Representation

We follow the same approach as in the time-driven approach of Chapter 5.3 and use an MSF to define our agent due to the following requirements:

- As in the time-driven approach, we need a random number stream within our agents, for which we will make use of the *Rand* monad.

- In the Sugarscape, the agents act on an environment which they both read *and* write. We will make use of the *State* monad for this functionality.
- The agents state in the Sugarscape model is much more complex than in the time-driven example where it was implicit. In this approach the agents have an explicit data-structure which they can mutate whenever they are acting. We will make use of the *State* monad for this functionality.
- For synchronous and one-way agent-interactions, agents send messages to other agents. This is completely different from the time-driven approach where no direct agent-interactions in form of messages where possible because agents were all acting at the same time and synchronous agent-interactions would violate that principle. We will make use of the *Writer* monad for this functionality.
- Due to the agent-interaction through messaging we need a clearly defined concept of agent identity, which is immutable for each agent and fixed at agent-creation time. At the same time we also need to generate new agent identities e.g. when an agent needs to create a new born from mating action. Further we need to access the read-only model configuration which defines if e.g. trading is turned on or off. We will make use of the *Reader* and *State* monad for this functionality.

The interface of an agent changes now substantially with very different inputs and outputs due to the fundamental different model and ABS type. Because we are dealing with events now, it makes very much sense to define the input-type of an agent as the event-type: this indicates that an agent always needs an event to run, to which it will react. As output we need a much richer structure than simply the current state the agent is in as in the time-driven approach: we need to be able to communicate to the simulation kernel whether the agent should be removed from the simulation, what new agents this agent wants to create and what messages it sends to other agents. Additionally we need to communicate the observable properties of an agent to the simulation kernel for visualisation purposes. We will show below how to conveniently construct such an output in a monadic way through the *Writer* monad. Note that many additional inputs and outputs are implicitly covered by the monadic context as will be shown below, e.g. we could also pass the environment as input and returning it as output instead providing it through a *State* monad but the latter is much more convenient to program with.

6.2.1.1 A generic MSF for event-driven ABS

We start with defining the basic types for a general event-driven agent. Besides the obligatory Time and Δt which are defined in this case as *Int* we define an *AgentId* as *Integer* for uniquely identifying agents in the process of messaging. Further an event type is defined which is either a *Tick* with a given Δt or a *DomainEvent* from a given sender with the given event where the type of the

actual *DomainEvent* is generic. As already mentioned we need a way of generating new agent identities which happens through the *ABSState* data structure which holds the next agent-id plus the current virtual simulation time so that agents don't need to keep track of it themselves. Now we can define the initial *AgentT* monad-transformer for which we start with a *StateT* and the previously defined *ABSState*.

Finally we define the type of the Agent MSF as a simple monadic stream function (MSF) with the monadic context *AgentT* and the *ABSEvent* as input. The output is the previously mentioned structure and the observable properties of the agent. The Agent MSF is parametrised by m indicating the type of (additional) monadic context, e indicating the type of the *DomainEvent* and o indicating the type of the observable properties. These types are highly polymorphic and are applicable to a wide range of event-driven ABS models. Note that no explicit type of an environment is used because some models rather omit an explicit environment and have it implicitly encoded in the model itself e.g. a fully connected network of agent-neighbourhoods as in the agent-based SIR model in Chapter 5.2. Further, we also don't provide random-number generator functionality on the type-level at that point because concrete models might opt for a different approach to randomness e.g. providing their own random number generator through a monad-transformer or omitting it altogether. All this optional behaviour is possible because the monadic context of the agents MSF is a transformer which allows to add arbitrary number of layers of behaviour as we will see below: we are polymorphic in the side-effects, which is only possible in a pure functional language like Haskell and can't be achieved in the established OOP approaches to ABS.

```
type Time  = Int
type DTime = Int

type AgentId = Integer

data ABSEvent e = Tick DTime | DomainEvent AgentId e

data ABSState = ABSState
  { absNextId :: AgentId -- holds the next agent-id
  , absTime   :: Time    -- current simulation time
  }

type AgentT m       = StateT ABSState m
type AgentMSF m e o = MSF (AgentT m) (ABSEvent e) (AgentOut m e o, o)

-- definition of a new agent
data AgentDef m e o = AgentDef
  { adId      :: AgentId         -- unique agent-id
  , adSf      :: AgentMSF m e o  -- the agent behaviour function
  , adInitObs :: o               -- the value of the initial observable properties
  }

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)
  }
```

What is striking, and underlines the even-driven approach, is that we are using an MSF and not a monadic SF (Streamfunction) from Bearriver as we did in Chapter 5.3. This means that there is no inherent notion of a Δt available to the agent, which is precisely what we need in event-driven ABS. Time only advances through specific events, in this case the *Tick DTime* event.

6.2.1.2 Parametrising for Sugarscape

For our Sugarscape implementation, we need to parametrise the polymorphic types by concrete types for m, e and o. Further, we need access to a random-number stream and we want to make the unique agent-id, the environment and the model-configuration explicit in the types. We start with defining both the environment and model-configurations as new data-structures (see below) and the data-type for an individual agent-state and the observable properties of the agent. Note that we explicitly decided to use two different types for the agent-state and its observable properties - after all we could have used the type of the agent-state also as the type for its observable properties. With two different types we make the distinction between the (read/write) local agent-state and the (read-only) observable properties very clear. Also, it allows us to expose a much smaller subset of the agent-local state to the visualisation and export layer of the simulation. This gives us the ability to hide away agent-state fields, which are implementation detail or unimportant for visualisation and exporting purposes.

```
data SugEnvironment     = ...
data SugarScapeScenario = ...

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

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

As already mentioned, we can add additional behaviour through the monad transformer of the agents MSF. We use this to make the Sugarscape environment explicit in the types and add random number-functionality. We simply start out with a *StateT* transformer for the Sugarscape environment, because the agent

can read and write it and then terminate the transformer by adding the *Rand* monad for the random-number functionality. We then simply parametrise the existing monad transformer of *AgentMSF* with this additional transformer stack to arrive at the final type of the Sugarscapes agent MSF.

```
data SugEvent = ... -- all events of the sugarscape model

type SugAgentMonad g = StateT SugEnvironment (Rand g)
-- the sugarscape agent MSF with the monadic type expanding to:
-- AgentMSF (SugAgentMonad g)
-- =>
-- MSF (AgentT (SugAgentMonad g))
-- =>
-- MSF (StateT ABSState (SugAgentMonad g))
-- =>
-- MSF (StateT ABSState (StateT SugEnvironment (Rand g)))
type SugAgentMSF g = AgentMSF (SugAgentMonad g) SugEvent SugAgentObservable
```

Finally we define the type of the top-level agent-behaviour function. As already noted, we want to make the unique agent-id and the model-configuration explicit, so it will be passed as an argument to the function. Further we pass the initial agent state as an additional input. This function will then construct a corresponding initial MSF (see below) and returns it. Thus the intended behaviour is as follows: an agent is defined in terms of this top-level function, which will be passed to the simulation kernel (see below) which will in turn run this function to get the agents initial MSF which will then be run subsequently (see below).

```
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 won't get sneaked in: the transformer-stack of the agents MSF is terminated through the *Rand* monad - it is simply not possible to add other layers. In the next step we look at how the agent handles and send events - that is we are looking at an implementation of a *SugarScapeAgent* function.

6.2.1.3 Agent-Local abstractions

The top-level *SugarScapeAgent* function encapsulates the whole agent-behaviour, which is completely driven by events, passed in as input. We now 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 is locally encapsulated and hidden from the simulation kernel. We want to achieve the following functionality, local to the agent: encapsulation of the agents' state which should still be read/writeable, sending of events and read-only access to the agents unique id and the model configuration.

To implement the local encapsulation of the agents' state is straight forward with MSFs as they are continuations, which allow to capture local data using closures. Fortunately we don't need to implement the low-level plumbing, as dunai provides us with the a feedback function: *feedback :: Monad m ⇒ c → MSF m (a, c) (b, c) → 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).

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

Next we want to write a monadic function which handles our event. As already pointed out this function must be able to manipulate the agent-local state we just encapsulated through *feedback*, sending events and accessing the model configuration and the unique agent-id. Providing the local agent state is trivially done using the *State* monad. Providing the model configuration and the unique agent-id is trivially done using the *Reader* monad. For providing the event sending function we opted for the *Writer* monad: as shown above in the generic types, an agent outputs (amongst others) a list of events it wants to send to receivers. Thus we start with an empty initial list and provide functionality to append to this list - which is exactly what the *Writer* monad does: it allows to write to a data-type which implements the *Monoid* class. The lists in Haskell are instances of the *Monoid* class, thus this is covered for us already. Instead of only covering the event sending functionality with the *Writer* monad, we extend it to the whole *SugAgentOut* type which respective fields are instances of the *Monoid* class themselves thus writing an instance of the *Monoid* class for *SugAgentOut* is trivial. We thus define the following monad which is local to the agent and is only used *within AgentMSF*.

```
type SugAgentMonadT g = AgentT (SugAgentMonad g)
-- FULLY EXPANDS TO:
-- AgentT (SugAgentMonad g)
--  =>
-- AgentT (StateT SugEnvironment (Rand g))
--  =>
-- StateT ABSState (StateT SugEnvironment (Rand g))

type AgentLocalMonad g = WriterT (SugAgentOut g)
                           (ReaderT (SugarScapeScenario, AgentId)
                             (StateT SugAgentState (SugAgentMonadT g)))
-- FULLY EXPANDS TO (one step replacement of SugAgentMonadT):
-- 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 with less - similar to the *Control.Monad* approach. 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 normal input to the event handler. Note that *WriterT* does not need an initial value, it will be provided through the *Monoid* instance of *AgentOut*.

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

generalEventHandler :: RandomGen g => EventHandler g

sugObservableFromState :: SugAgentState -> SugAgentObservable
```

6.2.1.4 Handling and sending of Events

Now we can handle 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 to the agent (controlled by the simulation kernel) and part local to the agent (controlled by the agent itself). The layers are the following (outer to inner):

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 model configuration and unique agent-id read-only.

3. StateT SugAgentState: *agent-local*, provides the local agent state for reading and writing.

4. StateT ABSState: *global*, provides unique agent-ids for new agents and the current simulation time. The usage of a *StateT* is slightly flawed here because it provides too much power: the current simulation time should be read-only to the agent. Drawing the next agent-id involves reading the current id and writing the incremented value, thus technically it is a *StateT* but ideally we would like to hide the writing operation and only provide a *read-current-and-increment* operation. A possible solution would be to provide the current simulation time through a *ReaderT* and the new agent-id through a new monad which uses the *StateT* under the hood, like the *Rand* monad.
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.

The event handler 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 *agentMSF* 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 (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).

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

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, as used in Chapter 5.2. It allows to swap out the event-handler for a different one, which is the foundation for the synchronous agent-interactions, where it will be discussed more in-depth.

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 (see *generalEventHandler*) 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
      then return Nothing
      else do
        sugLvl  <- agentProperty sugAgSugarLevel
        spiLvl  <- agentProperty sugAgSpiceLevel
        metab   <- agentProperty sugAgSugarMetab
        vision  <- agentProperty sugAgVision
        culTag  <- agentProperty sugAgCultureTag
        imSysGe <- agentProperty sugAgImSysGeno

        return Just (sugLvl / 2, spiLvl / 2, metab, vision, culTag, imSysGe)

  sendEventTo sender (MatingReply acc)
```

Next we look at how synchronous agent-interactions work - that is we look closer at the mating-mechanism which requires multiple synchronous interaction steps, which need to happen within the same simulation tick and both agents must not engage with other agents.

6.2.1.5 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 asynchronous agent-interactions where one agent sends a message to another agent but does not await an answer within the same tick. This one-directional asynchronous 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 6.1 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 having 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 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 3.2.4 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 *continueWithAfter* only applies the new MSF in the *next* step, *switch* runs the new MSF immediately. 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))
```

We can now look at the Tick handling function. It returns a 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, 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. When calling *agentDisease* we are passing a default

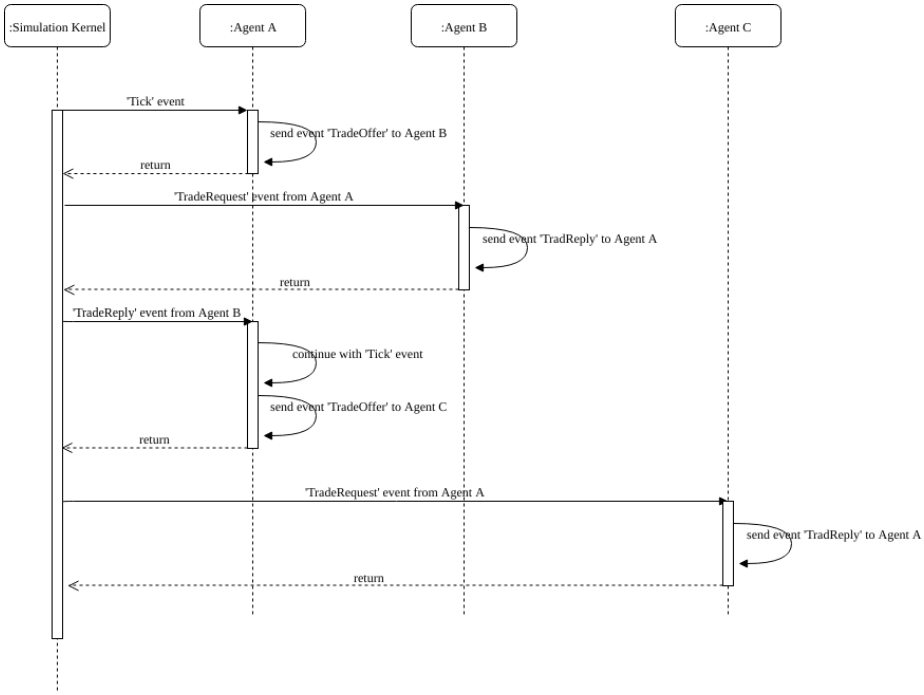


Figure 6.1: Sequence diagram of synchronous agent-interaction with the trading use-case. Upon the handling of the 'Tick' event, Agent A looks for trading partners and finds Agent B within its neighbourhood and sends a 'TradingOffer' message. Agent B replies to this message and Agent A continues with 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 and is thus not included fully in this diagram.

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
  agentAgeing dt

  harvestAmount <- agentMove
  metabAmount   <- agentMetabolism
  agentPolute harvestAmount metabAmount

  ifThenElseM
    (starvedToDeath `orM` dieOfAge)
    (do
      agentDies agentMsf
      return Nothing)
    -- 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)
```

6.2.2 Environment Representation

Environment representation is quite simpl, after we have solved the problem of how the agent can access it by using StateT. It is only a matter of selecting the right data-structure and writing domain-specific functions of the corresponding model to mutate the environment. Initially we used an indexed array from the *array* package. This data-structure has excellent read performance but in performance tests it was shown that it has 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. We used the discrete 2d-coordinates to

map the environment cells to a unique index. This solved both the performance and memory leak issues completely.

6.2.2.1 Environment Behaviour

We must make a clear distinction between the environments data-structure and 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 (see below).

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

1. *Passive read-only* - implemented in Chapter 5.2, 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 5.2 this is enforced at compile time by simply excluding it from the data an agent can emit. 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* - implemented in Chapter 5.4. 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 random sequentially.

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.

6.2.3 The simulation kernel

The simulation kernel is the heart of the simulation mechanism: it holds the full simulation state and iterates the simulation step-by-step through virtual time. The full simulation state is comprised of the following:

- A mapping of agent MSFs to their id.
- A list of the current observable state of each agent.
- The state of the environment.
- A random-number generator.
- A step counter.

In each step the simulation state is used to compute the next step of the simulation and is thus updated after a state has been computed. Using pure functional programming, where we have persistent data-structures and immutable data, we can easily keep record of the simulation state for each step for debugging purposes e.g. instead of overwriting the state after each step, we can keep the state of each step and can go backwards and forwards in the time-series of steps.

The very heart of the simulation kernel is the step function, which computes the next step of the simulation. It is a *pure* function, taking the current simulation state and returns a new simulation state together with the output of the new step. The output of a step is the current simulation time, the number of events processed, the environment state and a list of the observable states of all agents.

```
type SimStepOut = (Time, Int, SugEnvironment, [AgentObservable SugAgentObservable])
-- Need RandomGen g because holding a random-number generator in SimulationState
simulationStep :: RandomGen g => SimulationState g -> (SimulationState g, SimStepOut)
```

The working horse behind *simulationStep* is another *pure* function which processes all events scheduled in the current step. It takes the list of events to process and the simulation state and returns the simulation state.

```
type EventList  = [(AgentId, ABSEvent SugEvent)]  -- from, to, event
processEvents :: RandomGen g => EventList -> SimulationState g -> SimulationState g
```

To void getting too technical and mixed up in implementation details, we provide the internals of *processEvents* in terms of steps done instead of code. The function does the following:

1. Extract the event at the front of the *EventList*. In case the list is empty, return the simulation state.
2. Look up the receivers' agent MSF in the agent mapping of the simulation state.

3. If the receiver was not found, the function ignores this and processes the next event through a recursive call.
4. If the receiver is found: run the agents' MSF and get the result.
5. Update the agents' current MSF in the mapping (note that an MSF produces a new MSF as a result!).
6. Update the agents' current observable state.
7. Handle the agents' output: create new agents and remove the agent from the simulation if it killed itself.
8. Prepend the events the agent has emitted through its output to the front *EventList* and do a recursive call to *processEvents*.

The initial *EventList* passed to *processEvents* is a list with *Tick* events scheduled for every agent, in random order. It is very important to understand that the events an agent emits, are prepended to the front of the *EventList*. This ensures that those events are processed next, which is of utmost importance for a correct working of the synchronous agent-interactions. 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.

Finally we have a look at how to actually run an agents' MSF using the function *runAgentSF*. 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.

```
runAgentSF :: RandomGen g          -- ^ RandomGen typeclass, 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)
runAgentSF 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.

6.3 Event-Driven SIR

This short section shows how to implement the SIR model, as introduced in Chapter 5.1, with an event-driven approach. This is in stark contrast to the time-driven implementation in Chapter 5.2. The 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 below.

To keep this section simple, we reduce code-examples as far as possible and focus on the most fundamental differences to the approach used in Sugarscape. An agent in the event-driven SIR has no output (that is: the return-type of the MSF is the empty tuple ()), because the SIR model is much less dynamic than the Sugarscape one: agents don't spawn other agents and agents can't die. Further there is no environment (see Chapter 5.2 how to add an environment to the SIR model) and observable dynamics happen not through agent-output but through side-effects.

The very heart of this implementation is the simulation state, which holds (amongst others) a priority queue and a tuple with the number of susceptible, infected and recovered agents. Agents schedule events with a time-stamp and receiver agent id, using this priority queue, which the simulation kernel processes then in order of the time-stamps to run the next agent. This is conceptionally very close to the Sugarscape implementation but events have now an additional time-stamp, which indicates the time when they are about to be scheduled - the priority queue is sorted according to the time-stamps and the simulation kernel simply processes them in order. When agents change their state they also increment / decrement the number of susceptible / infected / recovered agents, depending on which transition they make.

This makes the structure of the whole implementation much smaller than the one of Sugarscape: there is no local monad transformer stack to the agent, only a global one, which holds the simulation state as described above and the random-number monad. To get a feeling on the different approach between the Sugarscape and the SIR we show the initial function of the SIR agent. It is called by the simulation kernel to schedule initial events, adjust the simulation state and get the agents MSF.

```
-- | A sir agent is in one of three states
sirAgent :: RandomGen g
         => SIRState    -- ^ the initial state of the agent
         -> SIRAgent g  -- ^ the continuation
sirAgent Susceptible aid = do
    modifyDomainState incSus -- increment number of susceptible agents
    scheduleEvent aid MakeContact makeContactInterval -- schedule make contact event to self
    return (susceptibleAgent aid) -- return susceptible MSF
sirAgent Infected aid = do
    modifyDomainState incInf -- increment number of infected agents
    dt <- lift (randomExpM (1 / illnessDuration)) -- draw random illness duration
    scheduleEvent aid Recover dt -- schedule recovery to self
    return (infectedAgent aid) -- returns infected MSF
sirAgent Recovered _ = do
```

```
modifyDomainState incRec -- increment number of recovered agents
return recoveredAgent -- return recovered MSF
```

In the next sections we have a quick look at how we translate the time-driven susceptible, infected and recovered behaviours of into event-driven behaviours.

6.3.1 Susceptible Agent

We use the same switch mechanism of making the transition from a susceptible to an infected agent in case of an infection but how a susceptible agent gets infected works now different:

- A susceptible agent initially schedules a *MakeContact* event with $\Delta t = 1$ to itself.
- When receiving *MakeContact*, the agent sends a *Contact* event to 5 random other agents with $\Delta t = 0$. This will result in these events to be scheduled immediately. Further the agent schedules *MakeContact* with $\Delta t = 1$ to itself.
- When the agent receives a *Contact* event, it checks if it is from an infected agent. If the event is not from an infected agent, it ignores it. Otherwise it becomes infected with a given probability. In case of infection the agent decrements the number of susceptible and increments the number of infected agents.

6.3.2 Infected Agent

We use the same switch mechanism of making the transition from an infected to a recovered agent after the illness duration. The main difference is that agents send *Contact* events to each other to indicate that contacts have happened. Thus susceptible and infected agents need to react to incoming *Contact* events.

- An infected agent initially schedules a *Recover* event with a random Δt (following exponential distribution) to itself.
- When the agent receives a *Contact* event, it checks if it is from a susceptible agent. 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 $\Delta t = 0$.

6.3.3 Recovered Agent

The recovered agent does not change any more, reacts to no incoming events and schedules no events - it stays constant forever and thus outputs the empty tuple forever.

6.3.4 Reflections

Transforming a time-driven into an event-driven approach should always be possible because the ability to schedule events with time-stamps 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.

The event-driven implementation from this Chapter is around 60 - 70% faster than the time-driven implementation from Chapter 5.2, which is non-monadic and uses the FRP library Yampa. For the monadic time-driven approach of Chapter 5.4 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 5.4. 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.

6.4 Discussion

This section takes a step back and reflects on various points worth noting in the event-driven approach.

6.4.1 A similar approach

After having finished our implementation, we realised that the work of [16] aimed to solve a similar problem which we also had to in this approach. 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" [16]. Note that the authors of [16] define their agents with a polymorphic agent-state type s, which

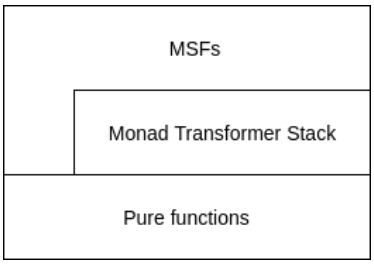


Figure 6.2: The architecture as a 3-layered system.

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. TODO: can we really say this in a thesis?

6.4.2 Layered architecture

The approach is designed as a 3-layered architecture, see Figure 6.2:

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.

6.4.3 Imperative nature

Both event-driven use-cases (Sugarscape and SIR) makes 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. We agree that our approach is just one way of implementing ABS in FP but we think we have come a long way thus making our approach quite valuable even if there might be other approaches like shallow EDSLs or freer monads (see Chapter 20). 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 [2]: *"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.

6.4.4 Multiple types of agents

TODO shortly mention, that we can also do different types of agents: add to the data declaration of observable output an additional case. unfortunately 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 stack.

6.4.5 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 as shown in the short section 6.3 on the event-driven SIR.

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 OOP, which does the same. 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.

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

Chapter 7

The structure of ABS computation

generalising the structure of agent computation - with our case studies we explore them in a more practical / applied way and in this chapter we extract and distil the general concepts and abstractions behind agent computation: how can ABS, which is pure computation, can be seen structurally? This gives the ABS field for the first time a deeper understanding of the deeper structure of the computations behind agent-based simulation, which has so far always been more ad-hoc without a proper, more rigorous formulation.

pure functional computation with effects can be seen as computations over some data-structure where the data-structure defines the structure of the computation as well e.g. monoids, applicatives, monads, traversable, foldable

Note that agent-based simulation is almost always entirely pure computation without the need for direct, synchronous user-interaction or impure IO. When IO is really needed we can keep purity by creating IO actions and pass them to the simulation kernel which executes them and communicates the result back if needed - in this case only the simulation kernel needs to run in IO monad but not the agents and the environment computations.

agentout as monoid with writer: solves the Problem of iteratively constructing it the output during an event.

BUT: isnt our approach similar to the early days IO of Haskell with continuations? if this is the case we should be able to get the direct method style by writing an agent monad?

NOTE: "And a closure is just a primitive form of object: the special case of an object with just one method." https://www.tedinski.com/2018/11/20/message-oriented-programming.html

TODO: this is still research which needs to be done by reading the papers below and reflecting and understanding on co-monads and my implementations in general.

TODO: can we derive an agent-monad?

TODO: what about comonads? read essence dataflow paper [126]: monads not capable of stream-based programming and arrows too general therefor comonads, we are using msfs for abs therefore streambased so maybe applicable to our approach/agents=comonads. comonads structure notions of context-dependent computation or streams, which ABS can be seen as of. this paper says that monads are not capable of doing stream functions, maybe this is the reason why i fail in my attempt of defining an ABS in idris because i always tried to implement a monad family. TODO: stopped at comonad section, continue from there. TODO understand comonads: https://www.schoolofhaskell.com/user/edwardk/cellular-automata and https://kukuruku.co/post/cellular-automata-using-comonads/ and https://chshersh.github.io/posts/2019-03-25-comonadic-builders

TODO: Conal Elliott has examined a comonadic formulation of functional reactive programming http://conal.net/blog/posts/functional-interactive-behavior

TODO: comonads https://fmapfixreturn.wordpress.com/2008/07/09/comonads-in-everyday-life/

TODO: comonads are objects very important and closely related http://www.haskellforall.com/2013/02/you-could-have-invented-comonads.html

TODO: if conal elliott can make a comonadic formulatin of FRP and comonads are objects, then i guess i am very close to a pure functional representation of objects? pure functional objects?

independent of time-driven or event-driven, our agents are MSFs.

in fact i am deriving pure functional objects

TODO: i have the feeling that co-algebras might be an underlying structure, which in CS come up in infinite streams - ABS can be seen as this where the agents are such streams with their output and potentially running for an infinite time, depending on the model. Ionescus thesis might reveal more information / might be an additional source on that.

In general it is easy to see why agents can not be represented by pure functions: they change over time. This is precisely what pure functions cannot do: they can't rely on some surrounding context / or on history - everything what they do is determined by their input arguments and their output. In general we have two ways of approaching this: we either have the agents changing data and behaviour internalised as we did in the previous chapters or we externalise it e.g. in the simulation kernel and provide all necessary information through arguments which was the case in the sugarscape environment.

TODO: FREE MONADS

7.1 A Functional View

Due to the fundamentally different approaches of FP, an ABS needs to be implemented fundamentally differently, compared to established OOP approaches. We face the following challenges:

1. How can we represent an Agent, its local state and its interface?
2. How can we implement direct agent-to-agent interactions?

3. How can we implement an environment and agent-to-environment interactions?

7.1.1 Agent representation

The fundamental building blocks to solve these problems 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 are functions which allow to encapsulate the execution state of a program by capturing local variables (known as closure) and pick up computation from that point later on by returning a new function. As an illustratory example, we implement a continuation in Haskell which sums up integers and stores the sum locally as well as returning it as return value for the current step:

```haskell
-- define the type of the continuation: it takes an arbitrary type a
-- and returns a type a with a new continuation
newtype Cont a = Cont (a -> (a, Cont a))

-- an instance of a continuation with type a fixed to Int
-- takes an initial value x and sums up the values passed to it
-- note that it returns adder with the new sum recursively as
-- the new continuation
adder :: Int -> Cont Int
adder x = Cont (\x' -> (x + x', adder (x + x')))

-- this function runs the given continuation for a given number of steps
-- and always passes 1 as input and prints the continuations output
runCont :: Int -> Cont Int -> IO ()
runCont 0 _ = return () -- finished
runCont n (Cont cont) = do -- pattern match to extract the function
  -- run the continuation with 1 as input, cont' is the new continuation
  let (x, cont') = cont 1
  print x
  -- recursive call, run next step
  runCont (n-1) cont'

-- main entry point of a Haskell program
-- run the continuation adder with initial value of 0 for 100 steps
main :: IO ()
main = runCont 100 (adder 0)
```

We implement an agent as a continuation: this lets us encapsulate arbitrary complex agent-state which is only visible and accessible from within the continuation - the agent has exclusive access to it. Further, with a continuation it becomes possible to switch behaviour dynamically e.g. switching from one mode of behaviour to another like 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.

The fact that we design an agent as a function, raises the question of the interface of it: what are the inputs and the output? Note that the type of the function has to stay the same (type a in the example above) although we might

switch into different continuations - our interface needs to capture all possible cases of behaviour. The way we define the interface is strongly determined by the direct agent-agent interaction. In case of Sugarscape, agents need to be able to conduct two types of direct agent-agent interaction: 1. one-directional, where agent A sends a message to agent B without requiring agent B to synchronously reply to that message e.g. repaying a loan or inheriting money to children; 2. bi-directional, where two agents negotiate over multiple steps e.g. accepting a trade, mating or lending. Thus it seems reasonable to define as input type an enumeration (algebraic data-type in Haskell, see example below) which defines all possible incoming messages the agent can handle. The agents continuation is then called every time the agent receives a message and can process it, update its local state and might change its behaviour.

As output we define a data-structure which allows the agent to communicate to the simulation kernel 1. whether it wants to be removed from the system, 2. a list of new agents it wants to spawn, 3. a list of messages the agent wants to send to other agents. Further because the agents data is completely local, it also returns a data-structure which holds all *observable* information the agent wants to share with the outside world. Together with the continuation this guarantees that the agent is in full control over its local state, no one can mutate or access from outside. This also implies that information can only get out of the agent by actually running its continuation. It also means that the output type of the function has to cover all possible input cases - it cannot change or depend on the input.

```
type AgentId    = Int
data Message    = Tick Int | MatingRequest AgentGender ...
data AgentState = AgentState { agentAge :: Int, ... }
data Observable = Observable { agentAgeObs :: Int, ... }
data AgentOut   = AgentOut
  { kill       :: Bool
  , observable :: Observable
  , messages   :: [(AgentId, Message)] -- list of messages with receiver
  }
-- agent continuation has different types for input and output
newtype AgentCont inp out = AgentCont (in -> (out, AgentCont inp out))
-- taking the initial AgentState as input and returns the continuation
sugarscapeAgent :: AgentState -> AgentCont (AgentId, Message) AgentOut
sugarscapeAgent asInit = AgentCont (\ (sender, msg) ->
  case msg of
    agentCont (sender, Tick t) = ... handle tick
    agentCont (sender, MatingRequest otherGender) = ... handle mating request)
```

7.1.2 Stepping the simulation

The simulation kernel keeps track of the existing agents and the message-queue and processes the queue one element at a time. The new messages of an agent are inserted *at the front* of the queue, ensuring that synchronous bi-directional messages are possible without violating resources constraints. The Sugarscape model specifies that in each tick all agents run in random order, thus to start the agent-behaviour in a new time-step, the core inserts a *Tick* message to each

agent in random order which then results in them being executed and emitting new messages. The current time-step has finished when all messages in the queue have been processed. See algorithm 7.1.2 for the pseudo-code for the simulation stepping.

```
input  : All agents as
input  : List of agent observables
shuffle all agents as;
messageQueue = schedule Tick to all agents;
agentObservables = empty List;
while messageQueue not empty do
    msg = pop message from messageQueue;
    a = lookup receiving agent in as;
    (out, a') = runAgent a msg;
    update agent with continuation a' in as;
    add agent observable from out to agentObservables;
    add messages of agent at front of messageQueue;
end
return agentObservables;
```

Algorithm 1: Stepping the simulation.

7.1.3 Environment and agent-environment interaction

The agents in the Sugarscape are located in a discrete 2d environment where they move around and harvest resources, which means the need to read and write data of environment. This is conveniently implemented by adding a State side-effect type to the agent continuation function. Further we also add a Random effect type because dynamics in most ABS in general and Sugarscapes in particular are driven by random number streams, so our agent needs to have access to one as well. All of this low level continuation plumbing exists already as a high quality library called Dunai, based on research on Functional Reactive Programming [60] and Monadic Stream Functions [99, 98].

Part III:

Parallel computation

TODO: we have looked into this also because we think our implementations are too slow and because pure FP is notoriously slow. This chapter can also be seen as an attempt on overcoming this problem

Pure functional programming as in Haskell is well known and accepted as a remedy against the difficulties and problems of parallel computation [61]. The reason for it is clear: immutable data and explicit control of side-effects removes a large class of bugs due to data-conflicts, data-races. A fundamental benefit and strength of Haskell is, that it clearly distinguishes between parallelism and concurrency *in its types* [71]. 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 cures. 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 [76]. Those parts *can* be run in parallel which as a consequence *might* give rise to asynchronous, non-deterministic events [1].

  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

[1] 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.

> 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* [2].

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 speed up 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 properly understood one can decide whether parallelism or concurrency is applicable - or none at all because the problem is actually completely sequential. As D. Knuth famously put it: *"Premature optimisation is the root of all evil"*, thus, when we see adding parallel computation as one way of optimising a problem, we need hard facts instead of wild guesses.

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 hypothesize that parallelism will allow us to retain *all* static guarantees about reproducibility *and* gives us a noticeable speed up. Further we hypothesize, that in concurrency we might see a bigger speed up 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*.

[2] 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 8

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 [82] 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 5.2, 5.4 and Sugarscape 6.2 and compare their performance over the original sequential approaches.

8.1 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 constructor or lambda 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 basically thunks which get 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 even though evaluating in parallel through sparks is extremely cheap, it still has some overhead. Thus, the workload 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.

8.1.1 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 5.2. 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 e.g. using map. Now we should be able to add parallelism without needing to re-implement *dpSwitch* which is the function which runs the agents in parallel (Also re-implementing switch functions would not get us very far because of WHNF evaluation it is the wrong end to start parallel evaluation: probably only the arguments would be evaluated but not the agent behaviour.)

The solution is to add evaluation parallelism in the agent-output collection phase: where the recursive switch into the *stepSimulation* function happens. There we use a 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 8.3.1 below.

8.2 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 TODO: cite A Monad for Deterministic Parallelism Marlow paper. 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 with this it is also 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* typeclass. This simply means that a value of this type can be fully evaluated, not just to WHNF but to evaluate the full expression the value represents.

8.2.1 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 5.4 and the Sugarscape of Chapter 6.2. An important fact is that if the Par monad is used, it has to be the innermost monad because it cannot be a transformer. This is emphasised by the fact that there exists no ParT transformer instance, like for other monads (e.g. StateT, RandT, ReaderT,... we used in the Sugarscape chapter). Making the Par monad a transformer would have (probably) the meaning 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*. Thus follows that 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 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.

8.3 Case-Studies

In this section we go a little bit more into detail how we applied the parallelism concepts as already outline above to our use-cases from Chapters 5.2, 5.4 and Sugarscape 6.2. We only show briefly 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.

8.3.1 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 resulted in the best performance increase turned out to depend 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 (3) 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 8.1: 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 through 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 out (more or less) 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 8.1. All timings were measured with 1000 agents running for 150 time-steps, and $\Delta = 0.1$. We performed 8 runs and report the 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 speed up of 4.24 on 8 cores - parallelism clearly pays off here, especially after 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 Ram (plugged in).

the IO is decoupled from the computation, making the latter one completely CPU bound and resulting in an impressive speed-up 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 speed up factor of roughly 2.6. The combined approach (4) then shows that we actually can have the substantial speed up 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 [82] 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 speed up 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 speed up 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))
```

8.3.2 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 Random 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 hypothesize that our approach is doomed from the beginning and that we will not see any speed up 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 [82].

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 speed up. 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 innermost 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 an impressively *reduced* performance, compared to a sequential implementation, due do massive overhead and too fine-grained parallelism.

This leaves us basically without any options of parallelism for the monadic SIR model, we will come back to this use-case in the concurrency section, where we will show that by using concurrency it is possible to achieve a substantial speed up by orders of magnitude.

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

8.3.3 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 speed up - 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.

This leaves us only with a very minor thing we could optimize in the Sugarscape model: 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 [82] shows that it is indeed possible through a combination of the Par monad and the *Applicative* typeclass. The *IntMap* provides the function *traverseWithKey :: Applicative t ⇒ (Key → a → t b) → IntMap a → 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, 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 speed up it provides.

Another thing we can parallelise is the computation of the pollution diffusion which uses a standard 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 performance improvements in orders of magnitude when we employ concurrency instead of parallelism.

8.4 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, pure computations. Our approaches shown in the Part II make this very explicit: the top level functions can always be made pure computations because we are ruling out IO (so far) 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 hypothesize 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 as well, one can use the combined ParRand monad which provides both.

8.5 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 5.2. The speed up stemmed from the fact that the agents ran indeed in parallel as our original goal was, thus resulting in a significant speed up 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 meaning of a Monad is).

We further showed how to apply parallelism *within* an SIR agent and for updating the environment of the Sugarscape in parallel using the *Par* monad. It didn't show any speed up 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 speed ups 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 [82] for a more in-depth discussion of both 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 the only possible within a concurrent context, 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 speed up compared to a lock-based approach based on IO.

Chapter 9

Concurrent ABS

TODO REFINE ARGUMENT: in this chapter we asume that concurrent execution has no influence. we can use the techniques of property-based testing to check that but we leave that for further research.

TODO: Concurrency is a control-flow oriented concern. Parallelism is a data-flow oriented concern. Using STM allows us to treat the concurrent problem within an agent as a data-flow oriented one

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 - 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 guar-

antees 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 [35] 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 5.1) and Sugarscape (Chapter 6.1) 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.

Note that there exists also the actor model of concurrency, which is especially well suited to implement concurrent applications in functional languages. We give a short overview over it, existing research and its use in ABS in the section 20.3 but leave it for further research as it has very different implications, which are beyond the focus of this thesis.

TODO: shortening, no comparison to io or repast: the story is "concurrency with compile time guarantees", only mention that an io based single lock performs much worse in sir and slightly worse in sugarscape. leave Array i IORef for further research

9.1 Software Transactional Memory

Software Transactional Memory (STM) was introduced by [111] 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 block rolled-back and automatically restarted.

9.1.1 STM in Haskell

The work of [54, 55] added STM to Haskell, which was one of the first programming languages to incorporate STM into its main core and added the ability to composable operations. 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* context. 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 an *STM* context. 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* a block of actions without any visible effects is only possible due to the nature of Haskells type-system: by restricting the effects to STM only, ensures that no uncontrolled effects, which cannot 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 block 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* block. 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 block will be run again. This is useful to await the arrival of data in a *TVar* or put more concretely, to block on arbitrary conditions.
- *orElse :: STM a → STM a → STM a* - Allows to combine two blocking operations where either one is executed but not both. The first operation 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 an *IO* context. It takes the STM action which returns a value of type *a* and returns an *IO* action which returns a value of type *a*. This *IO* action can only be executed within an *IO* context, 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: whether other threads have written to memory it has read.

In the paper [56] 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 [101] 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.

9.1.2 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 innermost 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 innermost 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 distinguish 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 = 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 innermost monad in a stack with a *StateT* transformer. When the *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 s
  -- atomically run the STM block
  (a, s') <- atomically ret

  putStr ("final StateT state     = " ++ show s')
  putStrLn (", 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 block are persistent and can obviously not be rolled back - Debug.trace is an IO action masked as pure using *unsafePerformIO*.

9.2 STM in ABS

In this section we give a short overview of how we apply STM in our ABS. In both case-studies we fundamentally follow a time-driven, parallel approach as introduced in Chapter 4.1.2, 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 9.1 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 is 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

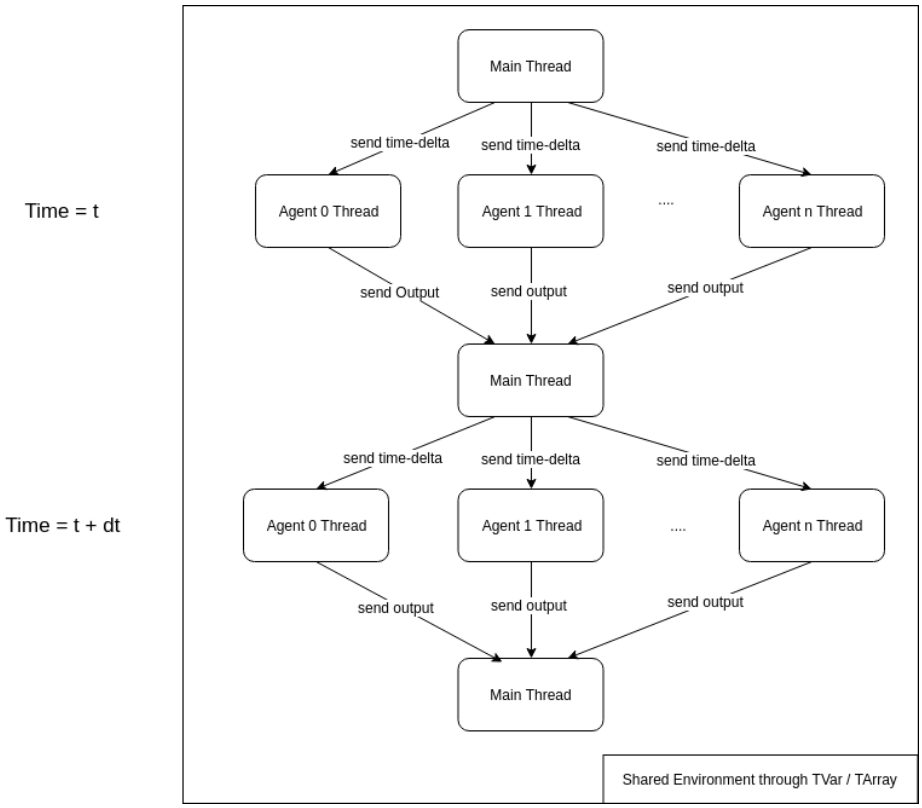


Figure 9.1: Diagram of the parallel time-driven lock-step approach.

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.

9.2.1 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.

The first step is to simply add the *STM* monad on the *innermost* level to

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 Signal Function. 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 signal function 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
              -- updated the environment through the TVar
      arrM_ (lift $ lift $ modifyTVar env (changeCell coord Recovered)) -< ()
      returnA -< (Recovered, Event ()))
    else returnA -< (Infected, NoEvent)
```

The agent thread is straight forward: 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 signal function
            -> g                -- ^ Random-number 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

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

Computing a simulation step is now trivial within the main-thread: all agent threads *MVar*s are signalled to unblock followed by an immediate block on the *MVar*s 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
               -> [MVar DTime]
               -> [MVar SIRState]
               -> DTime
               -> 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 innermost 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*.

9.3 Case Study I: SIR

Our first case study is the SIR model as introduced in Chapter 5.1. The aim of this case study is to investigate the potential speed up 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 5.4 where the discrete 2D grid is shared amongst all agents as read-only data and the agents are executed sequentially within the main thread without any concurrency.

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 grid

[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 9.1: 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 9.2: Experiments on 51x51 (2,601 agents) grid with varying number of cores.

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 grid 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 9.1) 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.

9.3.1 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 9.2.

The *STM* implementation running on 4 cores shows a speed up 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.

Comparing the performance and scaling to multiple cores shows that the *STM* implementation significantly outperforms the *Lock-Based* one and scales

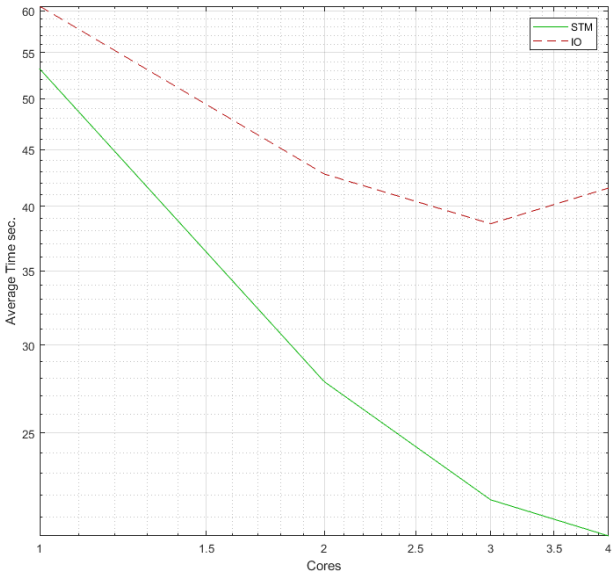


Figure 9.2: Comparison of performance and scaling on multiple cores of STM and Lock-Based. Note that the Lock-Based implementation seems to perform slightly worse on 4 than on 3 cores probably due to lock-contention.

better to multiple cores. The *Lock-Based* implementation performs best with 3 cores and shows slightly worse performance on 4 cores as can be s een in Figure 9.2. 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.

9.3.2 Varying Grid Size, Constant Cores

In this experiment we varied the grid size and used always 4 cores. The results are reported in Table 9.3 and plotted in Figure 9.3.

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

9.3.3 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 *stm-stats* library to record statistics of commits, retries and the ratio. The results are reported in Table 9.4.

Grid-Size	STM	Lock-Based	Ratio
51 x 51 (2,601)	**20.2**	41.9	2.1
101 x 101 (1,0201)	**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 9.3: Performance on varying grid sizes.

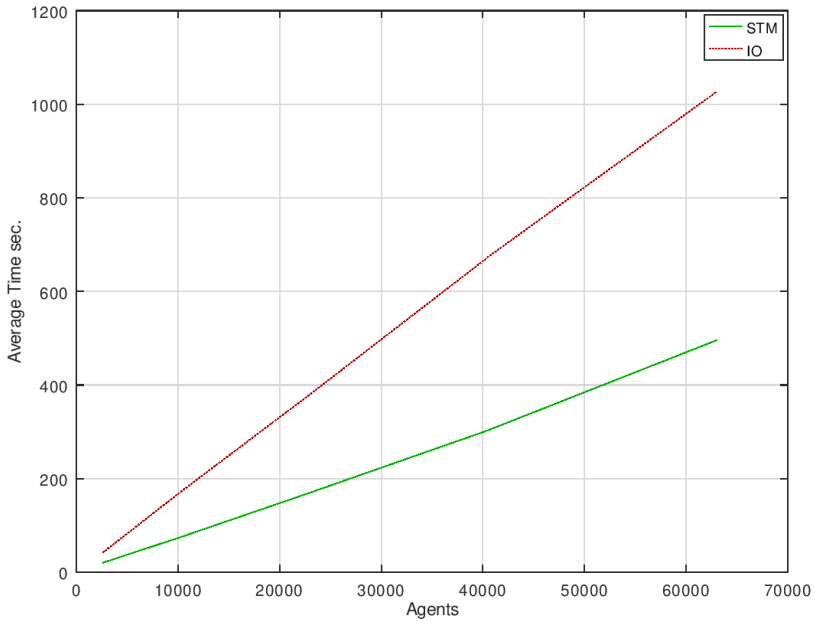


Figure 9.3: Performance on varying grid sizes.

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 9.4: 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 9.5: Performance on varying cores on Amazon S2 Services.

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.

9.3.4 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 9.5.

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. 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 performance 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 [101] because the workload of an STM action in our SIR implementation is quite small.

Compared to the *Sequential* implementation, *STM* reaches a speed up 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.

9.3.5 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 speed

up 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 is something we can not guarantee in the case of the *Lock-Based* approach as all bets are off when running within *IO*. It seems that we are quite lucky to have *both* the better performance *and* the stronger static guarantees in the *STM* approach.

9.4 Case Study II: Sugarscape

The second case study is the Sugarscape model as introduced in Chapter 6.1. 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 [38]. In each step agents search (move) to the cell with the most 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 Chapter IV 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 1.0 unit per step until the maximum capacity of a cell is reached;
- Agent Number: 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 6.2 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 [78] 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 9.1) under no additional work-load and report the average. In the experiments we varied the number of cores when running concurrently - the numbers are always indicated clearly.

Ordering The model specification requires to shuffle agents before every step (Footnote 12 on page 26 [38]). 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 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. This is not the main interest of this section though and we leave it for further research as it is completely beyond the focus of this thesis.

[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 9.6: Steps per second and retries on 50x50 grid with 500 initial agents on varying cores.

9.4.1 Constant Agent Size

In a first approach we compare the performance of all implementations on varying numbers of cores. The results are reported in Table 9.6 and plotted in Figure 9.4.

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 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 at 4 cores as well.

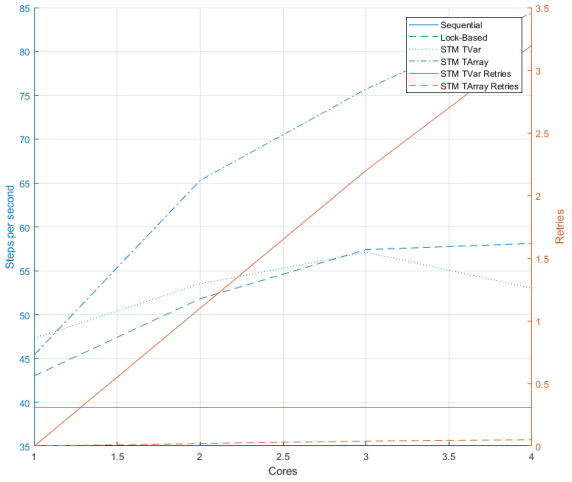


Figure 9.4: Steps per second and retries on 50x50 grid and 500 initial agents on varying cores.

9.4.2 Scaling up Agents

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 [38] 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 [38]. This ensures that we keep the number of agents roughly constant (still fluctuates but doesn't drop to 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 9.7 and plotted in Figure 9.5.

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.

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

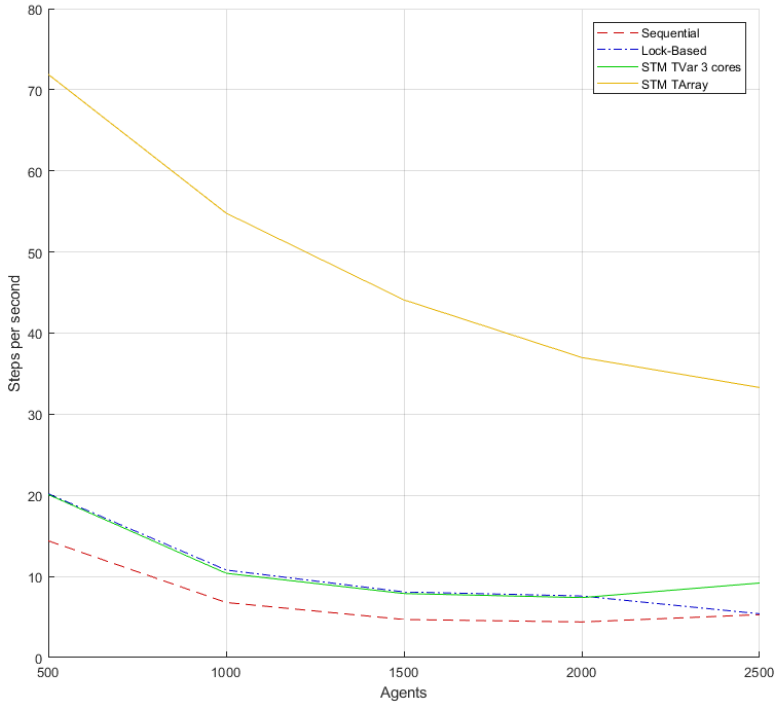


Figure 9.5: Steps per second on 50x50 grid and varying number of agents with 4 (and 3) cores except Sequential (1 core).

	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 9.8: Steps per second on varying cores on Amazon S2 Services.

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 won't rank the cells in order of their pay-off (max sugar) to move to but just stays where it is. We hypothesize 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 (see Chapter 16 for a short discussion on drawbacks) 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 chapter.

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

9.4.3 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 9.8.

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.

9.4.4 Comparison with other approaches

The paper [78] 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 [78], 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 speed up 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 is easily possible 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 which we haven't included here due to lack of space, 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 paper.

9.4.5 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 very 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 hypothesize that *STM* will still outperform the *IO* approach but to a lesser degree - we leave the proof of this for further research.

9.5 Discussion

In this chapter we have shown how to apply concurrency to monadic ABS to gain substantially speed up. We developed a novel approach, using Software Transactional Memory (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. This gives strong evidence that *STM* should be favoured over lock-based approaches in general for implementing concurrent ABS in Haskell.

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 [82] (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, for the same reason that we cannot implement TMVar with fairness: 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 hypothesize that for most models, where the *STM* approach is applicable, this has no qualitative influence on the dynamics as agents are assumed to act conceptually 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 completely beyond the scope of this thesis.

Part IV:

Property-Based testing

TODO REFINE - don't use resize population and always use normal size to be consistent and feed random population as random parameter - use Property and TimeRange consistently everywhere - WORDING: don't use replications in terms of QuickCheck! we test properties with random test-cases - IMPROVE EXPLANATION: more explanations, its too flat, code straight in the face, prepare it gently as it is not directly clear what the intention is. discuss it more generally, otherwise it reads like an advanced technical report and not like a thesis. - when using cover we cannot use random model parameters as it changes distributions - only use cover with/without checkCoverage instead of maxFailPercent =¿ rework sugarscape hypotheses chapter; need not much change only remove maxFailPercent and use cover and run all cases again.

Introduction

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 [107]. 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 [8].

One can argue that ABS should require more rigorous programming standards than other computer simulations [103]. 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 [36, 37], there exists some uncertainty about the dynamics the simulation will produce before running it. The authors [94] 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 [44].

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, software engineers have developed the concept of test-driven development (TDD).

Test-Driven Development (TDD) was popularised in the early 00s by Kent Beck [11] 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 re-

peated 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 code-coverage in particular, guarantee the correctness of an implementation 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 [27] 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 [89] 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 [93] 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 [51] 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 [34], unit testing in Haskell is quite common and robust. Although generally speaking unit tests tend 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 [34]. Although it would be interesting to see how we

can apply unit testing to our approach, it is straight forward, 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 [23, 24, 108], 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 [66]. 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 10. 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 Chapters 11 and 12. We also show briefly how to use property-tests for hypothesis testing in the context of the exploratory Sugarscape model for which no ground-truth exists in Chapter 13. We conclude with a deeper discussion on the connection between an equilibrium of a model and the totality of its simulation implementation in Chapter 14. 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.

Note that property-based testing has a close connection to model-checking [84], 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 [93] 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 prob-

lem by *automating* the test-data generation. Note that this is closely related to data-generators [51], load generators and random testing [21]. 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 [23, 24] where the authors present the QuickCheck library in Haskell, which tries to falsify the specifications by *randomly* sampling the test space. We argue, that the stochastic sampling nature of this approach is particularly well suited to ABS, because it is itself almost always driven by stochastic events and randomness in the agents behaviour, thus this correlation should make it straightforward to map ABS to property-testing. According to the authors of QuickCheck *"The major limitation is that there is no measurement of test coverage."* [23]. 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 [108], 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* [108]. 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 tests 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 on 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
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.

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. Lets 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] → 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)] → Gen a* function, where a frequency can be specified for each element. For example to generate 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 straight forward, 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
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
  = runTimeSIRFor 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 ⇒ Double → 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* which indicates 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 quite nice 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 [133], 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 ran 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. 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 later chapters.

Chapter 10

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 tests 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 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 5.1.

10.1 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 6.3.

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 3 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.

10.1.1 Deriving the specification

TODO: MakeContact has changed

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 a random number of *Contact ai Susceptible* events, where ai is the agents self id and the random number follows an exponential distribution with $\lambda = \beta$ (contact rate). The events have to be scheduled immediately without delay, thus having the current time as scheduling time-stamp. 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 won't 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 won't 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 the sender, where *ai* is the infected agents id and the scheduling time-stamp 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. In the specification there are stated three probabilities regarding β (contact rate), γ (infectivity) and δ (illness duration). We will only check one, γ (infectivity) using the coverage features of QuickCheck and write additional property-tests for the other two. The reason for that is, that checking γ is natural with the invariant checking whereas the others need a slightly different approach and are more obviously stated in separate property-tests.

10.1.2 Encoding invariants

TODO: change to random model params: beta, gamma, delta, positive time, non-empty agent ids list all in function type TODO: remove cover and checkCoverage, that overcomplicates things and makes it really difficult for the reader to understand, move the cover bit into a separate infectivity test TODO: restructure and clean up: define the properties first in pure functions and then build the property-test around, like i did in other chapters. also use pattern matching instead of case analysis

We start by giving our property a name and use *checkCoverage* to ask QuickCheck to enforce statistical testing to ensure soundness of our coverage which we will encode within this property. Also we define default values for the model parameters β, γ and δ.

```
prop_susceptible_invariants :: Property
prop_susceptible_invariants = checkCoverage (do
  let contactRate     = 5     -- beta
      infectivity     = 0.05  -- gamma
      illnessDuration = 15.0  -- delta
```

Next we generate random attributes for the agent we want to create

```
-- need a random number generator
g <- genStdGen
-- generate non-empty list of agent ids, we have at least one agent
-- the susceptible agent itself
ais <- genNonEmptyAgentIds
-- generate positive time
(Positive t) <- arbitrary
-- the susceptible agents id is picked randomly from all empty agent ids
ai <- elements ais
```

Next we need to generate a random event out of *MakeContact*, *Recover* and *Contact Int SIRState*. For this we have written a custom *Generator*, which allows to specify frequencies and the agent ids to draw the contact ids from:

```
genEventFreq :: Int -> Int -> Int -> (Int, Int, Int) -> [AgentId] -> Gen SIREvent
genEventFreq mcf _ rcf _ []
  -- no agent ids, will not generate Contact event
  = frequency [ (mcf, return MakeContact), (rcf, return Recover)]
genEventFreq mcf cof rcf (s,i,r) ais
  = frequency [ (mcf, return MakeContact)
              , (cof, do
                  -- draw SIRState with frequency
                  ss <- frequency [ (s, return Susceptible)
                                  , (i, return Infected)
                                  , (r, return Recovered)]
                  -- draw random element from agent ids
                  ai <- elements ais
                  return (Contact ai ss))
              , (rcf, return Recover)]
```

It is important to understand that together with the γ (infectivity) parameter, the frequency of the *Contact * Infected* event determines the probability of a susceptible agent to become infected. Thus we explicitly state the frequencies so we can compute the probabilities for the coverage feature.

```
let mkEvtFreq = 1
    -- will never happen as Recover will never be sent to a Susceptible
    recEvtFreq = 0
    contEvtFreq = 5
    contSusEvtFreq = 1
    contInfEvtFreq = 3
    -- will never happen, as a Recovered agent does not send any event,
    contRecEvtFreq = 0
    sirFreq    = (contSusEvtFreq, contInfEvtFreq, contRecEvtFreq)
    sirFreqSum = contSusEvtFreq + contInfEvtFreq + contRecEvtFreq
    evtFreqSum = mkEvtFreq + recEvtFreq + contEvtFreq

-- generate a random event with given frequencies
evt <- genEventFreq mkEvtFreq contEvtFreq recEvtFreq sirFreq ais
```

Then we simply create and run the random susceptible agent and collect its output.

```
-- create susceptible agent with agen id
let a = susceptibleAgent ai contactRate infectivity illnessDuration
-- run agent with given event and configuration and collect its output state ao
-- the events es it has scheduled
let (_, _, ao, es) = runAgent g a evt t ais
```

Having generated the output of the random susceptible agent, we can now start encoding the invariants. In case the random generated event was *Recover* we encode that the agent stays Susceptible and does not schedule any events. Further we compute its coverage given the frequencies of the events.

```
case evt of
  Recover -> do
    -- compute coverage
    let cp = 100 * (recEvtFreq / evtFreqSum)
    return (cover cp True "Susceptible receives Recover"
           -- must stay Susceptible and not schedule any events
           (null es && ao == Susceptible))
```

Next is the invariant of the *MakeContact* event. This is quite complex as it has a lot of small invariants encoded. We use a helper function to iterate over all events generated by the agent in which most of the invariants are encoded.

TODO: count number of ContactRate messages, should be equal contact rate parameter

```
MakeContact -> do
  -- compute coverage
  let cp  = 100 * (mkEvtFreq / evtFreqSum)
  -- check invariants
  let ret = checkMakeContactInvariants ai ais t es
  return (cover cp True "Susceptible receives MakeContact"
         -- must stay Susceptible
         (ret && ao == Susceptible))

checkMakeContactInvariants :: AgentId -> [AgentId] -> Time -> [QueueItem SIREvent] -> Bool
checkMakeContactInvariants sender ais t es
    -- make sure there was exactly one MakeContact event
    = uncurry (&&) ret
  where
    -- start out with all OK and no MakeContact event found
    ret = foldr checkMakeContactInvariantsAux (True, False) es

    checkMakeContactInvariantsAux :: QueueItem SIREvent -> (Bool, Bool) -> (Bool, Bool)
    checkMakeContactInvariantsAux
        (QueueItem receiver (Event (Contact sender' Susceptible)) t') (b, mkb)
      = (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)        -- the Contact event is scheduled immediately
    checkMakeContactInvariantsAux
        (QueueItem receiver (Event MakeContact) t') (b, mkb)
      = (b && receiver == sender  -- the receiver of MakeContact is the Susceptible agent itself
           && t' == t + 1.0       -- the MakeContact event is scheduled 1 time-unit into the future
           && not mkb, True)      -- there can only be one MakeContact event
    checkMakeContactInvariantsAux evt (_, mkb) = error ("failure " ++ show evt) (False, mkb)
```

What is left is the *Contact* event. We have to differentiate between the *SIRState* it carries. We start by looking into *Recovered*. In this case the agent stays *Susceptible* and schedules no events.

```
Contact _ s ->
  case s of
    Recovered -> do
      -- compute coverage
      let cp = contEvtSplitProb * (contRecEvtFreq / sirFreqSum)
      return (cover cp True "Susceptible receives Contact * Recovered"
             -- stays Susceptible and schedules no events
             (null es && ao == Susceptible))
```

The behaviour in case of *Susceptible* within the *Contact* event is the same and thus not repeated here. What is left is the handling of a *Contact* event with *Infected*. In case the *susceptible* agent didn't get infected nothing happens and the agent stays *Susceptible*. On the other hand, in case of infection, the invariants of scheduled events must be checked.

```
Infected -> do
  -- compute coverage
  let contInfProb = contEvtSplitProb * (contInfEvtFreq / sirFreqSum)
  -- compute infection coverage
  let infProb = contInfProb * infectivity

  if ao /= Infected
    -- not infected, nothing happens
    then return
          cover (contInfProb - infProb) True "Susceptible receives Contact * Infected, stays Susceptible"
            -- stays Susceptible and does not schedule any events
            (null es && ao == Susceptible)
    -- infected, check invariants
    else return
          cover infProb True ("Susceptible receives Contact * Infected, becomes Infected with prob " ++ show infP
            (checkInfectedInvariants ai t es)

checkInfectedInvariants :: AgentId -> Time -> [QueueItem SIREvent] -> Bool
checkInfectedInvariants sender t
    -- pattern match on exactly one Recovery event
    [QueueItem receiver (Event Recover) t']
  = sender == receiver && t' >= t           -- receiver is sender (self) and scheduled into the future
checkInfectedInvariants _ _ _ = False       -- no other event allowed
```

The specifications and encodings of the infected and recovered agent follow same patterns and are not repeated here.

10.1.3 Encoding probabilities

TODO: move infectivity and illness duration to here

10.1.4 Running the tests

When running the tests we see QuickChecks coverage features at work. It will generate as many test-cases as necessary to ensure the soundness and robustness of the coverage properties and indeed all go through.

```
  Susceptible invariants:                OK (29.00s)
    +++ OK, passed 204800 tests:
    59.4336% Susceptible receives Contact * Infected, stays Susceptible
    20.8301% Susceptible receives Contact * Susceptible
    16.6772% Susceptible receives MakeContact
     3.0591% Susceptible receives Contact * Infected, becomes Infected with prob 3.13
```

We see that the mean contact rate lies 3% above the expected value, which is normal due to ABS stochastic nature, which makes the actual values vary.

In repeated runs of the tests we get slightly different percentages due to different random number streams and random behaviour - still our properties hold, guaranteed by QuickChecks sequential statistical hypothesis testing.

10.2 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 agents get 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*.

10.2.1 Specifications of the susceptible agent

We start with the susceptible agent. We cannot directly observe that an 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 of up to 1,000 agents, 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 tast-cases accordingly.

```
prop_susceptible_invariants :: Positive Double  -- ^ Random beta, contact rate
                            -> Positive Double  -- ^ Random gamma, infectivity
                            -> Positive Double  -- ^ Random delta, illness duration
                            -> Positive Double  -- ^ Random t, duration
                            -> Property
prop_susceptible_invariants
      (Positive t) (Positive cor) (Positive inf) (Positive ild) = property (do
    -- generate population with size of up to 1000
    as <- resize 1000 (listOf genSIRState)
    -- 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 -> Susceptible"
```

This test so far did not state anything about the probability of a susceptible agent getting infected. The probability for it is bi-modal due to the combined probabilities of the exponential distribution of the contact rate and the uniform distribution of the infectivity. Unfortunately, the bi-modality makes it not possible to compute a coverage percentage of infected in this case, as we did in the event-driven test because the bi-modal 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 *MakeContact* 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.

10.2.2 Probabilities of the infected agent

Lets look at the infected agent now. 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.

```
prop_infected_invariants :: Property
prop_infected_invariants = checkCoverage (do
   -- delta, illness duration
  let illnessDuration = 15.0
  -- compute percentage of agents which recover in less or equal
  -- illnessDuration time-units. Follows the exponential distribution
  -- thus we use the CDF to compute the probability.
  let perc = 100 * expCDF (1 / illnessDuration) illnessDuration
  -- fixed sampling rate
  let dt = 0.01

  -- generate population with size of up to 1000
  as <- resize 1000 (listOf genSIRState)
```

```
-- run a random infected agent without time-limit (0) and sampling rate
-- of 0.01 and return its infinite output stream
aos <- genInfected illnessDuration as 0 dt

-- compute the recovery time
let durMay = infectedInvariant aos dt

return (cover perc (fromJust durMay <= illnessDuration)
          ("infected agents have an illness duration of  " ++ show illnessDuration ++
          " or less, expected " ++ printf "\%.2f" perc)
          -- test passes if the agent recovered
          (isJust durMay))
```

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 from which on elements are all *Infected*, thus resulting in *Just* recovery time. This is a direct expression of the fact that the CDF of the exponential distribution reaches 100% 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 practise it has turned out to terminate within a very short time (see below).

10.2.3 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 proof that the recovered agent is really behaving as specified. We made this fact very clear at the beginning of this section: property-based testing is not a proof for the correctness but only supports one in 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 will come back to the topic of computability in pure functional ABS in a slightly different context in Chapter 14.

10.2.4 Running the tests

Due to the high dimensionality of the random sampling space, especially in the susceptible invariant property-test we run 10,000 tests. All succeed as expected and there is no surprise in their output.

```
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
  Infected agent invariants:     OK (0.43s)
    +++ OK, passed 3200 tests (62.28% infected agents have an illness
        duration of 15.0 or less, expected 63.21).
  Recovered agent invariants:    OK (0.09s)
    +++ OK, passed 10000 tests.
```

10.3 Discussion

TODO: comparison of time- vs. event-driven testing - time-driven is rather straight forward: just feed the population and increment the time, the agents are much more a black-box than in the event-driven approach, where there is much revealed about their inner workings through the events, thus we have to be much more explicit in event-driven. Note that we were not able to state a coverage for the infection of susceptible agents because the distribution is bimodal due to the combined probabliities of the exponential and uniform distriubtion. In event-driven we were able to state a coverage because we decoupled the process of making contact from infection: the makeContact events followed a uniform distribution as well, generated by oureselves, so we could state an expected coverage.

Funnily the implementation of all the specifications and property-tests is longer than the original implementation. Though thats not the point 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 which would justify a whole paper alone. Due to its inherent stateful nature with complex dependencies between valid states and agents actions we need a more sophisticated approach as outlined in [31], 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.

What is particularly powerful is that one has complete control and insight over the changed state before and after e.g. a function was called on an agent: thus it is very easy to check if the function just tested has changed the agent-state itself or the environment: the new environment is returned after running the agent and can be checked for equality of the initial one - if the environments are not the same, one simply lets the test fail. This behaviour is very hard to emulate in OOP because one can not exclude side-effect at compile time, which means that some implicit data-change might slip away unnoticed. In FP we get this for free.

TODO: with lazy evaluation 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.

Chapter 11

Testing SIR Invariants

So far, our tests 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.

11.1 Deriving the invariants

By informally reasoning about the agent-specification and by realising that they are in fact a state-machine with a on-directional flow from Susceptible to Infected to 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. For each event an output of the current number of S,I and R agents is generated together with the time when the events occurred. This event time can stay the same between steps, will eventually increase but 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 in SIR. 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 due to 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 $N = S + I + R$.

11.2 Encoding the invariants

All of those properties are easily expressed directly in code (TODO: add a note in the benefits how extremely well this is encoded in functional programming).

```
sirInvariants :: Int -> [(Time, (Int, Int, Int))] -> Bool
sirInvariants n aos = timeInc && aConst && susDec && recInc && infInv
  where
    (ts, sirs)  = unzip aos
    (ss, _, rs) = unzip3 sirs

    -- 1. time is monotonic increasing
    timeInc = mono (<=)  ts
    -- 2. number of agents N stays constant in each step
    aConst = all agentCountInv sirs
    -- 3. number of susceptible S is monotonic decreasing
    susDec = mono (>=) ss
    -- 4. number of recovered R is monotonic increasing
    recInc = mono (<=)  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)

    mono :: (Ord a, Num a) => (a -> a -> Bool) -> [a] -> Bool
    mono 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 use 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
                    -> UnitRange        -- ^ 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) (UnitRange inf) (Positive ild) (TimeRange 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)
```

Unsurprisingly all 100 tests pass. Note that we put a time-limit of 150 on the simulations to run, meaning that if a simulation does not terminate before that limit, it will be terminated at $t = 150$. 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 Chapter 14.

11.2.1 Random Event Sampling

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 time-stamp, the receiver agent id and the actual event: the time-stamp is ensured to be increasing, to hold up the monotonic time property, the receiver agent id is drawn randomly from the constant list of all agents in the simulation and the actual event is generated completely randomly. As it turns out, the implementation of the agents ensure that the SIR properties are also invariant under *random event sampling* - all tests pass.

11.3 Time-driven

TODO: there is no need for this extrem technical detail. describe conceptually what happened and make clear that we fixed it.

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. Because of this, there is no need to check this property explicitly.

When we run the property-test we get a big surprise though: after a few test-cases the property-test fails due to a violation of the invariants! After a little bit of investigation it becomes clear that the invariant *(3) number of susceptible agents is monotonic decreasing* is 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. How can this happen?

The source of the problem is the use of *dpSwitch* in the implementation of the *stepSimulation* function as shown in Chapter 5.2. The d in *dpSwitch* stands for delayed observation, which means that the output of the switch at time of switching is the output of the *old* signal functions [28]. Speaking more technically: *dpSwitch* is non-strict in its switching event, which ultimately results in old signal functions being run on new output which were but produced by those old signal functions: the output of time-step t is only visible in the time-step t=t+dt but the signal functions at time-step t are actually one step ahead. This is particularly visible at $t = 0$ and $t = \Delta t$, where the outputs are the same but the signal functions are not: the one at $t = \Delta t$ has changed already.

This has the desired effect that in the case of our SIR implementation, the population the agents see at time $t = t + \Delta t$ is the one from the previous step t, generated with the signal functions at time t. Due to the semantics of *dpSwitch*, in the next switching event those signal functions from time t are run again with the new input to produce the next output *and* the new signal functions - in each step output is produced but due to the delay of *dpSwitch* and the use of *notYet*, we get this alternating behaviour.

This leads to trouble if the very rare case happens when a susceptible agent makes the transition from susceptible to recovered within one time-step. This is indeed possible due to the semantics of *switch*, which is employed to make the state-transitions. In case a switching event occurs, *switch* runs the signal function into which was switched immediately, which makes it highly unlikely but possible, that the susceptible agent, which has just switched into infected recovers immediately by making the *switch* to recovered. Why does this violate the property then?

Lets assume that at $t = t + \Delta t$ the agents receive as input the population from time t which contains, say 42, susceptible agents. A susceptible agent then makes the highly unlikely transition to recovered, reducing the number of susceptible agents by 1 to 41. In the next step the old signal function is run again but with the new input, which is slightly different, thus leading to slightly different probabilities. A susceptible agent has become recovered, which reduces probabilities of a susceptible agent becoming infected. When now the old signal function is run, it leads to a different output due to different probabilities:

the susceptible agent stays susceptible instead of becoming infected (and then recovering).

One way to solve this problem would be to use *pSwitch*. It is the non-delayed, stricht version of *dpSwitch*, which output at time of switching is the output of the *new* signal functions. Using *pSwitch* instead of *dpSwitch* solves the problem because it will use the new signal functions in the second run because it is strict. Indeed, when using this, the property-test passes. This comes though at a high cost: due to *pSwitch* strict semantics, which runs all the signal functions before and at time of switching, all agents are run twice in each step! This is clearly an unacceptable solution, especially because the time-driven approach already suffers severe performance problems. A more performant solution is to delay the susceptible agents output, as we have done already in 5.2. This solves the problem as well and the property-test passes.

11.4 Comparing time- and event-driven

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: do 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 e.g. 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 [79] that both implementations produce a bi-modal distribution, thus we have to use a non-parametric test like Mann-Whitney to compare them. We discuss the issue of bi-modality more in Chapter 12.

The property-test requires a 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     -- ^ Random beta, contact rate
                      -> UnitRange        -- ^ Random gamma, infectivity, within (0,1) range
                      -> Positive Double  -- ^ Random delta, illness duration
                      -> TimeRange        -- ^ Random time to run within (0, 50) range
                      -> [SIRState]       -- ^ Random population
                      -> Property
prop_event_time_equal
```

```
  (Positive cor) (UnitRange inf) (Positive ild) (TimeRange 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.
-- We assume a 90% similarity
return
  (cover 90 allPass "SIR event- and time-driven 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 Chapter Chapter 12.

11.5 Discussion

TODO write a full reflection on this chapter

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 5.2.

Chapter 12

Testing the SIR model specification

In the previous chapters 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 5.1. What we are lacking is a verification whether the implementations also match the formal SD specification or not. In the process of verification, we need to make sure it is correct up to some specification. 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 5.1, which we repeat here:

$$\begin{aligned} \frac{\mathrm{d}S}{\mathrm{d}t} &= -infectionRate & infectionRate &= \frac{I\beta S\gamma}{N} \\ \frac{\mathrm{d}I}{\mathrm{d}t} &= infectionRate - recoveryRate & recoveryRate &= \frac{I}{\delta} \\ \frac{\mathrm{d}R}{\mathrm{d}t} &= recoveryRate \end{aligned} \tag{12.1}$$

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 won't change any more.

$$S(t) = N - I(0) + \int_0^t -infectionRate \, \mathrm{d}t \tag{12.2}$$

$$I(0) = 1 \tag{12.3}$$

$$I(t) = \int_0^t infectionRate - recoveryRate \, \mathrm{d}t \tag{12.4}$$

$$R(t) = \int_0^t recoveryRate \, \mathrm{d}t \tag{12.5}$$

12.1 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 from 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 the 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 encoded in 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.

12.2 Implementing the test

Implementing a property-test is straight-forward. 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
                   -> UnitRange        -- ^ 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) (UnitRange inf) (Positive ild) (TimeRange t) as = checkCoverage (do
  -- run 100 replications of time-driven SIR implementation
  (ss, is, rs) <- unzip3 <$> genTimeSIRRepls 100 as cor inf ild t
  let prop = compareSDToABS as ss is rs cor inf ild t
  return $ cover 75 prop "SIR time-driven passes t-test with simulated SD" True
```

12.3 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 11, 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 [79] in the case of the SIR model, where the authors showed that it generates a bimodal distribution. Further, the authors report that a 70% envelopes 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 [79].

The actual question in the end is whether it actually even makes sense to compare the approaches to the SD one or even amongst each other - after all they can be seen as fundamentally different approaches. We can argue that they are qualitatively equal as [41] 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 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.

12.4 Discussion

After having shown in previous chapters, that individual agent behaviour is correct up to some specification, in this chapter we 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 [79, 41] have already shown. Our approach might work out better for a different model, which has a better behaved underlying specification than the bimodal SIR.

Chapter 13

Hypotheses in Sugarscape

TODO: replace maxFailPercent with cover and checkCoverage and configure quickcheck to run only 10 tests

In this chapter we look at how property-based testing can be made of use to verify the *exploratory* Sugarscape model [38] as already introduced in Chapter 6.1. Whereas in the previous chapter on testing the explanatory SIR case-study 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 then (hopefully) come up 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 are 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 6.1. We validated it against the book [38] and a NetLogo implementation [134] [2]. A longer report on the details of this validation process is attached as Appendix B, in this section we focus on QuickChecks role in this process.

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

[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`

different random-number streams and require that the tests all pass. During the verification process described in Appendix B we have derived and implemented property-tests for the following hypotheses:

1. Disease Dynamics 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 minority recover - 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 green, 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 (chapter II), 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 and not moving any more.

13.1 Implementation

To implement this, we make use of QuickChecks *Generator* concept. A *Generator* defines how specific (random) data can be generated and is implemented using the *Gen* monad where all of QuickChecks random distribution functionality is available. Thus it is only natural that we implement a *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 -> SugarScapeScenario -> Gen [SimStepOut]
sugarscapeUntil ticks params = do
```

```
  -- draw a seed for the random-number generator from the full range of Int
  seed <- choose (minBound, maxBound)
  -- create a random-number generators
  let g = mkStdGen seed
  -- initialise the simulation state with the given random-number generator
  -- and the parameters
  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 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 encoding of the *Disease Dynamics (1)* hypothesis. Its type is *Property*, which is required by QuickChecks top-level testing function. To generate a property, the *property* function is used which takes a *Gen Bool* computation or a simple *Bool* function as predicate to indicate success (True) or failure (False). In this case we use a *Gen Bool* to be able to run the sugarscape data generator.

```
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 (infected == 0))
```

QuickCheck runs multiple replications of a property when testing it, where the number is 100 by default. 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. Here is how we encode this into a property-test:

```
prop_gini :: Int -> Double -> 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 repls replciations ...
```

```
  gini <- vectorOf repls (genGiniCoeff ticks params)

  -- on a two-tailed t-test with given confidence
  let giniTTest = tTestSamples TwoTail expGini (1 - confidence) gini

  return giniTTest)

genGiniCoeff :: Int -> SugarScapeScenario -> Gen Double
genGiniCoeff ticks params = do
  -- generate sugarscape data
  aos <- sugarscapeUntil ticks params
  -- extract wealth of the agents in the last step
  let agentWealths = map (sugObsSugLvl . snd) (last aos)
  -- compute gini coefficient and return it
  return (giniCoeff agentWealths)
```

13.2 Running the tests

As already pointed out, QuickCheck tries to run by default up to 100 replications of a property and if all evaluate to *True* the property-test succeeds. On the other hand, QuickCheck will stop at the first predicate which evalutes to *False* and marks the whole property-test as failed, no matter how many replications got through already.

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. Unfortunately, when running the tests only the Disease Dynamics (1) and (2) go through, all the other tests fail. It is important to understand that QuickCheck is always initialised with a new random-number seed when run, thus we might have just been unlucky. Unfortunately, when we run it again, this happens again, thus there seems to be something wrong with our approach.

13.2.1 Allowing failure

It is arguably the case that the binary approach of QuickCheck, where the whole property-test fails when a single replication fails, is too strict for testing ABS in general and our hypotheses in particular. The reason for that is, that due to ABS stochastic nature, the hypotheses might hold for a large number of replications but not strictly for all.

As a remedy, we can use *maxFailPercent* [3] as a configuration argument to QuickCheck, which allows the failure of a given percentage of replications. The argument behaves in a way that it tries to run up to the 100 default (or whatever

[3] As of the time of writing this thesis (2nd April 2019), this only exists as a pull request `https://github.com/nick8325/quickcheck/pull/239` and has not been merged into the main branch of QuickCheck. Thus we use the QuickCheck from `https://github.com/stevana/quickcheck/tree/feat/max-failed-percent` who has provided the implementation of *maxFailPercent*.

the configuration is) successful replications but fails the overall property-test if the percentage of failed replications is reached. By switching from a binary PASS/FAIL to a more probabilistic measure, reflecting reliability, we have now a more appropriate tool for testing the suitability of our hypotheses.

We run the tests again with 10 replications each but now allowing 100% of failure in each case to see how reliable each hypothesis is. In one specific run we get the following result:

1. Disease Dynamics all recover: *+++ OK, passed 10 tests.*
2. Disease Dynamics minority recover: *+++ OK, passed 10 tests.*
3. Trading Dynamics: *+++ OK, passed 10 tests; 2 failed (16%).* (In total 12 tests (replications) were run, out of which 2 failed, which is a 16% failure rate.)
4. Cultural Dynamics: *+++ OK, passed 10 tests; 3 failed (23%).*
5. Inheritance Gini Coefficient: **** Failed! Passed only 0 tests; 10 failed (100%) tests.*
6. Carrying Capacity: *+++ OK, passed 10 tests; 2 failed (16%).*
7. Terracing: *+++ OK, passed 10 tests; 2 failed (16%).*

How to deal with the failure of the 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 test.

Note that we disabled shrinking for hypothesis testing as it has no meaning here. The only thing which would be shrunk would be the seed of the random-number generator, which has no intrinsic meaning.

13.3 Discussion

In this chapter 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 allow failure using the *maxFailPercent* argument 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.

Chapter 14

The Equilibrium-Totality Correspondence

TODO UNFINISHED - write NON-DEPENDENTLY TYPED ARGUMENT - NEEDS SUBSTANTIAL RESEARCH: IMPLEMENT A TOTAL SIR IMPLEMENTATION IN IDRIS - write a short intro into dependent types - the question will remain: does this chapter really belong in this thesis?

In the property-tests of Chapter 11 and 10.2 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 practical to guarantee that we will actually terminate 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 which then make the transition to recovered after having gone through the infection phase. The infected agents themselves *will* recover within finite time. Thus we can conclude that a correct implementation of the SIR model must enter a steady state in finite time.

Using this informal reasoning, we change the property-test from Chapter 11 to encode this property implicitly.

```
prop_sir_invariants :: Positive Int     -- ^ beta, contact rate
                    -> UnitRange        -- ^ gamma, infectivity in range (0,1)
                    -> Positive Double  -- ^ delta, illness duration
                    -> [SIRState]       -- ^ population
                    -> Property
prop_sir_invariants
```

[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.

```
  (Positive cor) (UnitRange 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
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 which would violate 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 non-computable. Obviously this is nothing new and was established in the 1930s through the work of Turing [125]. The question is now: what can we do about it?

The 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 is defined in [20] as one that terminates with a well-typed result or produces a non-empty finite prefix of a well-typed infinite result in finite time. 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 non-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 (see 14.2) which must terminate, this means a well-typed program (which is itself a proof) is always terminating which in turn means that it must consist out of total functions.

There exist a number of excellent introduction to dependent types which we use as main ressources for this section: [123, 105, 115, 20, 102]. We are using Idris [19] as the language of choice as it is very close to Haskell with focus on real-world application and running programs as opposed to other languages with dependent types e.g. Agda and Coq which serve primarily as proof assistants.

Dependent types are a very powerful addition to functional programming as they allow us to express even stronger guarantees about the correctness of

programs *already at compile-time*. They go as far as allowing to formulate programs and types as constructive proofs which must be *total* by definition [123, **?**, 4].

So far no research using dependent types in agent-based simulation exists at all. We have already started to explore this for the first time and ask more specifically how we can add dependent types to our functional approach, which conceptual implications this has for ABS and what we gain from doing so. We are using Idris [19] as the language of choice as it is very close to Haskell with focus on real-world application and running programs as opposed to other languages with dependent types e.g. Agda and Coq which serve primarily as proof assistants.

We hypothesise, that dependent types will allow us to push the correctness of agent-based simulations to a new, unprecedented level by narrowing the gap between model specification and implementation. The investigation of dependent types in ABS will be the main unique contribution to knowledge of my Ph.D.

There is 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. This is the subject of the next part of the thesis which tries to move towards dependent types in ABS.

14.1 A total SIR implementation

In this section we want to implement a total agent-based SIR simulation, where the termination does NOT depend on time (is not terminated after a finite number of time-steps, which would be trivial).

Dependent Types and Idris' ability for totality- and termination-checking should theoretically allow us to proof 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. Note that 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 non-total as reasoned above.

The HOTT book [105] 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?

By this reasoning, a non-total, correctly implemented agent-based simulations of the SIR model will eventually terminate (note that this is independent of which environment is used and which parameters are selected). Still this does not formally proof that the agent-based approach itself will terminate and so far no formal proof of the totality of it was given.

Thus an agent-based implementation of the SIR simulation has to terminate if it is implemented correctly because all infected agents will recover after a finite number of steps after then the dynamics will be in equilibrium. Thus we have the following conditions for totality:

1. The simulation shall be terminated when there are no more infected agents.
2. All infected agents will recover after a finite number of time, which means that the simulation will eventually run out of infected agents.

   Unfortunately this criterion alone does not suffice because when we look at the SIR+S model, which adds a cycle from Recovered back to Susceptible, we have the same termination criterion, but we cannot guarantee that it will run out of infected. We need an additional criteria.
3. The source of infected agents is the pool of susceptible agents which is monotonic decreasing (not strictly though!) because recovered agents do NOT turn back into susceptibles.
4. finite number of initial susceptibles
5. finite number of initial infected
6. finite illness duration
7. finite contact rate

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?

14.2 Constructivism in ABS

The main theoretical and philosophical underpinnings of dependent types as in Idris are the works of 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 revolve 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 [3]. 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. [123] (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 the general case.

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 computation, they are not computable. This leads to a fundamental tenet in constructive mathematics: [123] (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 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.

All this is directly reflected in dependently typed programs as we introduced

[2] We follow the excellent introduction on constructive mathematics [123], chapter 3.

[3] Polynomial of degree n has n complex roots; continuous functions which change sign over a compact real interval have a zero in that interval,...

above: functions need to be total (finitary) and produce proofs like in *checkEqNat* which allows the compiler to extract additional relevant computational information. Also the way we described the (infinite) natural numbers was in an finitary way. In the case of decidable equality, the case where it is not equal, we need to provide an actual proof of contradiction, with the type of Void which is Idris representation of $\bot$.

14.2.1 Verification, Validation and Dependent Types

Dependent types allow to encode specifications on an unprecedented level, narrowing the gap between specification and implementation - ideally the code becomes the specification, making it correct-by-construction. The question is ultimately how far we can formulate model specifications in types - how far we can close the gap in the domain of ABS. Unless we cannot close that gap completely, to arrive at a sufficiently confidence in correctness, we still need to test all properties at run-time which we cannot encode at compile-time in types.

Nonetheless, dependent types should allow to substantially reduce the amount of testing which is of immense benefit when testing is costly. Especially in simulations, testing and validating a simulation can often take many hours - thus guaranteeing properties and correctness already at compile time can reduce that bottleneck substantially by reducing the number of test-runs to make.

Ultimately this leads to a very different development process than in the established object-oriented approaches, which follow a test-driven process. There one defines the necessary interface of an object with empty implementations for a given use-case first, then writes tests which cover all possible cases for the given use-case. Obviously all tests should fail because the functionality behind it was not implemented yet. Then one starts to implement the functionality behind it step-by-step until no test-case fails. This means that one runs all tests repeatedly to both check if the test-case one is working on is not failing anymore and to make sure that old test-cases are not broken by new code. The resulting software is then trusted to be correct because no counter examples through test hypotheses, could be found. The problem is: we could forget / not think of cases, which is the easier the more complex the software becomes (and simulations are quite complex beasts). Thus in the end this is a deductive approach.

With pure functional programming and dependent types the process is now mostly constructive, type-driven (see [20]). In that approach one defines types first and is then guided by these types and the compiler in an interactive fashion towards a correct implementation, ensured at compile-time. As already noted, the ABS methodology is constructive in nature but the established object-oriented test-driven implementation approach not as much, creating an impedance mismatch. We expect that a type-driven approach using dependent types reduces that mismatch by a substantial amount. Models like the Sugarscape 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 be-

fore running the model and then compare them informally against the resulting dynamics.

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.

Note that *validation* is a different matter here: independent of our implementation approach we still need to validate the simulation against the real-world / ground-truth. This obviously requires to run the full simulation which could take up hours in either programming paradigm, making them absolutely equal in this respect. Also the comparison of the output to the real-world / ground-truth is completely independent to the paradigm. The fundamental difference happens in case of changes made to the code during validation: in case of the established test-driven object-oriented approach for every minor change one (should) re-run all tests, which could take up a substantial amount of additional time. Using a constructive, type-driven approach this is dramatically reduced and can often be completely omitted because the correctness of the change can be either guaranteed in the type or by informally reasoning about the code.

ABS as a constructive / generative science, follows Poperian approach of falsification: we try 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 say that we have found *a* hypothesis (the model) which emergent properties explains the real-world phenomenon sufficiently enough. This is not a proof but only one possible explanation which holds for now and might be falsified in the future.

When we implement our simulation things change a bit as we add another layer: the conceptual model, describing the phenomenon, which is an abstraction of reality. This description can be of many forms but can be regarded on a line between completely formal (economic models) to informal (sociology) but the implementation will follow that description. The fundamental difference here is that in this case we want our implementation to be exactly the same as the conceptual model. Contrary to the real-world, where it is not possible to find a *true* model (as was argued by Popper), on this level we actually can construct an implementation which matches the conceptual model exactly because we have a description of the conceptual model. In the end we transform the conceptual model description in code, which is itself a formal description. In this translation process (speak: implementation / programming), one can make an endless number of mistakes. Generally we can distinguish between two classes of mistakes: 1) conceptual mistakes - wrong translation of the model specifications into code due to various reasons e.g. imprecise description, human error. The more precise an unambiguous a model description is, the less probable conceptual mistakes will be. 2) internal mistakes - normal programming mistakes e.g. access of arrays out of bounds, ... also using correlated Random Number generators. Level 0: Real-World phenomenon Level 1: Conceptual model of the real-world phenomenon Level 2: Implementation of the conceptual model Note

that we must speak of falsification and constructiveness on two different levels: - validation level: do the results of the conceptual model match the real-world phenomenon? the conceptual model is the hypothesis which says that its mechanics are sufficient to generate / construct the real-world phenomenon. At this level we are not interested in the implementation level anymore - the implemented model *is* (seen as) the conceptual model, and one only compares its output to the real-world. If the dynamics match, then we got a valid hypothesis which works for now. If the dynamics do NOT match, then the hypothesis (the model) is falsified and one needs to adjust / change the hypothesis (model). The validation will happen by tests, there is no other way, we have no formal specification of the real-world, we can only observe empirically the phenomena, so we run tests which try to falsify the outputs of the model: assuming it will generate phenomena of the real-world and test if it does. - implementation & verficiation level: in this step we are matching the code to the conceptual model. Here we are not only restricted to a test-driven approach because we have a more or less formal description of the conceptual model which we directly encode in our programming language. If the language allows to express model specifications already at compile-time then this means that the implementation narrows the gap between model specification and implementation which means it does not need to be tested at run-time because it is guaranteed for all inputs for all time.

The constructiveness of ABS and impendance mismatch: ABS methodology is constructive but the established implementation approach not too much, creating an impedance mismatch. this is especially visible in the test-driven development dependent types constructive nature could close this mismatch.

14.3 Discussion

In this chapter we have shown how to encode equilibria properties in the types in a way that the simulation automatically terminates when they are reached. This results then in a *total* simulation, creating a *correspondence between the equilibrium of a simulation and the totality of its implementation.* Of course this is only possible for models in which we know about their equilibria a priori or in which we can reason somehow that an equilibrium exists.

Models like the Sugarscape 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.

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, nothing prevents one from making the transition back to susceptible.

Using dependent types it might be possible to encode invariants and state-machines on the type level which can prevent such invalid transitions already at compile-time. This would be a huge benefit for ABS because of the popularity of state-machines in agent-based models.

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 can dependent types be used to express the flow of time and that it is strongly monotonic increasing?

In more sophisticated models agents interact in more complex ways with each other e.g. through message exchange using agent IDs to identify target agents. The existence of an agent is not guaranteed and depends on the simulation time because agents can be created or terminated at any point during simulation.

Dependent types could be used to implement agent IDs as a proof that an agent with the given id exists *at the current time-step*. This also implies that such a proof cannot be used in the future, which is prevented by the type system as it is not safe to assume that the agent will still exist in the next step.

In case of an SD this will take forever to reach 0 due to the dynamics of the equations and floating point arithmetic is another difficulty. On the other hand, due to ABS discrete nature this is not an issue anymore: agents are discrete and as soon as we hit 0 infected agents - which, due to Integer representation, can be exact - an equilibrium is reached.

Note that there exists a SIR+S model, which adds a cycle back from Recovered to Susceptible - if we add this cycle in our total implementation, this should make it immediately non-total as an important criteria for totality gets violated: the source of susceptible is not finite anymore and we might run in non-stationary cycles like in a prey-predator model with Lotka-Volterra equations.

TODO: connect to property-based testing and put emphasise on the constructive nature and hypothesis testing: this is a popperian approach.

FP is the first step towards a more structural understanding of ABS implementations where dependent types should allow us to develop this even further. we leave this for further research and outline only broadly the ideas we want to follow.

Linear and Dependent Types with Idris 2: more general ideas / hints / research on how it is applicable to ABS

By definition, ABS is of constructive nature, as described by Epstein [36]: "If you can't grow it, you can't explain it" - thus an agent-based model and the simulated dynamics of it is itself a constructive proof which explain a real-world phenomenon sufficiently well. Although Epstein certainly wasn't 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 agent-based simulations, 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 agent-based simulations. Probably the answer might be that a dependently typed agent-based simulation 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.

Discussion

TODO: wrap-up, comparison of time- and event-driven implementations.

we can see quickcheck not only as testing but also as verification and validation allowing to test hypotheses and properties during the development process, also has connections to Monte-Carlo simulation

further research: inclusion of Environment: all properties should still hold under different Environments which can be generated randomly as Well e.g. random Networks.

further research: does rng correlation as discussed in time-driven chapter implementation show up with property-tests?

prop test with cover but without checkcoverage helps quick overview

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.

Although property-based testing has its origins in Haskell, similar libraries have been developed for other languages e.g. Java, Pyhton, C++ as well and we hope that our research has sparked an interest in applying property-based testing to the established object-oriented languages in ABS as well.

Part V:

Discussion

Discussion

This part re-visits the aim, objective and hypotheses of the introduction and puts them into perspective with the contributions. Also additional ideas, worth mentioning here (see below) will be discussed here.

Chapter 15

Benefits

TODO

Probably the biggest strength 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) - we can guarantee that in our pure functional approach already *at compile time*.

Refactoring is very convenient and quickly becomes the norm: guided by types (change / refine them) 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 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.

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 (STM) has been shown [122] which underlines the strength of Haskell for concurrent ABS due to its strong guarantees about retry-semantics.

Testing in general allows much more control and checking of invariants due to the explicit handling of effects - together with the strong static type system, the testing-code is in full, explicit control over the functionality it tests. Property-based testing in particular is a perfect match to testing ABS due to the random nature in both and because it supports convenient expressing of specifications. Thus we can conclude that in a pure functional setting, testing is very expressive and powerful and supports working towards an implementation which is very likely to be correct.

15.0.1 Verification and Correctness

General there are the following basic verification & validation requirements to ABS [107], which all can be addressed in our *pure* functional approach as described in the paper in Appendix **??**:

- Fixing random number streams to allow simulations to be repeated under same conditions - ensured by *pure* functional programming and Random Monads
- Rely only on past - guaranteed with *Arrowized* FRP
- Bugs due to implicitly mutable state - reduced using pure functional programming
- Ruling out external sources of non-determinism / randomness - ensured by *pure* functional programming
- Deterministic time-delta - ensured by *pure* functional programming
- Repeated runs lead to same dynamics - ensured by *pure* functional programming

1. Run-Time robustness by compile-time guarantees - by expressing stronger guarantees already at compile-time we can restrict the classes of bugs which occur at run-time by a substantial amount due to Haskell's strong and static type system. This implies the lack of dynamic types and dynamic casts [1] which removes a substantial source of bugs. Note that we can still have run-time bugs in Haskell when our functions are partial.
2. Purity - By being explicit and polymorphic in the types about side-effects and the ability to handle side-effects explicitly in a controlled way allows to rule out non-deterministic side-effects which guarantees reproducibility due to guaranteed same initial conditions and deterministic computation. Also by being explicit about side-effects e.g. Random-Numbers and State makes it easier to verify and test.
3. Explicit Data-Flow and Immutable Data - All data must be explicitly passed to functions thus we can rule out implicit data-dependencies because we are excluding IO. This makes reasoning of data-dependencies and data-flow much easier as compared to traditional object-oriented approaches which utilize pointers or references.
4. Declarative - describing *what* a system is, instead of *how* (imperative) it works. In this way it should be easier to reason about a system and its (expected) behaviour because it is more natural to reason about the behaviour of a system instead of thinking of abstract operational details.

[1] Note that there exist casts between different numerical types but they are all safe and can never lead to errors at run-time.

5. Concurrency and parallelism - due to its pure and 'stateless' nature, functional programming is extremely well suited for massively large-scale applications as it allows adding parallelism without any side-effects and provides very powerful and convenient facilities for concurrent programming. The paper of (TODO: cite my own paper on STM) explores the use Haskell for concurrent and parallel ABS in a deeper way.

TODO: haskell-titan TODO: Testing and Debugging Functional Reactive Programming [100]

Static type system eliminates a large number run-time bugs.

TODO: can we apply equational reasoning? Can we (informally) reason about various properties e.g. termination?

Follow unit testing of the whole simulation as prototyped for towards paper. in this we explore something new: property-based testing in ABS

Chapter 16

Drawbacks

16.1 Space-Leaks

discuss the problem (and potential) of lazy evaluation for ABS: can under some circumstances really increase performance when some stuff is not evaluated (see STM study) but mostly it causes problems by piling up unevaluated thunks leading to crazy memory usage which is a crucial problem in simulation. Using strict pragmas, annotations and data-structures solves the problem but is not trivial and involves carefully studying the code / getting it right from the beginning / and using the haskell profiling tools (which are fucking great at least). TODO: show the stats of memory usage

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: because simulations might run for a (very) long time or conceptually forever, one must make absolutely sure that the memory usage stays somewhat constant. As a remedy, Haskell allows to add so-called strictness pragmas to code-modules which forces strict evaluation of all data even if it is not used. Carefully adding this conservatively file-by file applying other techniques of forcing evaluation removed most of the memory leaks.

16.2 Efficiency

ordering of the transformers when to run a transformer, lazyness vs. strictness

The main drawback of our approach is performance, which at the moment does not come close to OO implementations. There are two main reasons for it: first, FP is known for being slower due to higher level of abstractions, 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 we have done in our paper on STM [122]. The second reason can be addressed by the use of linear types [12], which allow to annotate a variable with how often it is used within a function. From this a compiler can derive aggressive optimisations, resulting in imperative-style performance but retaining the declarative nature of the code.

16.3 Productivity and learning curve

A case study in [53] hints that simply by switching to a static typesystem alone does not gain anything and can even be detriment. It also needs to have a certain level of abstractions like Haskell type system does or even dependent types as in Idris. This also mean that there is a substantial learning curve to master when one wants to enter pure functional ABS.

Chapter 17

The Gintis Case

Discuss my developed techniques to the Gintis paper (and its follow ups: the Ionescu paper [17] and a Masterthesis [39] on it). Answer the following:

1. Do the techniques transfer to this problem and model?
2. Could pure functional programming have prevented the bugs which Gintis made?
3. Would property-based tests have been of any help to preven the bugs?
4. Could dependent and / or types have prevented the bugs which Gintis made?
5. How close is our (dependently typed) implementation to Ionescus functional specification?
6. When having Cezar Ionescu as external examiner, this chapter will be of great influence as it deals heavily with his work.

Not yet started, need to implement it but there exists code for it already (gintis and java implementations)

Chapter 18

Generalising Research

We hypothesize that our research can be transferred to other related fields as well, which puts our contributions into a much broader perspective, giving it more impact than restricting it just to the very narrow field of Agent-Based Simulation. Although we don't have the time to back up our claims with in-depth research, we argue that our findings might be applicable to the following fields at least on a conceptual level.

18.1 Simulation in general

We already showed in the paper [**?**], that purity in a simulation leads to repeatability 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 my approach to dependent types in ABS might be applicable to simulations in general due to the correspondence between equilibrium & totality, in use for hypotheses formulation and specifications formulation as pointed out in Chapter **??**.

18.2 System Dynamics

discuss pure functional system dynamics - correct by construction: benefits: strictly deterministic already at compile time, encode equations directly in code =¿ correct by construction. Can serve as backend implementation of visual SD packages.

18.3 Discrete Event Simulation

pure functional DES easily possible with my developed synchronous messaging ABS DES in FP: we doing part of it in event-driven SIR. this is a potential hint at how to achieve DES in pure FP: arrowized FRP seems to be perfect

for it because it allows to express data-flow networks, which is exactly what DES is as well. The problem with pure FP is that it will be more involved to propagate events through the network especially when they don't originate from the source but e.g. after a time-out from a queue. Generally all the techniques are there as discussed in the event-driven chapter but it would be interesting to do research on how to achieve the same in DES. Because data-flow networks of DES generally don't change at runtime, they are fixed already at compile time - it would be interesting to see what dependent types could offer for additional compile-time guarantees.

PDES, should be conceptually easil possible using STM, optimistic approach should be conceptually easier to implement due to persistent data-structures and controlled side-effects

18.4 Recursive Simulation

Due to the recursive nature of FP we believe that it is also a natural fit to implement recursive simulations as the one discussed in [45]. In recursive ABS agents are able to halt time and 'play through' an arbitrary number of actions, compare their outcome and then to resume time and continue with a specifically chosen action e.g. the best performing or the one in which they haven't died. 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 FP, we think that implementing a recursive simulation in such a setting should be straight-forward.

Inspired by [45], add ideas about recursive simulation described in 1st year report and "paper". functional programming maps naturally here due to its inherently recursive nature and controlled side-effects which makes it easier to construct correct recursive simulations. recursive simulation should be conceptually easier to implememt and more likely to be correct due to recursive Nature of haskell itself, lack of sideeffeccts and mutable data

18.5 Multi Agent Systems

The fields of Multi Agent Systems (MAS) and ABS are closely related where ABS has drawn much inspiration from MAS [139], [135]. It is important to understand that MAS and ABS are two different fields where in MAS the focus is more on technical details, implementing a system of interacting intelligent agents within a highly complex environment with the focus on solving AI problems.

Because in both fields, the concept of interacting agents is of fundamental importance, we expect our research also to be applicable in parts to the field of MAS. Especially the work on dependent types 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 specifications.

Chapter 19

Applicability of Object-Oriented modelling Frameworks

TODO: discusses if and how peers object-oriented agent-based modelling framework can be applied to our pure functional approach. TODO: i need to re-read peers framework specifications / paper from the simulation bible book. Although peers framework uses UML and OO techniques to create an agent-based model, we realised from a short case-study with him that most of the framework can be directly applied to our pure functional approach as well, which is not a huge surprise, after all the framework is more a modelling guide than an implementation one. E.g. a class diagram identifies the main datastructures, their operations and relations, which can be expressed equally in our approach - though not that directly as in an oo language but at least the class diagram gives already a good outline and understanding of the required datafields and operations of the respective entities (e.g. agents, environment, actors,...). A state diagram expresses internal states of e.g. an agent, which we discussed how to do in both our time- and even-driven approach. A sequence diagram e.g. expresses the (synchronous) interactions between agents or with their environment, something for which we developed techniques in our event-driven approach and we discuss in depth there.

Chapter 20

Alternatives

Shortly discuss alternative implementation directions which we didn't / couldn't follow because of not enough time / not enough experience / developed experience too late.

20.1 Haskell

Freer Monads: `https://reasonablypolymorphic.com/blog/freer-monads/`. They aim to separate Definition from implementation by writing a domain-specific language using GADTs which are then interpreted. This allows to strictly separate implementation from specification, composes very well and thus is easier to test as parts can be easily mocked. Also Freer Monads free one from the order of effects imposed through Monad Transformers. In general Freer Monads seem to aim for the same abstraction what modern interface-based oop does. Problem: Yes, freer monads are today somewhere around 30x slower than the equivalent mtl code. because its $O(n^2)$. ABS are not IO bound, so raw computation is all what counts and this is undoubtly worse with Freer monads. Given that we are already having problematic performance, we can't sacrifice even more. There seem to be a better encoding possible, which is about 2x slower than MTL: `https://reasonablypolymorphic.com/blog/too-fast-too-free/`. Still it might prove to be useful in other terms like proving correctness and then translating it, but how could we do it? ContT is Not an Algebraic Effect so it seems to be difficult to implement continuations in freer monads. Unfortunately this is what we really need as shown in Event driven ABS and generalising structure chapters. Other criticism of Freer Monads are: `https://medium.com/barely-functional/freer-doesnt-come-for-free-c9fade793501` are: boilerplate code which though can be generated automatically by some libraries, performance when not IO based because the program is bascially a data-structure which is interpreted, concurrency seems to be tricky, TODO read: `https://reasonablypolymorphic.com/blog/freer-yet-too-costly/`

TODO: it seems that Final Tagless is another alternative. Look into it:

https://jproyo.github.io/posts/2019-03-17-tagless-final-haskell.html

20.2 Languages

We precisely pointed out in the beginning of this thesis why we chose Haskell as language of choice. Obviously Haskell is not the only (pure) functional language and there exist a number of other alternatives which would be equally worth of systematic investigation of their use for ABS. Shortly we can conclude that the use of Haskell moves the nature of the structure of ABS computation into the light, together with compile-time guarantees, and benefits in testing and parallel implementations. Depending on each language though we get a very different direction:

LISP Being the oldest functional programming language and the 2nd oldest high-level programming language ever created and still used by many people, LISP had to be considered in the beginning of the thesis. The language has the immense powerful feature of homoiconicity: data is code and code is data at the same time. This allows a LISP program to generate data-structures, which resemble valid LSIP code thus mutating its own code at runtime. This would give immense power to create powerful abstractions in terms of ABS. Unfortunately LISP is fully interpreted and has no types and is also impure, which would probably have led to very imperative, traditional approaches to ABS. Still, there exists research [72] which implements a MAS in LISP.

Erlang The programming-model of actors [1] was the inspiration for the Erlang programming language [5], which was created in the 1980's by Joe Armstrong for Eriksson for developing distributed high reliability software in telecommunications. The implication is that, the focus would shift immediately to the use of the actor model for concurrent interaction of agents through messages. The languages type-system is strong and dynamic and thus lacks type-checking at compile-time. Thus the structure of computation plays naturally no role because we cannot look at it from the abstract perspective as we can in Haskell. Purity can not be guaranteed and due to agents being processes concurrency is everywhere, and even though it is very tamed through shared-nothing messaging semantics, this implies that repeated runs with same initial conditions might lead to different results. Obviously we could avoid implementing agents as processes but then we basically sacrifice the very heart and feature of the language.

Scala Scala is a multi-paradigm language, which also comes with an implementation of the actor-model as a library which enables to do actor-programming in the way of Erlang. It was developed in 2004 and became popular in recent years due to the increased availability of multi-core CPUs which emphasised the distributed, parallel and concurrent programming for which the actor-model is

highly suited. There exists research on using Scala for ABS [75, 124]. The benefit Scala has over Erlang is that it has type-checking at compile-time and is thus more robust, still it is impure due to side-effects and messages can contain references, thus violating the original shared-nothing semantics of Erlang.

F# Widely used in finance TODO

20.3 Actors

TODO: this seems not to fit into the narrative here, maybe it fits into discussion part or further research

The Actor-Model, a model of concurrency, was initially conceived by Hewitt in 1973 [59] and refined later on [57], [58]. It was a major influence in designing the concept of agents and although there are important differences between actors and agents there are huge similarities thus the idea to use actors to build agent-based simulations comes quite natural. The theory was put on firm semantic grounds first through Irene Greif by defining its operational semantics [50] and then Will Clinger by defining denotational semantics [25]. In the seminal work of Agha [1] he developed a semantic mode, he termed *actors* which was then developed further [2] into an actor language with operational semantics which made connections to process calculi and functional programming languages (see both below).

An actor is a uniquely addressable entity which can do the following *in response to a message*

- Send an arbitrary number (even infinite) of messages to other actors.
- Create an arbitrary number of actors.
- Define its own behaviour upon reception of the next message.

In the actor model theory there is no restriction on the order of the above actions and so an actor can do all the things above in parallel and concurrently at the same time. This property and that actors are reactive and not pro-active is the fundamental difference between actors and agents, so an agent is *not* an actor but conceptually nearly identical and definitely much closer to an agent in comparison to an object. The actor model can be seen as quite influential to the development of the concept of agents in ABS, which borrowed it from Multi Agent Systems [139]. Technically, it emphasises message-passing concurrency with share-nothing semantics (no shared state between agents), which maps nicely to functional programming concepts.

There have been a few attempts on implementing the actor model in real programming languages where the most notable ones are Erlang and Scala. Erlang was created in 1986 by Joe Armstrong for Eriksson for developing distributed high reliability software in telecommunications. It implements light-weight processes, which allows to spawn thousands of them without heavy memory over-

head. The language saw some use in implementing ABS with notable papers being [32, 33, 127, 112, 13]

Scala is a modern mixed paradigm programming language, which also allows functional programming and also incorporates a library for the actor model. It also saw the use in the implementation of ABS with a notable paper [75] and ScalABM [1] which is alibrary for ABM in economics.

The paper of [70] gives an excellent overview over the strengths and weaknesses of agent-based software-engineering, which can be directly applied to both Erlang and Scala.

Due to the very different approach and implications the actor model of concurrency implies, we don't explore it further and leave it for further research as it is beyond the focus of the thesis.

[1] https://github.com/ScalABM

Chapter 21

Favouring less power

For many 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 more potential to make mistakes. Thus we should always look for the technique with minimal power, which solves our problem sufficiently.

A very important field, which picked up ABS in recent years is economics. 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 [18]. 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 [26] defined through behaviour of individual agents. These agents are characterized by rational expectations, optimizing behaviour, having perfect information, equilibrium [42]. This approach to economics has come under heavy critizism 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 [74] 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 [42]. 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 [40]. Works which investigate ACE as a discipline and discuss its methodology are [118], [106], [9], [14]. Tesfatsion [120] defines ACE as *[...] computational modelling of economic processes (including whole economies) as open-ended dynamic systems of interacting agents.*. She gives a broad overview [119] of ACE, discusses advantages and disadvantages and giving the four primary objectives of it which are:

1. Empirical understanding: why have particular global regularities evolved and persisted, despite the absence of centralized planning and control?

2. Normative understanding: how can agent-based models be used as laboratories for the discovery of good economic designs?

3. Qualitative insight and theory generation: how can economic systems be more fully understood through a systematic examination of their potential dynamical behaviours under alternatively specified initial conditions?

4. Methodological advancement: how best to provide ACE researchers with the methods and tools they need to undertake the rigorous study of economic systems through controlled computational experiments?

It is important to understand, that ACE utilises ABS different than the social sciences do. The latter one focuses more on agent-interactions, 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 agent-interactions and are very rarely located in a spatial environment but focus more on network connections [137, 47] or avoid the notion of connectivity altogether.

To investigate this point more in-depth we implemented [1] a simulation with so called Zero Intelligence traders [48], inspired by an implementation in Python [2]. We don't go into any technical detail here but the implementation drives the main points home:

- Even though it is an agent-based model and there is a clear notion of agents in the Python code, where they are represented as objects, the agents are extremely simple. They are characterised by a single floating-point value, identifying how much value they attribute to an asset. Their behaviour is also very simple and does not change over time: they always bid randomly within their profit range. Thus we do *not* implement agents as MSFs in this case but represent them indeed only through a *Double* value, reducing the complexity of the implementation considerably.

- There are no direct agent-interactions. Although agents trade with each other, this happens through a central authority (the simulation kernel), which acts like a market with a limit order book. This reduces the complexity of the implementation considerably because there is no need for the full approach of synchronised direct agent-interactions. We could have implemented it in that way but that would have only increased complexity through the use of a quite powerful technique, which is actually not really needed because the same effect can be achieved in much simpler terms.

[1] Freely available at `https://github.com/thalerjonathan/zerointelligence`

[2] `http://people.brandeis.edu/~blebaron/classes/agentfin/GodeSunder.html`

- There is no environment whatsoever and a fully connected network is implicitly assumed because each agent can trade with all other agents. This implies that the full technique of applying an environment is not necessary, which makes the simulation a lot less complex. Still adding an environment e.g. a network would be quite simple and does not require any monadic code as the network information can be made read-only in the way as we do in Chapter 5.4.

- The only side-effect necessary in this simulation is to draw random-numbers. By fixing the seed, repeated runs of initial conditions will always lead to same output, which is guaranteed at compile time. This was already shown in Part II and is a direct consequence of Haskells type-system and explicit way of dealing with effects. Further, we focused on keeping as much code *pure* as possible thus splitting code which does not require random numbers into pure functions and only having the basic structure of the implementation running in the Rand Monad. This makes testing and reasoning considerably easier than running everything in the Rand Monad.

We are very well aware that this simple example is only one of many ACE models but even though it implements very simple *zero* intelligence agents, it shows that ABS in Haskell does not need to be as complex as the use-cases in Part II - on the contrary, ABS implementations can be very concise and highly performant in Haskell.

Chapter 22

Agents As Objects

After having undertaken this long journey on how to implement ABS pure functionally, what the general computational structures are in ABS and what benefits and drawbacks there are, at the very end of our discussion I want to return to the claim that *agents map naturally to objects* [90].

My approach of doing ABS and representing agents in pure FP can be interpret as trying to emulate objects in a purely functional way. In this case we have to say: yes agents map naturally to objects. The question is then: are there other, better mechanisms, more in FP I missed / didnt think of ... to implement ABS in FP? I hypothesize that this is probably NOT the case and that every approach in pure FP follows a roughly similar direction with only a few differences. Obviously it is apparent that both OOP and FP are not silver-bullets to ABS and both come with their benefits and drawbacks and both have their existence. Thus I hypothesize that we might see the emergence of different computation paradigms in the future which might fit better to ABS than either one.

yes agents map naturally to objects, but what kind of objects? they differ in implementation details and in this thesis proposed a pure functional approach to a notion of objects. objects in Java work different, as well as in Smalltalk and objective c. processes in the functional language Erlang can be seen as objects too as they fullfill all criteria. also cite alan kay

Part VI:

Conclusion

Chapter 23

Conclusions

The case-study strongly hints that our claim that pure FP has indeed its place in ABS is valid but 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 FP due to different techniques required. 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 FP are really needed. Still, we plan on distilling the developed techniques of the case-study into a general purpose ABS library. This should allow implementing models much easier and quicker, making the pure FP approach an attractive alternative for prototyping, opening the direction for a broad use of FP in the field of ABS.

We come back to the initial point where we started from: "Agents map naturally to Objects". oop puts modelling into the Center, fp data. some abs models require more modelling (social science), some more Data (economics), these differences show up in the Implementation: choose oop if modelling is more important, otherwise use fp. but the event driven sugarscape shows Impressively that we can achieve similar abstractions found in oop with fp, thus we can agree: "yes, agents mal naturally onto objects" but throughout the journey of this thesis we have come to a much richer and deeper understanding of the meaning behind those words and we have understood that objects have a variety of representations and need to be thought more abstract than just in their technical reptesentations in java, Python or c++ for example. whether they are newtypes, tuples, GADTs, monads, comonads, arrows, SFs or MSFs: those are all valid ways of representing agents with varying degree of abstraction, flexibility and power.

23.1 Further Research

clearly outline the ideas for further research

23.1.1 A general purpose library

generalise concepts explored into a pure functional ABS library in Haskell (called chimera)

23.1.2 Dependent and linear types

dependent types and linear types are the next big step, towards a stronger formalisation of agents and ABS, focus on the equilibrium - totality correspondence

23.1.3 Concurrent event-driven ABS

stm based concurrency for event-driven ABS using parallel DES. challenge is the time-warp implementation using monads. in general it should be easy to roll-back agents actions but with monads we have to be careful - for some monads rolling back is not neccessary e.g. rand and reader, for others it is, and for some it is impossible e.g. IO

References

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

[2] Agha, G. A., Mason, I. A., Smith, S. F., and Talcott, C. L. A Foundation for Actor Computation. *J. Funct. Program. 7*, 1 (Jan. 1997), 1–72.

[3] Allen, C., and Moronuki, J. *Haskell Programming from First Principles*. Allen and Moronuki Publishing, July 2016. Google-Books-ID: 5FaXDAEACAAJ.

[4] 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.

[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] 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.

[8] Balci, O. Verification, Validation, and Testing. In *Handbook of Simulation*, J. Banks, Ed. John Wiley & Sons, Inc., 1998, pp. 335–393.

[9] 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.

[10] Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984. Google-Books-ID: KbZFAAAAYAAJ.

[11] Beck, K. *Test Driven Development: By Example*, 01 edition ed. Addison-Wesley Professional, Boston, Nov. 2002.

[12] 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.

[13] Bezirgiannis, N. *Improving Performance of Simulation Software Using Haskells Concurrency & Parallelism*. PhD thesis, Utrecht University - Dept. of Information and Computing Sciences, 2013.

[14] 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.

[15] Borshchev, A., and Filippov, A. From System Dynamics and Discrete Event to Practical Agent Based Modeling: Reasons, Techniques, Tools.

[16] 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.

[17] 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.

[18] Bowles, S., Edwards, R., and Roosevelt, F. *Understanding Capitalism: Competition, Command, and Change*, 3 edition ed. Oxford University Press, New York, Mar. 2005.

[19] Brady, E. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming 23*, 05 (2013), 552–593.

[20] Brady, E. *Type-Driven Development with Idris*. Manning Publications Company, 2017. Google-Books-ID: eWzEjwEACAAJ.

[21] Burnstein, I. *Practical Software Testing: A Process-Oriented Approach*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[22] Church, A. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics 58*, 2 (Apr. 1936), 345–363.

[23] 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.

[24] Claessen, K., and Hughes, J. Testing Monadic Code with QuickCheck. *SIGPLAN Not. 37*, 12 (Dec. 2002), 47–59.

[25] Clinger, W. D. Foundations of Actor Semantics. Tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA, 1981.

[26] Colell, A. M. *Microeconomic Theory*. Oxford University Press, 1995. Google-Books-ID: dFS2AQAACAAJ.

[27] Collier, N., and Ozik, J. Test-driven agent-based simulation development. In *2013 Winter Simulations Conference (WSC)* (Dec. 2013), pp. 1551–1559.

[28] 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.

[29] 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.

[30] De Jong, T. Suitability of Haskell for Multi-Agent Systems. Tech. rep., University of Twente, 2014.

[31] De Vries, E. An in-depth look at quickcheck-state-machine, Jan. 2019.

[32] 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.

[33] Di Stefano, A., and Santoro, C. eXAT: an Experimental Tool for Programming Multi-Agent Systems in Erlang. Tech. rep., 2007.

[34] Diehl, S. WHAT I WISH I KNEW WHEN LEARNING HASKELL.

[35] 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.

[36] 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.

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

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

[39] Evensen, P., and Märdin, M. An Extensible and Scalable Agent-Based Simulation of Barter Economics. Master's thesis, Chalmers University of Technology, Göteborg, 2010.

[40] Farmer, J. D., and Foley, D. The economy needs agent-based modelling. *Nature 460*, 7256 (Aug. 2009), 685–686.

[41] 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.

[42] Focardi, S. Is economics an empirical science? If not, can it become one? *Frontiers in Applied Mathematics and Statistics 1* (2015), 7.

[43] Fujimoto, R. M. Parallel Discrete Event Simulation. *Commun. ACM 33*, 10 (Oct. 1990), 30–53.

[44] 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.

[45] 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.

[46] Gintis, H. The Emergence of a Price System from Decentralized Bilateral Exchange. *Contributions in Theoretical Economics 6*, 1 (2006), 1–15.

[47] Glasserman, P., and Young, P. Contagion in Financial Networks. SSRN Scholarly Paper ID 2681392, Social Science Research Network, Rochester, NY, Oct. 2015.

[48] 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.

[49] Gregory, J. *Game Engine Architecture, Third Edition*. Taylor & Francis, Mar. 2018.

[50] Grief, I., and Greif, I. SEMANTICS OF COMMUNICATING PARALLEL PROCESSES. Tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA, 1975.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] Harris, T., and Peyton Jones, S. Transactional memory with data invariants.

[56] 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.

[57] 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.

[58] Hewitt, C. Actor Model of Computation: Scalable Robust Information Systems. *arXiv:1008.1459 [cs]* (Aug. 2010). arXiv: 1008.1459.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] Hughes, J. Why Functional Programming Matters. *Comput. J. 32*, 2 (Apr. 1989), 98–107.

[64] Hughes, J. Generalising Monads to Arrows. *Sci. Comput. Program. 37*, 1-3 (May 2000), 67–111.

[65] 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.

[66] 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.

[67] Hutton, G. *Programming in Haskell*. Cambridge University Press, Aug. 2016. Google-Books-ID: 1xHPDAAAQBAJ.

[68] 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.

[69] Jankovic, P., and Such, O. Functional Programming and Discrete Simulation. Tech. rep., 2007.

[70] Jennings, N. R. On Agent-based Software Engineering. *Artif. Intell. 117*, 2 (Mar. 2000), 277–296.

[71] 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.

[72] Kawabe, Y., Mano, K., and Kogure, K. The Nepi2programming System: A pi-Calculus-Based Approach to Agent-Based Programming. In *Formal Approaches to Agent-Based Systems* (Apr. 2000), Springer, Berlin, Heidelberg, pp. 90–102.

[73] 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.

[74] Kirman, A. *Complex Economics: Individual and Collective Rationality*. Routledge, London ; New York, NY, July 2010.

[75] Krzywicki, D., Turek, W., Byrski, A., and Kisiel-Dorohinicki, M. Massively concurrent agent-based evolutionary computing. *Journal of Computational Science 11* (Nov. 2015), 153–162.

[76] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM 21*, 7 (July 1978), 558–565.

[77] Lipovaca, M. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, Apr. 2011. Google-Books-ID: QesxXj_ecD0C.

[78] 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.

[79] 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.

[80] Macal, C. M. Everything you need to know about agent-based modelling and simulation. *Journal of Simulation 10*, 2 (May 2016), 144–156.

[81] MacLennan, B. J. *Functional Programming: Practice and Theory*. Addison-Wesley, Jan. 1990. Google-Books-ID: JqhQAAAAMAAJ.

[82] Marlow, S. *Parallel and Concurrent Programming in Haskell*. O'Reilly, 2013. Google-Books-ID: k0W6AQAACAAJ.

[83] Marlow, S., Peyton Jones, S., and Singh, S. Runtime Support for Multicore Haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2009), ICFP '09, ACM, pp. 65–78.

[84] McMillan, K. L. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[85] 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.

[86] Michaelson, G. *An Introduction to Functional Programming Through Lambda Calculus*. Courier Corporation, 2011. Google-Books-ID: gKvw-PtvsSjsC.

[87] 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.

[88] 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.

[89] 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.

[90] 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.

[91] Odell, J. Objects and Agents Compared. *Journal of Object Technology 1*, 1 (May 2002), 41–53.

[92] Okasaki, C. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1999.

[93] 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.

[94] 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.

[95] O'Sullivan, B., Goerzen, J., and Stewart, D. *Real World Haskell*, 1st ed. O'Reilly Media, Inc., 2008.

[96] 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.

[97] 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.

[98] Perez, I. *Extensible and Robust Functional Reactive Programming*. Doctoral Thesis, University Of Nottingham, Nottingham, Oct. 2017.

[99] 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.

[100] Perez, I., and Nilsson, H. Testing and Debugging Functional Reactive Programming. *Proc. ACM Program. Lang. 1*, ICFP (Aug. 2017), 2:1–2:27.

[101] 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.

[102] 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.

[103] 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.

[104] 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.

[105] Program, T. U. F. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[106] Richiardi, M. Agent-based Computational Economics. A Short Introduction. LABORatorio R. Revelli Working Papers Series 69, LABORatorio R. Revelli, Centre for Employment Studies, 2007.

[107] Robinson, S. *Simulation: The Practice of Model Development and Use*. Macmillan Education UK, Sept. 2014. Google-Books-ID: Dtn0oAEACAAJ.

[108] 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.

[109] 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.

[110] 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.

[111] 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.

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

[113] Siebers, P.-O., and Aickelin, U. Introduction to Multi-Agent Simulation. *arXiv:0803.3905 [cs]* (Mar. 2008). arXiv: 0803.3905.

[114] Sorokin, D. Aivika 3: Creating a Simulation Library based on Functional Programming, 2015.

[115] Stump, A. *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA, 2016.

[116] Sulzmann, M., and Lam, E. Specifying and Controlling Agents in Haskell. Tech. rep., 2007.

[117] 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.

[118] Tesfatsion, L. Agent-Based Computational Economics. Computational Economics, EconWPA, Aug. 2002.

[119] Tesfatsion, L. Agent-Based Computational Economics: A Constructive Approach to Economic Theory. Handbook of Computational Economics, Elsevier, 2006.

[120] Tesfatsion, L. Agent-based Computational Economics (ACE) - Growing Economies from the Bottom Up. Tech. rep., May 2017.

[121] Thaler, J., and Siebers, P.-O. The Art Of Iterating: Update-Strategies in Agent-Based Simulation.

[122] 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.

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

[124] Todd, A. B., Keller, A. K., Lewis, M. C., and Kelly, M. G. *Multi-agent System Simulation in Scala: An Evaluation of Actors for Parallel Simulation*.

[125] Turing, A. M. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society s2-42*, 1 (1937), 230–265.

[126] 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.

[127] 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.

[128] 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.

[129] Vipindeep, V., and Jalote, P. List of Common Bugs and Programming Practices to avoid them. Tech. rep., Indian Institute of Technology, Kanpur, Mar. 2005.

[130] 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.

[131] 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.

[132] Wadler, P. How to Declare an Imperative. *ACM Comput. Surv. 29*, 3 (Sept. 1997), 240–263.

[133] 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.

[134] Weaver, I. Replicating Sugarscape in NetLogo, Oct. 2009.

[135] Weiss, G. *Multiagent Systems*. MIT Press, Mar. 2013. Google-Books-ID: WY36AQAAQBAJ.

[136] Wilensky, U., and Rand, W. *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NETLogo*. MIT Press, 2015.

[137] Wilhite, A. Economic Activity on Fixed Networks. Handbook of Computational Economics, Elsevier, 2006.

[138] 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.

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

[140] 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 section we step-by-step develop a correct-by-construction implementation. Note that the constant parameters *populationSize, infectedCount, contactRate, infectivity, illnessDuration* are defined globally and omitted for clarity.

Computing the dynamics of a SD model happens by integrating the time over the equations. So conceptually we treat our SD model as a continuous function which is defined over time = 0 -¿ infinity and at each point in time outputs the values of each stock. 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 our SD function: for each time-step we have the values of the 3 stocks:

```
type SIRStep = (Time, Double, Double, Double)
```

Next we define our continuous SD function which we obviously make a signal function. It has no input, because a SD system is only defined in its own terms and parameters without external input and has as output the *SIRStep*. Thus we define the following function type:

```
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 → SF (a, c) (b, c) → SF a b* function for that. It takes an initial value and a feedback signal function which receives the input a and the previous (or initial) value of the feedback 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. Our 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 a 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 our flows which are *infection rate* and *recovery rate*. The formulas for both of them can be seen in equations TODO (refer to the differential equations). This directly translates into Haskell code:

```
      let infectionRate = (i * contactRate * s * infectivity) / populationSize
          recoveryRate  = i / illnessDuration
```

Next we need to compute the values of the three stocks, following the formulas of TODO (refer to the Integral formulas). 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)
```

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.

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.

```
runSD :: Time -> DTime -> [SIRStep]
runSD t dt = embed sir ((), steps)
  where
    steps = replicate (floor (t / dt)) (dt, Nothing)
```

A.1 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 happening, neither in our nor Yampas library code 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.

A.1.1 Results

Although we have translated our model specifications directly into code we still we 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.2 Conclusion

In this paper we have shown how to implement System Dynamics in a way that the resulting implementation is correct-by-construction, where the gap between the formal model specifications and the actual implementation in code is closed. We used the pure functional programming language Haskell for it and built on the Functional Reactive Programming concept to express our continuous-time simulation. The provided abstractions of Haskell and Functional Reactive Programming allowed to close the gap between the specification and implementation and further guarantee the absence of non-deterministic influences already at compile-time, making our correct-by-construction claims even stronger.

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.

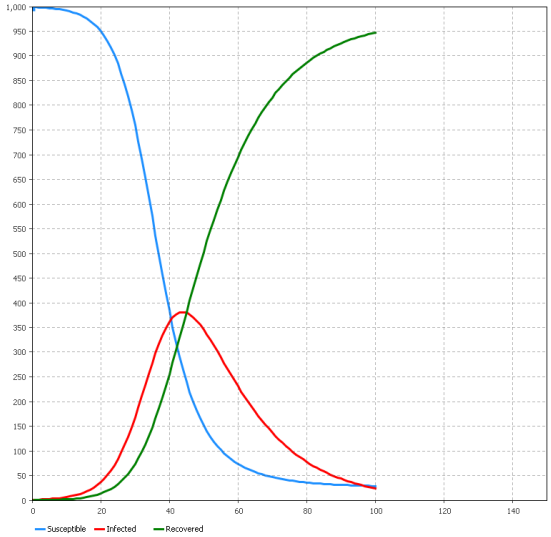
Obviously the numerical well behaviour depends on the integral function which uses the rectangle rule. We showed that for the SIR model and small enough Δt, the rectangle rule works well enough. Still it might be of benefit if we provide more sophisticated numerical integration like Runge-Kutta methods. We leave this for further research.

The key strength of System Dynamic simulation packages is their visual representation which allows non-programmers to express 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.

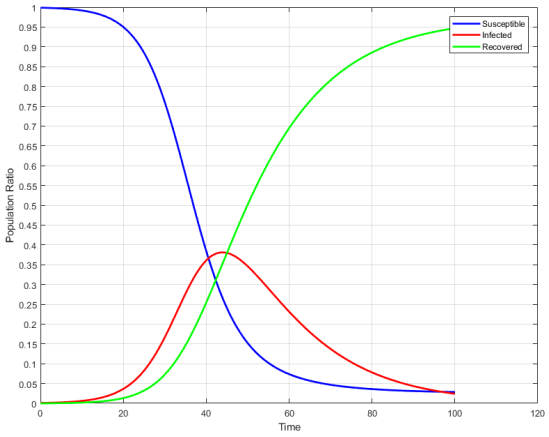
Also we are very well aware that due to the vast amount of visual simulation packages available for System Dynamics, there is no big need for implementing such simulations directly in code. Still we hope that our pure functional approach with Functional Reactive Programming might spark an interest in 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.3 Full Code

```
populationSize :: Double
populationSize = 1000
```



(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.

```
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

Obviously we wanted our implementation to be correct, which means we validated it against the informal reports in the book. Also we use the work of [134] [1] which replicated Sugarscape in NetLogo and reported on it [2].

In addition to the informal descriptions of the dynamics, we implemented tests which conceptually check the model for emergent properties with hypotheses shown and expressed in the book. Technically speaking we have implemented that with unit-tests where in general we run the whole simulation with a fixed scenario and test the output for statistical properties which, in some cases is straight forward e.g. in case of Trading the authors of the Sugarscape model explicitly state that the standard deviation is below 0.05 after 1000 ticks. Obviously one needs to run multiple replications of the same simulation, each with a different random-number generator and perform a statistical test depending on what one is checking: in case of an expected mean one utilises a t-test and in case of standard-deviations a chi-squared test.

B.1 Terracing

Our implementation reproduces the terracing phenomenon as described on page TODO in Animation 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

[1] `https://www2.le.ac.uk/departments/interdisciplinary-science/research/replicating-sugarscape`

[2] Note that lending didn't properly work in their NetLogo code and that they didn't implement Combat

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 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' [134] 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 sugarlevel 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.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.4 Migration

With the information provided by [134] 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 haven't implemented a regression-test for this property as we couldn't come up with a reasonable straight forward approach to implement it.

B.5 Pollution and Diffusion

With the information provided by [134] we could replicate the pollution behaviour as visible in the NetLogo implementation as well. We haven't implemented a regression-test for this property as we couldn't come up with a reasonable straight forward approach to implement it.

B.6 Mating

We could not replicate Figure III-1 (TODO: page) - 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 [134], 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 won't engage in mating solved that and produces the same swings as in [134]. 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.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.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 [134].

B.9 Combat

Unfortunately [134] 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.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 [134] 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 [134].

B.11 Trading

For trading we had a look at the NetLogo implementation of [134]: there an agent engages in trading with its neighbours *over multiple rounds* until either MRSs cross over or no trade has happened anymore. Because [134] 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 convergence 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 won't 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.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 [134] 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.