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

March 6, 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: reconcile all code into thesis code folder?

TODO: 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,...

TODO research:

1. Agent-Interaction Property-Based testing using quickcheck-state-machine and [20] `http://www.well-typed.com/blog/2019/01/qsm-in-depth/`

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

3. Dependent types: a full treatment of them in context of ABS is beyond the scope of this phd (and also don't have enough knowledge and time), and would be enough for a new phd / habilitation. Can we outline basics and ideas without going too much into depth or is this not possible in a phd?

4. Gintis Case-Study without dependent types

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:

Beginnings

Chapter 1

Introduction

The traditional approach to Agent-Based Simulation (ABS) has so far always been object-oriented techniques, due to the influence of the seminal work of Epstein et al [23] 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* [59] which still holds up today.

This thesis challenges that 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 [36, 11] 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 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 [79] - 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 [78] - 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 [47], 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 [19, 76, 43].

A multi-method simulation library in Haskell called *Aivika 3* is described in the technical report [75]. 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 [7] the author investigates Haskells' parallel and concurrency features to implement (amongst others) *HLogo*, a Haskell clone of the NetLogo [90] 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 [21, 82, 73] using the functional programming language Erlang [3] 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 [92]. 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 [7] 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 [47] discusses a framework, which allows to map Agent-Based Simulations to Graphics Processing Units (GPU). Amongst others they use the SugarScape model [23] 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 [43] 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 [71, 83]. 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 [18] 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 [31]. 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 [9] 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 [10] 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 [77], 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 [28] 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

History, methodology (what is the purpose of ABS: 3rd way of doing science: exploratory, helps understand real-world phenomena), classification according to [49], ABS vs. MAS, event- vs. time-driven [53], 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 [92]. 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 [74, 92, 49, 60]:

- 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 [22] 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) [92], [89]. 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 [49]: 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 [42] which tried to replicate the work of Gintis [27]. 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 [15] 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. [42] 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 [77], 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 [28]. 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: [84]

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

3.1.2 Verification & Validation

Introduction Verification & Validation (V & V in the context of ABS).

Research on TDD of ABS is quite new and thus there exist relative few publications. The work [16] is the first to discusses how to apply the TDD approach 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 [58] and make the important point that such a software need to be designed to be sufficiently modular otherwise testing becomes too cumbersome and involves too many parts. The paper [4] discusses a similar approach to DES in the AnyLogic software toolkit.

The paper [62] 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. This approach will become of importance later on in our SIR case study.

The paper [12] propose property-driven design of robot swarms. They propose a top-down approach by specifying properties a swarm of robots should have from which a prescriptive model is created, which properties are verified using model checking. Then a simulation is implemented following this prescriptive and verified model after then the physical robots are implemented. The authors identify the main difficulty of implementing such a system that the engineer must *"think at the collective-level, but develop at the individual-level*. It is arguably true that this also applies to implementing agent-based models and simulations where the same collective-individual separation exists from which emergent system behaviour of simulations emerges - this is the very foundation of the ABS methodology.

The paper [30] 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 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.

Although the work on TDD is scarce in ABS, there exists quite some research on applying TDD and unit-testing to multi-agent systems (MAS). Although MAS is a different discipline than ABS, the latter one has derived many technical concepts from the former one thus testing concepts applied to MAS might also be applicable to ABS. The paper [56] is a survey of testing in MAS. It distinguishes between unit tests which tests units that make up an agent, agent tests which test the combined functionality of units that make up an agent, integration tests which test the interaction of agents within an environment and observe emergent behaviour, system test which test the MAS as a system running at the target environment and acceptance test in which stakeholders verify that the software meets their goal. Although not all ABS simulations need acceptance and system tests, still this classification gives a good direction and can be directly transferred to ABS.

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 [13]. This is a fundamentally different approach to computing than imperative programming (includeing established object-orientation) which roots lie in the Turing Machine [80]. 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 [50] 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.

[2] 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.

[50] 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 [50] 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 [61] 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 [41]. Other, more exhaustive books on learning Haskell are [46, 2]. For an introduction to programming with the Lambda-Calculus we refer to [54]. For more general discussion of functional programming we refer to [38, 50, 36].

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: [55, 85, 86, 87, 44].

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: [51, 63, 32, 52].

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.

[50] 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 [50] and [6].

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 [36] 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 [37, 36], is applicable to a number of real-world problems [63] 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 [39, 40] as implemented in the library Yampa [35, 17, 57].

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

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

Yampa provides a number of combinators for expressing time-semantics, events and state-changes of the system. They allow to change system behaviour in case of events, run signal functions and generate stochastic events and random-number streams. We shortly discuss the relevant combinators and concepts we use throughout the paper. For a more in-depth discussion we refer to [35, 17, 57].

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)* $\rightarrow$ *(c* $\rightarrow$ *SF a b)* $\rightarrow$ *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* $\Rightarrow$ *g* $\rightarrow$ *Time* $\rightarrow$ *b* $\rightarrow$ *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)* $\Rightarrow$ *g* $\rightarrow$ *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* $\rightarrow$ *(a, [(DTime, Maybe a)])* $\rightarrow$ *[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 [39, 40].

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 [64], 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 [68].

```
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 [64] 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 [67, 66]:

```
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 [53]. 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 [48, 78]. 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 [53].

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 [79], 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.

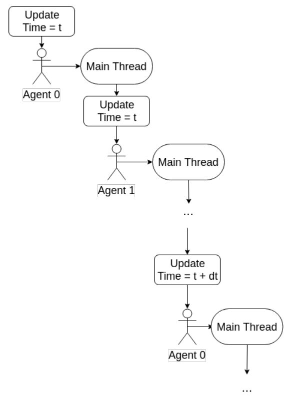


Figure 4.1: Control flow in the Sequential Strategy.

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.

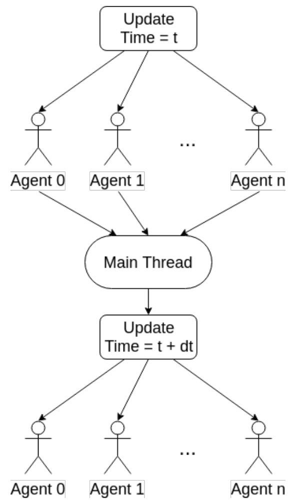


Figure 4.2: Control flow in the Parallel Strategy.

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.

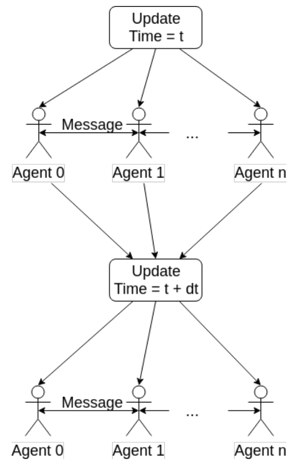


Figure 4.3: Control flow in the Concurrent Strategy.

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 [34] for which [29] and [14] 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.

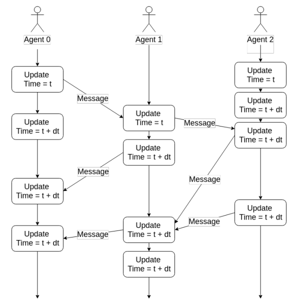


Figure 4.4: Control flow in the Actor Strategy.

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 8 on parallel ABS. The *actor* strategy is not used in this thesis but its implementation follows directly from the Chapters 5 and 8: 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 [58] 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:

How

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 [45], 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.

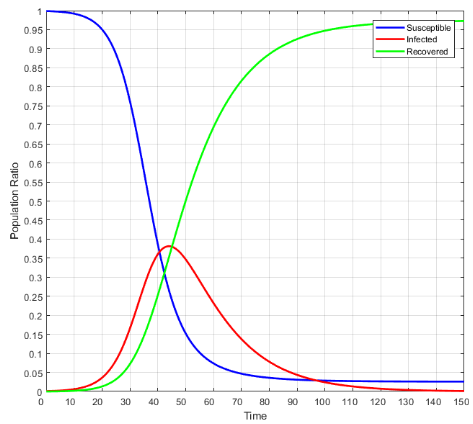


Figure 5.2: Dynamics of the SIR compartment model using the System Dynamics approach. Population Size $N = 1{,}000$, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time-steps. Generated using our pure functional SD approach (see Chapter 10.4.2).

This model was also formalized using System Dynamics (SD) [69]. 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 [93].

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 [8]. 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 [49], 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 [39] 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 [8] 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*:

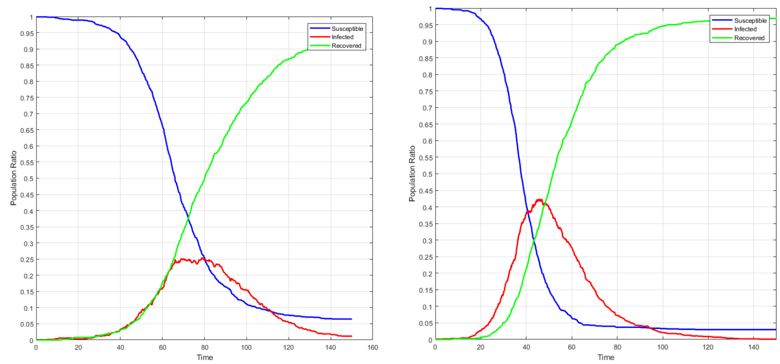
```
runSimulation :: RandomGen g => g -> Time -> DTime -> [SIRState] -> [[SIRState]]
runSimulation g t dt as
    = embed (stepSimulation sfs as) ((), dts)
  where
    steps     = floor (t / dt)
    dts       = replicate steps (dt, Nothing)
    n         = length as
    (rngs, _) = rngSplits g n [] -- 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, [57, 17] discusses implementing this in Yampa. The function *stepSimulation* is an implementation of such a closed feedback-loop. It takes the current signal functions and states of all agents, runs them all in parallel and returns this step's new agent states. Note the use of *notYet*, which is required because in Yampa switching occurs immediately at $t = 0$. If we don't delay the switching at $t = 0$ until the next step, we would enter an infinite switching loop - *notYet* simply delays the first switching until the next time-step.

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

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.

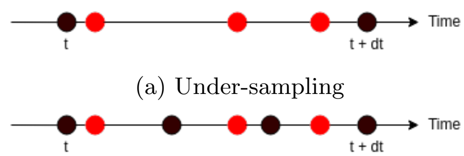
Its first argument is the pairing-function, which pairs up the input to the signal functions - it has to preserve the structure of the signal function collection. The second argument is the collection of signal functions to run. The third argument is a signal function generating the switching event.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

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

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



(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.

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 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 [91]. 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 [67], 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 [23] 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 [58], 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.

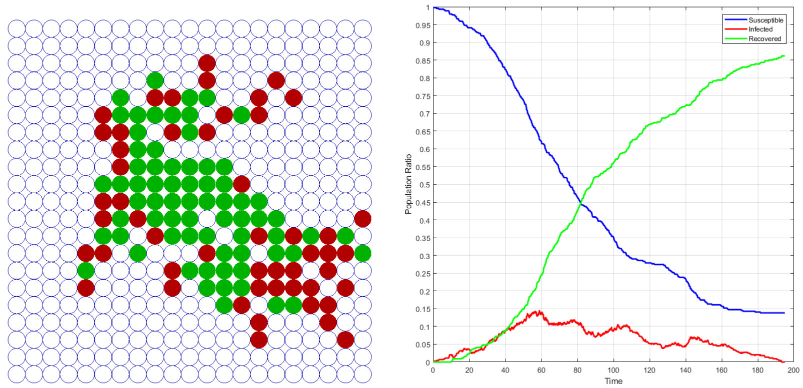
```
simulationStep :: RandomGen g => [(SIRAgent g, Disc2dCoord)]
               -> SIREnv -> SF (Rand g) () SIREnv
simulationStep sfsCoords env = MSF (\_ -> do
  let (sfs, coords) = unzip sfsCoords
  -- run agents sequentially but with shared, read-only environment
  ret <- mapM (`unMSF` env) sfs
  -- 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 [68]. Also by using FRP we gain all the benefits from it and can use research on testing, debugging and exploring FRP systems [68, 65].

Also we showed how to implement the *parallel* update-strategy [79] 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 [58]. We ran the simulation until $t = 100$ on a 51x51 (2,601 agents) with $\Delta t = 0.1$ (unknown in RePast) and averaged 8 runs. The performance results make the lack of speed of our approach quite clear: the pure functional approach needs around 72.5 seconds whereas the Java RePast version just 10.8 seconds on our machine to arrive at $t = 100$. It must be mentioned, that RePast does implement an event-driven approach to ABS, which can be much more performant [53] 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 [32, 33], 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 [72] 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 [53], 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) [93]. 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) [25], 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

[1] The code of all steps can be accessed freely from: `https://github.com/thalerjonathan/`

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 [5]; 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 [23]. 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 [49], 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 [23] 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 position - it is equally probable of being scheduled in any position. The semantics of the model require to step the agents sequentially but ideally

`phd/tree/master/public/towards/SugarScape/sequential` 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.

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

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.

TODO: link to full implementation in THESIS directory!

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.

```haskell
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: $feedback :: Monad\ m \Rightarrow c \rightarrow MSF\ m\ (a,\ c)\ (b,\ c) \rightarrow MSF\ m\ a\ b$. It takes an initial value of type c and an MSF which takes in addition to its input a also the given type c and outputs in addition to type b also the type c, which clearly indicates the read/write property of type c. The function returns a new MSF which only operates on

a as input and returns b as output by running the provided MSF and feeding back the c (with the initial c at the first call).

```
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

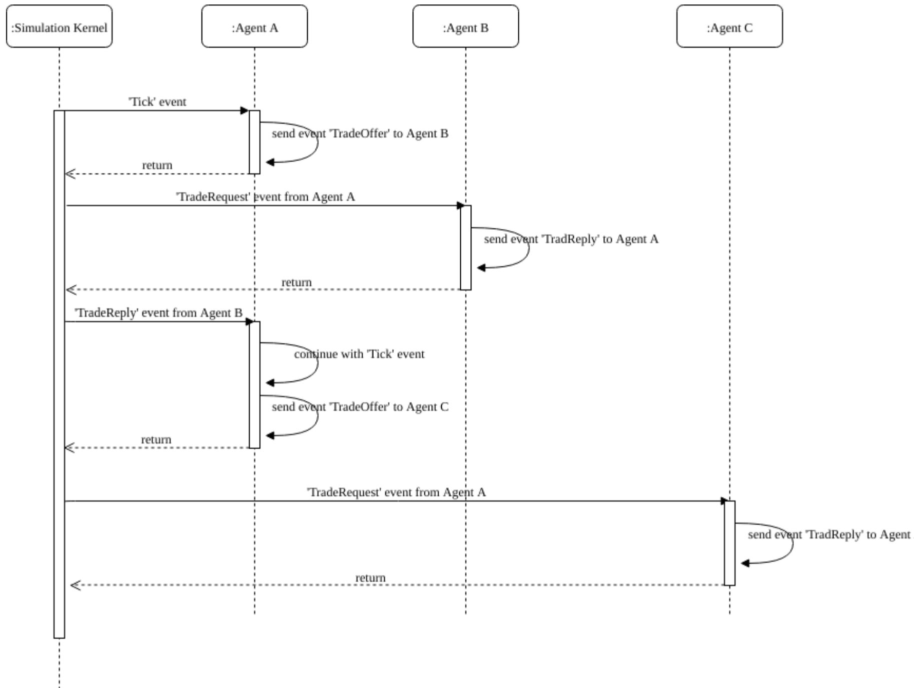


Figure 6.1: Sequence diagram of synchronous agent-interaction with the trading use-case. Upon the handling of the 'Tick' event, Agent A looks for trading partners and finds Agent B within its neighbourhood and sends a 'TradingOffer' message. Agent B replies to this message and Agent A continues with 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.

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 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 sf evt absState env g = (ao, obs, sf', absState', env', g')
  where
    -- extract the monadic function to run
    sfAbsState = unMSF sf evt
    -- peel away one State layer: ABSState
    sfEnvState = runStateT sfAbsState absState
    -- peel away the second State layer: SugEnvironment
    sfRand     = runStateT sfEnvState env
    -- peel away the 3rd and last layer: Rand Monad
    (((((ao, obs), sf'), absState'), env'), g') = runRand sfRand 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.

6.2.4 Discussion

maybe refactor sugarscape: write more pure functions which make the distinction between read only and write only of the agent state clear e.g. is agent fertile can be a pure function with state passed explicitly. also always make clear that we can write functions which are monadic but access e.g. only the agent state or the Environment state Without the full monad Transformer stack: try to refactor more in this direction and make thus very clear in the discussion

synchronous agent-interactions are cumbersome and clearly more complex than direct method invocation in oop which does the same. with the goal of staying pure we dont have much other options.

6.3 Event-Driven SIR

can show how we extend it to include time-stamps, also show its MUCH faster

6.4 Discussion

Although there are similarities to the work of [9] (the use of messages and the problem of when to advance time in models with arbitrary number synchronised agent-interactions), we approach our agents differently. 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" [9]. Note that the authors of [9] define their agents with a polymorphic agent-state type s, which implies that without knowledge of the specific type of s there would be no way of accessing the state, rendering it in fact also fully encapsulated. The problem of advancing time in our approach is solved not exactly the same but conceptually it is the same: 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.

our eventdriven approach makes heavy use of 2 state monads, thus one might ask what the benefits are, 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. on the other hand even our stateful programming is highly restricted to only those 2 local datatypes which makes it much more manageable than unrestricted data mutation 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 guaranteed at compile-time

quote carmack (`http://www.gamasutra.com/view/news/169296/Indepth_Functional_programming_in_C.php`): the main difficulty as a developer in software programming is to keep track of the states a program can be in and reason about them and their Validity

TODO: report LoC and compare it with other implementations we found on the internet

Chapter 7

Generalising the structure of agent 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.

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?

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 [81]: 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/

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

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.

Part III:

Why

Chapter 8

Parallel ABS

Establish how concurrency and parallelism can be made easily available in ABS using pure functional programming. Mostly follow STM paper and add pure parallelism in ABS. Make clear that Haskell allows to distinguish between pure, deterministic parallelism and impure, non-deterministic concurrency.

Also discuss where there is potential for adding parallelism: using data-parallel data-structures for the environment so cells can be updated in parallel, in time-driven ABS agents can be updated in parallel using parMap because they all act conceptually at the same time (and if they don't run in monadic code). 0% finished.

we can implement everything except synchronous direct agent-interactions atm: if agent-interaction is one-way e.g. paying back a loan then this is no problem. thus the following parts of the Sugarscape are not possible with our current STM approach: mating, trading and lending because all 3 require direct agent-to-agent interaction over multiple steps. We leave the problem of developing such an algorithm / implementation for further research.

use the STM paper but provide more haskell code

8.1 Discussion

Further research: implementing synchronous agent-interactions in a concurrent implementation with STM, the approach of executing dependent functions in different layers `https://academic.oup.com/bib/article/11/3/334/225993` might allow even further speed-up, also the same paper: "Mobile discrete simulations consisting of agents navigating a grid like environment are less well-suited to a GPU implementation as they are traditionally implemented in sequential environments. Parallel implementations of discrete mobile systems on the GPU must explicitly handle collision conditions which result from agents simultaneously moving to the same discrete location."

Chapter 9

Property-Based Testing in ABS

In this chapter we present another reason *why* pure functional ABS is of benefit: we can make direct use of property-based testing. This topic falls into the category of verification & validation and is of fundamental importance to ABS. We first will introduce the problem of verification & validation (V&V) of an ABS implementation in general where we very briefly discuss ways in which pure functional ABS can be of benefit. Then we present property-based testing and how it can be applied to explanatory models, in the case of the agent-based SIR model and to exploratory models, in the case of the Sugarscape model.

9.1 Verification in ABS

General there are the following basic verification & validation requirements to ABS [70], 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.

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 [68]

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

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

9.2 Property-Based Testing

Follow property-based testing as prototyped for towards paper. Also discuss property-based testing as explored in the SIR (time-driven) and Sugarscape (event-driven) case.

9.3 Explanatory Model Testing: SIR

Finding optimal Δt The selection of the right Δt can be quite difficult in FRP because we have to make assumptions about the system a priori. One could just play it safe with a very conservative, small $\Delta t < 0.1$ but the smaller Δt, the lower the performance as it multiplies the number of steps to calculate. Obviously one wants to select the *optimal* Δt, which in the case of ABS is the largest possible Δt for which we still get the correct simulation dynamics. To find out the *optimal* Δt one can make direct use of the black-box tests: start with a large $\Delta t = 1.0$ and reduce it by half every time the tests fail until no more tests fail - if for $\Delta t = 1.0$ tests already pass, increasing it may be an option. It is important to note that although isolated agent behaviour tests might result in larger Δt, in the end when they are run in the aggregate system, one needs to sample the whole system with the smallest Δt found amongst all tests. Another option would be to apply super-sampling to just the parts which need a very small Δt but this is out of scope of this paper.

Agents as signals Agents *might* behave as signals in FRP which means that their behaviour is completely determined by the passing of time: they only change when time changes thus if they are a signal they should stay constant if time stays constant. This means that they should not change in case one is sampling the system with $\Delta t = 0$. Of course to prove whether this will *always* be the case is strictly speaking impossible with a black-box verification but we can gain a good level of confidence with them also because we are staying pure. It is only through white-box verification that we can really guarantee and prove this property.

Black-Box Verification The interface of the agent behaviours are defined below. When running the SF with a given Δt one has to feed in the state of all the other agents as input and the agent outputs its state it is after this Δt.

```
data SIRState
  = Susceptible
  | Infected
  | Recovered

type SIRAgent = SF [SIRState] SIRState

susceptibleAgent :: RandomGen g => g -> SIRAgent
infectedAgent :: RandomGen g => g -> SIRAgent
recoveredAgent :: SIRAgent
```

Finding optimal Δt Obviously the *optimal* Δt of the SIR model depends heavily on the model parameters: contact rate β and illness duration δ. We fixed them in our tests to be $\beta = 5$ and $\delta = 15$. By using the isolated behaviour tests we found an optimal $\Delta t = 0.125$ for the susceptible behaviour and $\Delta t = 0.25$ for the infected behaviour.

Agents as signals Our SIR agents *are* signals due to the underlying continuous nature of the analytical SIR model and to some extent we can guarantee this through black-box testing. For this we write tests for each individual behaviour as previously but instead of checking whether agents got infected or have recovered we assume that they stay constant: they will output always the same state when sampling the system with $\Delta t = 0$. The tests are conceptual the complementary tests of the previous behaviour tests so in conjunction with them we can assume to some extent that agents are signals. To prove it, we need to look into white-box verification as we cannot make guarantees about properties which should hold *forever* in a computational setting.

Recovered Behaviour The implementation of the recovered behaviour is as follows:

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

Just by looking at the type we can guarantee the following:

- it is pure, no side-effects of any kind can occur
- no stochasticity possible because no RNG is fed in / we don't run in the random monad

The implementation is as concise as it can get and we can reason that it is indeed a correct implementation of the recovered specification: we lift the constant function which returns the Recovered state into an arrow. Per definition and by looking at the implementation, the constant function ignores its input and returns always the same value. This is exactly the behaviour which we need for the recovered agent. Thus we can reason that the recovered agent will return Recovered *forever* which means our implementation is indeed correct.

Because we use multiple replications in combination with QuickCheck obviously results in longer test-runs (about 5 minutes on my machine) In our implementation we utilized the FRP paradigm. It seems that functional programming and FRP allow extremely easy testing of individual agent behaviour because FP and FRP compose extremely well which in turn means that there are no global dependencies as e.g. in OOP where we have to be very careful to clean up the system after each test - this is not an issue at all in our *pure* approach to ABS.

Simulation Dynamics We won't go into the details of comparing the dynamics of an ABS to an analytical solution, that has been done already by [48]. What is important is to note that population-size matters: different population-size results in slightly different dynamics in SD =¿ need same population size in ABS (probably...?). Note that it is utterly difficult to compare the dynamics of an ABS to the one of a SD approach as ABS dynamics are stochastic which explore a much wider spectrum of dynamics e.g. it could be the case, that the infected agent recovers without having infected any other agent, which would lead to an extreme mismatch to the SD approach but is absolutely a valid dynamic in the case of an ABS. The question is then rather if and how far those two are *really* comparable as it seems that the ABS is a more powerful system which presents many more paths through the dynamics.

White-Box Verification In the case of the SIR model we have the following invariants:

- A susceptible agent will *never* make the transition to recovered.
- An infected agent will *never* make the transition to susceptible.
- A recovered agent will *forever* stay recovered.

All these invariants can be guaranteed when reasoning about the code. An additional help will be then coverage testing with which we can show that an infected agent never returns susceptible, and a susceptible agent never returned infected given all of their functionality was covered which has to imply that it can never occur!

We will only look at the recovered behaviour as it is the simplest one. We leave the susceptible and infected behaviours for further research / the final thesis because the conceptual idea becomes clear from looking at the recovered agent.

9.4 Exploratory Model Testing: Sugarsape

We implemented a number of tests for agent functions which just cover the part of an agents behaviour: checks whether an agent has died of age or starved to death, the metabolism, immunisation step, check if an agent is a potential borrower or fertile, lookout, trading transaction. What all these functions have in common is that they are not pure computations like utility functions but require an agent-continuation which means they have access to the agent state, environment and random-number stream. This allows testing to capture the *complete* system state in one location, which allows the checking of much more invariants than in approaches which have implicit side-effects.

We implement custom data-generators for the agents and let QuickCheck generate the random data and us running the agent with the provided data, checking for the properties. An example for such a property is that an agent has starved to death in case its sugar (or spice) level has dropped to 0. The corresponding property-test generates a random agent state and also a random sugar level which we set in the agent state. We then run the function which returns True in case the agent has starved to death. We can then check that this flag is true only if the initial random sugar level was less then or equal 0.

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: Agent-Interaction Property-Based testing using quickcheck-state-machine and [20] `http://www.well-typed.com/blog/2019/01/qsm-in-depth/`

9.5 Discussion

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.

Part IV:

Reflections

Chapter 10

Discussion

This chapter 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.

10.1 Benefits

10.2 Drawbacks

10.2.1 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

10.3 The Gintis Case

Discuss my developed techniques to the Gintis paper (and its follow ups: the Ionescu paper [10] and a Masterthesis [24] 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)

10.4 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.

10.4.1 Simulation in general

We already showed in the paper [78], 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 **??**.

10.4.2 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.

10.4.3 Discrete Event Simulation

pure functional DES easily possible with my developed synchronous messaging ABS DES in FP: we doing it in gintis study, 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

10.4.4 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 [26]. 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 [26], 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

10.4.5 Multi Agent Systems

The fields of Multi Agent Systems (MAS) and ABS are closely related where ABS has drawn much inspiration from MAS [92], [89]. 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.

10.5 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.

10.6 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.

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,

10.7 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* [59].

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.

Chapter 11

Conclusions

This chapter concludes the whole thesis and outlines future research. Roughly 20% exists already.

11.1 Further Research

1. generalise concepts explored into a pure functional ABS library in Haskell (called chimera), 2. dependent types and linear types are the next big step, towards a stronger formalisation of agents and ABS, 3. find an efficient algorithm for synchronous agent-interactions in concurrent STM ABS

References

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

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

[3] Armstrong, J. Erlang. *Commun. ACM 53*, 9 (Sept. 2010), 68–75.

[4] 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).

[5] 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.

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

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

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

[9] 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.

[10] 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.

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

[12] Brambilla, M., Pinciroli, C., Birattari, M., and Dorigo, M. Property-driven Design for Swarm Robotics. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 1* (Richland, SC, 2012), AAMAS '12, International Foundation for Autonomous Agents and Multiagent Systems, pp. 139–146.

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

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

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

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

[17] 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.

[18] 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.

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

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

[21] 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.

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

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

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

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

[26] 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.

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

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

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

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

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

[40] 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.

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

[42] 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.

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

[44] 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.

[45] 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.

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

[47] 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.

[48] 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.

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

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

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

[52] 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.

[53] 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.

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

[55] 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.

[56] Nguyen, C. D., Perini, A., Bernon, C., Pavón, J., and Thangarajah, J. Testing in Multi-agent Systems. In *Proceedings of the 10th International Conference on Agent-oriented Software Engineering* (Berlin, Heidelberg, 2011), AOSE'10, Springer-Verlag, pp. 180–190.

[57] 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.

[58] 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.

[59] 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.

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

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

[62] 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.

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

[64] 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.

[65] 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.

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

[67] 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.

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

[69] 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.

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

[71] 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.

[72] 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.

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

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

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

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

[77] 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.

[78] Thaler, J., Altenkirch, T., and Siebers, P.-O. Pure Functional Epidemics - An Agent-Based Approach. In *International Symposium on Implementation and Application of Functional Languages* (Lowell, Massachusettes, Aug. 2019).

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

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

[81] 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.

[82] 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.

[83] 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.

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

[85] 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.

[86] 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.

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

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

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

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

[91] 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.

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

[93] 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

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 [88] [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.

A.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.

A.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' [88] 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.

A.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.

A.4 Migration

With the information provided by [88] 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.

A.5 Pollution and Diffusion

With the information provided by [88] 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.

A.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 [88], 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 [88]. 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.

A.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.

A.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 [88].

A.9 Combat

Unfortunately [88] 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.

A.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 [88] 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 [88].

A.11 Trading

For trading we had a look at the NetLogo implementation of [88]: there an agent engages in trading with its neighbours *over multiple rounds* until either MRSs cross over or no trade has happened anymore. Because [88] 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.

A.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 [88] 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.