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

January 23, 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.

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 [29] 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* [67] 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 [43, 12] 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.

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 [90] - 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 [89] - 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 [54], 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 [25, 87, 50].

A multi-method simulation library in Haskell called *Aivika 3* is described in the technical report [85]. 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 [8] the author investigates Haskells' parallel and concurrency features to implement (amongst others) *HLogo*, a Haskell clone of the NetLogo [102] 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 [26, 94, 83] using the functional programming language Erlang [4] 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 [103]. 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 [8] 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 [54] discusses a framework, which allows to map Agent-Based Simulations to Graphics Processing Units (GPU). Amongst others they use the SugarScape model [29] 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 [50] 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 [81, 95]. 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 [24] 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 [38]. 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 [10] 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 [11] 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 [88], 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 [35] 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 [56], ABS vs. MAS, event- vs. time-driven [61], 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 [103]. 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 [84, 103, 56, 68]:

- 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 [28] 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) [103], [101]. 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 [56]: very good survey/review paper on ABMS in General,d sugarscape is level 3 (inzeractive), sir is level 2, level 4 ist auch kein Problem mit unserer Architektur. 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 [49] which tried to replicate the work of Gintis [34]. 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 [21] 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. [49] 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 [88], 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 [35]. 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: [96]

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 [22] 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 [66] 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 [5] discusses a similar approach to DES in the AnyLogic software toolkit.

The paper [70] 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 [18] 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 [37] 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 [64] 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 [19]. This is a fundamentally different approach to computing than imperative programming (includeing established object-orientation) which roots lie in the Turing Machine [92]. 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 [57] 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.

[57] 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 [57] 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 [69] 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 [48]. Other, more exhaustive books on learning Haskell are [53, 2]. For an introduction to programming with the Lambda-Calculus we refer to [62]. For more general discussion of functional programming we refer to [45, 57, 43].

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: [63, 97, 98, 99, 51].

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: [58, 71, 39, 59].

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.

[57] 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 [57] 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 [43] 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 [44, 43], is applicable to a number of real-world problems [71] 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 [46, 47] as implemented in the library Yampa [42, 23, 65].

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

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

Yampa provides a number of combinators for expressing time-semantics, events and state-changes of the system. They allow to change system behaviour in case of events, run signal functions and generate stochastic events and random-number streams. We shortly discuss the relevant combinators and concepts we use throughout the paper. For a more in-depth discussion we refer to [42, 23, 65].

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

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

Randomness In ABS, often there is the need to generate stochastic events, which occur based on e.g. an exponential distribution. Yampa provides the combinator *occasionally :: RandomGen g ⇒ g → Time → b → SF a (Event b)* for this. It takes a random-number generator, a rate and a value the stochastic event will carry. It generates events on average with the given rate. Note that at most one event will be generated and no 'backlog' is kept. This means that

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

Yampa also provides the combinator *noise :: (RandomGen g, Random b)* $\Rightarrow$ $g \rightarrow$ *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 [46, 47].

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 [72], 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 [76].

```
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 [72] 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 [75, 74]:

```
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.

3.3 Dependent Types

Dependent types are a very powerful addition to functional programming as they allow us to express even stronger guarantees about the correctness of programs *already at compile-time*. They go as far as allowing to formulate programs and types as constructive proofs which must be *total* by definition [91, 60, 3].

So far no research using dependent types in agent-based simulation exists at all. We have already started to explore this for the first time and ask more specifically how we can add dependent types to our functional approach, which conceptual implications this has for ABS and what we gain from doing so. We are using Idris [12] as the language of choice as it is very close to Haskell with focus on real-world application and running programs as opposed to other languages with dependent types e.g. Agda and Coq which serve primarily as proof assistants.

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

In the following section **??**, we give an introduction of the concepts behind dependent types and what they can do. Further we give a very brief overview of the foundational and philosophical concepts behind dependent types. In Section **??** we briefly discuss ideas of how the concepts of dependent types could be

applied to agent-based simulation and in Section 3.3.4 we very shortly discuss the connection between Verification & Validation and dependent types.

There exist a number of excellent introduction to dependent types which we use as main ressources for this section: [91, 79, 86, 15, 77].

Generally, dependent types add the following concepts to pure functional programming:

1. Types are first-class citizen - In dependently types languages, types can depend on any *values*, and can be *computed* at compile-time which makes them first-class citizen. This becomes apparent in Section 3.3.1 where we compute the return type of a function depending on its input values.

2. Totality and termination - A total function is defined in [15] as: it terminates with a well-typed result or produces a non-empty finite prefix of a well-typed infinite result in finite time. This makes run-time overhead obsolete, as one does not need to drag around additional type-information as everything can be resolved at compile-time. Idris is turing-complete but is able to check the totality of a function under some circumstances but not in general as it would imply that it can solve the halting problem. Other dependently typed languages like Agda or Coq restrict recursion to ensure totality of all their functions - this makes them non turing-complete. All functions in Section 3.3.1 are total, they terminate under all inputs in finite steps.

3. Types as *constructive* proofs - Because types can depend on any values and can be computed at compile-time, they can be used as constructive proofs (see 3.3.3) which must terminate, this means a well-typed program (which is itself a proof) is always terminating which in turn means that it must consist out of total functions. Note that Idris does not restrict us to total functions but we can enforce it through compiler flags. We implement a constructive proof of showing whether two natural numbers are decidable equal in the Section 3.3.2.

3.3.1 An example: Vector

To give a concrete example of dependent types and their concepts, we introduce the canonical example used in all tutorials on dependent types: the Vector.

In all programming languages like Haskell or in Java, there exists a List data-structure which holds a finite number of homogeneous elements, where the type of the elements can be fixed at compile-time. Using dependent types we can implement the same but adding the length of the list to the type - we call this data-structure a vector.

We define the vector as a Generalised Algebraic Data Type (GADT). A vector has a *Nil* element which marks the end of a vector and a *(::)* which is a recursive (inductive) definition of a linked List. We defined some vectors and we see that the length of the vector is directly encoded in its first type-variable of type Nat, natural numbers. Note that the compiler will refuse to

accept *testVectFail* because the type specifies that it holds 2 elements but the constructed vector only has 1 element.

```
data Vect : Nat -> Type -> Type where
     Nil  : Vect Z e
     (::) : (elem : e) -> (xs : Vect n e) -> Vect (S n) e

testVect : Vect 3 String
testVect = "Jonathan" :: "Andreas" :: "Thaler" :: Nil

testVectFail : Vect 2 Nat
testVectFail = 42 :: Nil
```

We can now go on and implement a function *append* which simply appends two vectors. Here we directly see *type-level computations* as we compute the length of the resulting vector. Also this function is *total*, as it covers all input cases and recurs on a *structurally smaller argument*:

```
append : Vect n e -> Vect m e -> Vect (n + m) e
append Nil ys = ys
append (x :: xs) ys = x :: append xs ys

append testVect testVect
["Jonathan", "Andreas", "Thaler", "Jonathan", "Andreas", "Thaler"] : Vect 8 String
```

What if we want to implement a *filter* function, which, depending on a given predicate, returns a new vector which holds only the elements for which the predicates returns true? How can we compute the length of the vector at compile-time? In short: we can't, but we can make us of *dependent pairs* where the *type* of the second element depends on the *value* of the first (dependent pairs are also known as Σ types).

The function is total as well and works very similar to *append* but uses dependent types as return, which are indicated by $**$:

```
filter : Vect n e -> (e -> Bool) -> (k ** Vect k e)
filter [] f = (Z ** Nil)
filter (elem :: xs) f =
  case f elem of
    False => filter xs f
    True  => let (_ ** xs') = filter xs f
             in  (_ ** elem :: xs')

filter testVect (=="Jonathan")
(1 ** ["Jonathan"]) : (k : Nat ** Vect k String)
```

It might seem that writing a *reverse* function for a Vector is very easy, and we might give it a go by writing:

```
reverse : Vect n e -> Vect n e
reverse [] = []
reverse (elem :: xs) = append (reverse xs) [elem]
```

Unfortunately the compiler complains because it cannot unify 'Vect (n + 1) e' and 'Vect (S n) e'. In the end, the compiler tells us that it cannot determine

that (n + 1) is the same as (1 + n). The compiler does not know anything about the commutativity of addition which is due to how natural numbers and their addition are defined.

Lets take a detour. The natural numbers can be inductively defined by their initial element zero Z and the successor. The number 3 is then defined as the successor of successor of successor of zero:

```
data Nat = Z | S Nat

three : Nat
three = S (S (S Z))
```

Defining addition over the natural numbers is quite easy by pattern-matching over the first argument:

```
plus : (n, m : Nat) -> Nat
plus Z right        = right
plus (S left) right = S (plus left right)
```

Now we can see why the compiler cannot infer that (n + 1) is the same as (1 + n). The expression (n + 1) is translated to (plus n 1), where we pattern-match over the first argument, so we cannot reach a case in which (plus n 1) = S n. To do that we would need to define a different plus function which pattern-matches over the second argument - which is clearly the wrong way to go.

To solve this problem we can exploit the fact that dependent types allow us to perform type-level computations. This should allow us to express commutativity of addition over the natural numbers as a type. For that we define a function which takes in two natural numbers and returns a proof that addition commutes.

```
plusCommutative : (left : Nat) -> (right : Nat) -> left + right = right + left
```

We now begin to understand what it means when we speak of *types as proofs*: we can actually express e.g. laws of the natural numbers in types and proof them by implementing a program which inhibits the type - we speak then of a constructive proof (see more on that below 3.3.3). Note that *plusCommutative* is already implemented in Idris and we omit the actual implementation as it is beyond the scope of this introduction

Having our proof of commutativity of natural numbers, we can now implement a working (speak: correct) version of *reverse*. The function *rewrite* is provided by Idris: if we have a proof for x = y, the 'rewrite expr in' syntax will search for x in the required type of expr and replace it with y:

```
reverse : Vect n e -> Vect n e
reverse [] = []
reverse (elem :: xs) = reverseProof (append (reverse xs) [elem])
  where
    reverseProof : Vect (k + 1) a -> Vect (S k) a
    reverseProof {k} result = rewrite plusCommutative 1 k in result
```

3.3.2 Equality as type

On of the most powerful aspects of dependent types is that they allow us to express equality on an unprecedented level. Non-dependently typed languages have only very basic ways of expressing the equality of two elements of same type. Either we use a boolean or another data-structure which can indicate equality or not. Idris supports this type of equality as well through *(==) : Eq ty ⇒ ty → ty → Bool*. The drawback of using a boolean is that, in the end, we don't have a real evidence of equality: it doesn't tell you anything about the relationship between the inputs and the output. Even though the elements might be equal, the compiler has no means of inferring this and we can still make programming mistakes after the equality check because of this lack of compiler support. Even worse, always returning False / True or whether the inputs are *not* equal is a valid implementation of (==), at least as far as the type is concerned.

As an illustrating example we want to write a function which checks if a Vector has a given length.

```
exactLength : (len : Nat) -> (input : Vect n k) -> Maybe (Vect len k)
exactLength {n} len input = case n == len of
                                 True  => Just input
                                 False => Nothing
```

Unfortunately this doesn't type-check ('type mismatch between n and len') because the compiler has no way of determining that *len* is equals *n* at compile-time. Fortunately we can solve this problem using dependent types themselves by defining *decidable* equality as a type.

First we need a decidable property, meaning it either holds given with some *proof* or it does not hold given some proof that it does *not* hold, resulting in a contradiction. Idris defines such a decidable property already as the following:

```
-- Decidability. A decidable property either holds or is a contradiction.
data Dec : Type -> Type where
  -- The case where the property holds
  -- @ prf the proof
  Yes : (prf : prop) -> Dec prop

  -- The case where the property holding would be a contradiction
  -- @ contra a demonstration that prop would be a contradiction
  No  : (contra : prop -> Void) -> Dec prop
```

With that we can implement a function which constructs a proof that two natural numbers are equal, or not. We do this simply by pattern matching over both numbers with corresponding base cases and inductions. In case they are not equal we need to construct a proof that they are actually not equal which is done by showing that given some property results in a contradiction - indicated by the type *Void*. In case of *zeroNotSuc* the first number is zero (Z) whereas the other one is non-zero (a successor of some k), which can never be equal, thus we return a *No* instance of the decidable property for which we need to provide the contradiction. In case of *sucNotZero* its just the other way around.

noRec works very similar but here we are in the induction case which says that if k equals j leads to a contradiction, (k + 1) and (j + 1) can't be equal as well (induction hypothesis).

```
checkEqNat : (num1 : Nat) -> (num2 : Nat) -> Dec (num1 = num2)
checkEqNat Z Z         = Yes Refl
checkEqNat Z (S k)     = No zeroNotSuc
checkEqNat (S k) Z     = No sucNotZero
checkEqNat (S k) (S j) = case checkEqNat k j of
                              Yes prf   => Yes (cong prf)
                              No contra => No (noRec contra)

zeroNotSuc : (0 = S k) -> Void
zeroNotSuc Refl impossible

sucNotZero : (S k = 0) -> Void
sucNotZero Refl impossible

noRec : (contra : (k = j) -> Void) -> (S k = S j) -> Void
noRec contra Refl = contra Refl
```

The important thing to understand here is that our Dec property holds much more information than just a boolean flag which indicates whether Yes/No that two elements of a type are equal: in case of Yes we have a type which says that num1 is equal to num2, which can be directly used by the compiler, both elements are treated as the same. Refl stands for reflexive and is built into Idris syntax, meaning that a value is equal to itself 'Refl : x = x'.

Finally we can implement a correct version of our initial *exactLength* function by computing a proof of equality between both lengths at run-time using *checkEqNat*. This proof can then be used by the compiler to infer that the lengths are indeed equal or not.

```
exactLength : (len : Nat) -> (input : Vect n k) -> Maybe (Vect len k)
exactLength {n} len input = case checkEqNat n len of
                                -- len vanishes as compiler can unify len to n
                                Yes Refl  => Just input
                                No contra => Nothing
```

3.3.2.1 Kinds of Equality

In type theory there are different kinds of equality [4], which in turn depend on the flavour of type theory which can be either *intensional* or *extensional*:

1. Definitional or intensional equality: the symbols '2' and 'S(S(Z))' are said to be definitional / intensionally equal terms, because their *intended meaning* is the same.

[4] We follow in these definitions mainly `https://ncatlab.org/nlab/show/equality`, `https://ncatlab.org/nlab/show/intensional+type+theory` and `https://ncatlab.org/nlab/show/extensional+type+theory`.

2. Computational or judgmental equality: two terms '2 + 2' and '4' are said to be computationally equal because when the result of the addition is computed by a program then they will reduce to the same term 'S(S(Z)) + S(S(Z))' to 'S(S(S(S(Z))))'. In intensional type theory this kind of equality is treated as definitional equality, thus '2 + 2' and '4' are equal by definition.

3. Propositional equality: when one wants to define general rules that e.g. 'a+b' and 'b+a' are equal, we are talking about a theorem, not a definition. Computational / definitional equality does not work here as to compute it one needs to substitute a and b for concrete natural numbers. In this case we are talking about extensional equality, which is a judgement, not a proposition and thus *not* internal to the formal system itself. It can be internalized through *propositional* equality by adding an identity type which allows to express '2+2 = 4' as a *type*. If such an expression (speak: proof) holds, then this type is inhabited, if not e.g. in the case of '2+2 = 5', this type holds no element and thus no proof exists for it (see section 3.3.3).

Still it is not very clear what *intensional* and *extensional* type theory means. The HOTT Book [79] says the following in Chapter 1: "Extensional theory makes no distinction between judgmental and propositional equality, the intensional theory regards judgmental equality as purely definitional, and admits a much broader proof-relevant interpretation of the identity type...". This means, that extensional type theory treats objects to be equal if they have the same external properties. In this type of theory, two functions are equal if they give the same results on every input (extensional equality on the function space). Intensional type theory on the other hand allows to distinguish between internal definitions of objects. In this type of theory, two functions are equal if their (internal) definitions are the same.

Applied to our examples this means the following: We have definitional equality through (==) and *Eq*. Propositional equality is exactly what we got when we introduced the identity type above in the *checkEqNat* function with *Dec (num1 = num2)*. The (=) in the type is built-in into Idris and defines the propositional equality. Dhe Dec type is required to indicate that the proposition may or may not be inhabited. Thus we can also follow that Idris is intensional (and so is Agda and Coq).

3.3.3 Philosophical Foundations: Constructivism

The main theoretical and philosophical underpinnings of dependent types as in Idris are the works of Martin-Löf intuitionistic type theory. The view of dependently typed programs to be proofs is rooted in a deep philosophical discussion on the foundations of mathematics, which revolve around the existence of mathematical objects, with two conflicting positions known as classic vs. con-

structive [5]. In general, the constructive position has been identified with realism and empirical computational content where the classical one with idealism and pragmatism.

In the classical view, the position is that to prove $\exists x.P(x)$ it is sufficient to prove that $\forall x.\neg P(x)$ leads to a contradiction. The constructive view would claim that only the contradiction is established but that a proof of existence has to supply an evidence of an x and show that $P(x)$ is provable. In the end this boils down whether to use proof by contradiction or not, which is sanctioned by the law of the excluded middle which says that $A \vee \neg A$ must hold. The classic position accepts that it does and such proofs of existential statements as above, which follow directly out of the law of the excluded middle, abound in mathematics [6]. The constructive view rejects the law of the excluded middle and thus the position that every statement is seen as true or false, independently of any evidence either way. [91] (p. 61): *The constructive view of logic concentrates on what it means to prove or to demonstrate convincingly the validity of a statement, rather than concentrating on the abstract truth conditions which constitute the semantic foundation of classical logic.*

To prove a conjunction $A \wedge B$ we need prove both A and B, to prove $A \vee B$ we need to prove one of A, B and know which we have proved. This shows that the law of the excluded middle can not hold in a constructive approach because we have no means of going from a proof to its negation. Implication $A \Rightarrow B$ in constructive position is a transformation of a proof A into a proof B: it is a function which transforms proofs of A into proofs of B. The constructive approach also forces us to rethink negation, which is now an implication from some proof to an absurd proposition (bottom): $A \Rightarrow \bot$. Thus a negated formula has no computational context and the classical tautology $\neg\neg A \Rightarrow A$ is then obviously no longer valid. Constructively solving this would require us to be able to effectively compute / decide whether a proposition is true or false - which amounts to solving the halting problem, which is not possible in the general case.

A very important concept in constructivism is that of finitary representation / description. Objects which are infinite e.g. infinite sets as in classic mathematics, fail to have computational computation, they are not computable. This leads to a fundamental tenet in constructive mathematics: [91] (p. 62): *Every object in constructive mathematics is either finite [..] or has a finitary description*

Concluding, we can say that constructive mathematics is based on principles quite different from classical mathematics, with the idealistic aspects of the latter replaced by a finitary system with computational content. Objects like functions are given by rules, and the validity of an assertion is guaranteed by a proof from which we can extract relevant computational information, rather than on idealist semantic principles.

All this is directly reflected in dependently typed programs as we introduced above: functions need to be total (finitary) and produce proofs like in *check-*

[5] We follow the excellent introduction on constructive mathematics [91], chapter 3.

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

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

3.3.4 Verification, Validation and Dependent Types

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

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

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

With pure functional programming and dependent types the process is now mostly constructive, type-driven (see [15]). In that approach one defines types first and is then guided by these types and the compiler in an interactive fashion towards a correct implementation, ensured at compile-time. As already noted, the ABS methodology is constructive in nature but the established object-oriented test-driven implementation approach not as much, creating an impedance mismatch. We expect that a type-driven approach using dependent types reduces that mismatch by a substantial amount.

Note that *validation* is a different matter here: independent of our implementation approach we still need to validate the simulation against the real-world / ground-truth. This obviously requires to run the full simulation which could take up hours in either programming paradigm, making them absolutely equal

in this respect. Also the comparison of the output to the real-world / ground-truth is completely independent to the paradigm. The fundamental difference happens in case of changes made to the code during validation: in case of the established test-driven object-oriented approach for every minor change one (should) re-run all tests, which could take up a substantial amount of additional time. Using a constructive, type-driven approach this is dramatically reduced and can often be completely omitted because the correctness of the change can be either guaranteed in the type or by informally reasoning about the code.

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 [61]. 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 [55, 89]. 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 [61].

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 [90], 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.

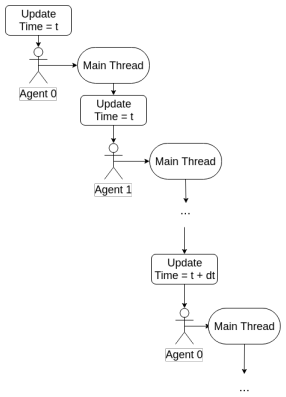


Figure 4.1: Control flow in the Sequential Strategy.

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.

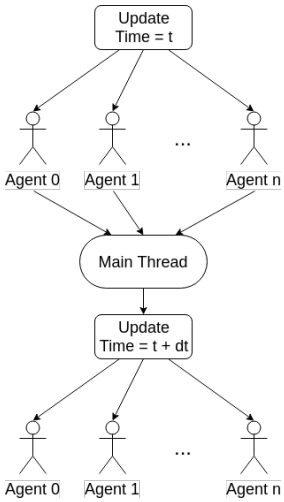


Figure 4.2: Control flow in the Parallel Strategy.

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.

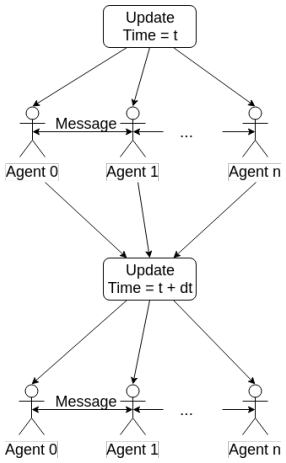


Figure 4.3: Control flow in the Concurrent Strategy.

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 [41] for which [36] and [20] 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.

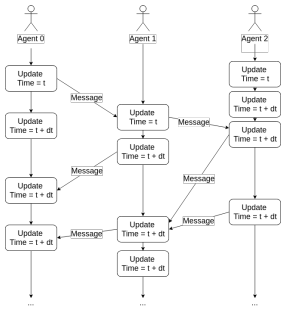


Figure 4.4: Control flow in the Actor Strategy.

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 established approaches to implementing 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 [66] 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 sections 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 the following, we derive a pure functional approach for an agent-based SIR model in which we pose solutions to the previously mentioned problems. 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, 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 [52], 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.

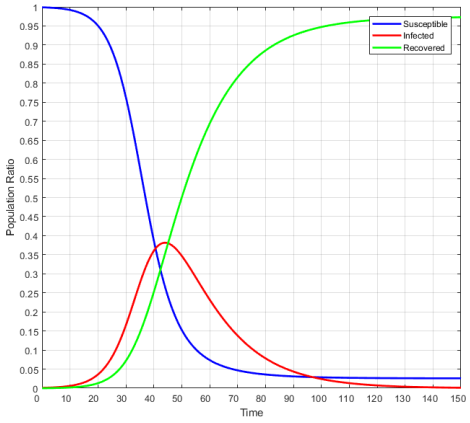


Figure 5.2: Dynamics of the SIR compartment model using the System Dynamics approach. Population Size $N = 1{,}000$, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time-steps. Generated using our pure functional SD approach (see Chapter 12.3.2).

This model was also formalized using System Dynamics (SD) [78]. 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 [104].

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 [9]. 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.

TODO: follow the PFE paper but add a few references back to the implementing ABS chapter so it is clear what problems we are currently solving. also might explain some concepts bit more in depth e.g. continuations?

5.2 First step: pure computation

As described in Chapter 3.2.4, Arrowized FRP [46] 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 [9] 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
```

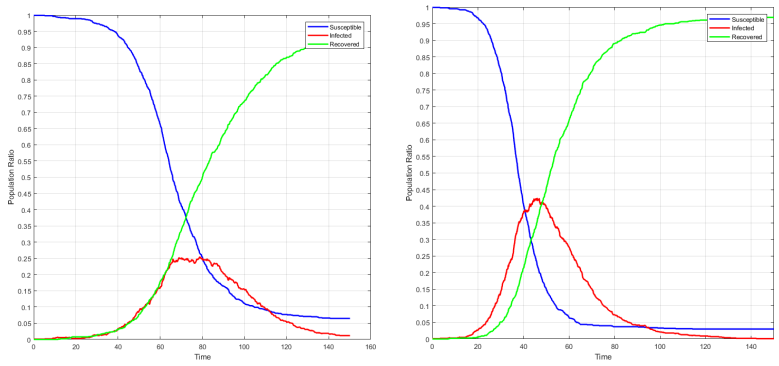
What we need to implement next is a closed feedback-loop - the heart of every agent-based simulation. Fortunately, [65, 23] discusses implementing this in Yampa. The function *stepSimulation* is an implementation of such a closed feedback-loop. It takes the current signal functions and states of all 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)
```

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



(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.

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 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 two options. The first one is to use a smaller

(a) Under-sampling

(b) Super-sampling

Figure 5.4: A visual explanation of under-sampling and super-sampling. The black dots represent the time-steps of the simulation. The red dots represent virtual events which occur at specific points in continuous time. In the case of under-sampling, 3 events occur in between the two time steps but *occasionally* only captures the first one. By increasing the sampling frequency either through a smaller Δt or super-sampling all 3 events can be captured.

Δt as can be seen in 5.3b, which results in the whole system being sampled more often, thus reducing performance. The other option is to implement super-sampling and apply it to *occasionally*, which would allow us to run the whole simulation with $\Delta t = 1.0$ and only sample the *occasionally* function with a much higher frequency.

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

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

    superSamplingAux sf' = SF' tf
      where
        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
```

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 [75], 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 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 [29] 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.

(a) von Neumann (b) Moore

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

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 [66], 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
```

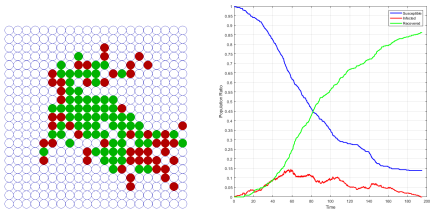
5.4.2 Results

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.

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.



(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 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 [76]. Also by using FRP we gain all the benefits from it and can use research on testing, debugging and exploring FRP systems [76, 73].

Also we showed how to implement the *parallel* update-strategy [90] 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 [66]. 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 [61] 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 [39, 40], 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 [82] 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.

5.5.1 Other approaches

Sequential: see event-driven Concurrent: see STM chapter Actor: not discussed in this thesis but briefly

5.5.2 Super-Sampling

Chapter 6

Event-Driven

Due to the necessity of imposing a correct ordering of events in this type of ABS, we are only left with one way to step such an implementation: sequentially, event by event. In this case the ABS is technically quite close to a DES. Note that there exists also Parallel DES (PDES) [32], which deals with processing events in parallel and dealing with inconsistencies in ordering in different ways - we leave this in the context for pure functional ABS for further research.

6.1 Sugarscape

6.2 Synchronised Agent-Interactions

Following towards papers SugarScape implementation

6.3 Discussion

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

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

Chapter 7

Generalising

TODO: can we derive an agent-monad?

TODO: what about comonads? read essence dataflow paper [93]: 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.

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.

About 50% finished.

8.1 Data-Parallelism

Discusses 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.

8.2 Software Transactional Memory

This is a shorter recap of the STM paper, 100% finished.

Chapter 9

Verification

Exploring ways in which pure functional ABS can be of benefit to verification & validation and increasing correctness of an ABS implementation.

General there are the following basic verification & validation requirements to ABS [80], 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.

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

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.

9.1 Debugging

TODO: haskell-titan TODO: Testing and Debugging Functional Reactive Programming [76]

9.2 Using the Type-System

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

9.3 Reasoning

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

9.4 Unit-Testing

Follow unit-testing of the whole simulation as prototyped for towards paper.

9.5 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.

9.5.1 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 [55]. 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.

9.5.2 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.

Chapter 10

Dependent Types

The pure functional implementation techniques have a number of technical benefits but don't help as much in closing the gap between specification and implementation as one is used from functional programming in general. Therefore we take a step back and abstract from these highly complex implementation techniques and move towards dependent types. Follow [10] and [11].

Conceptually discuss how dependent types can be made of use in ABS without going into lot of technical detail because: 1. i didn't do enough research on it and 2. dependent types seem to be nearly out of focus of the thesis.

After having established the concepts of dependent types, we want to briefly discuss ideas where and how they could be made of use in ABS. We expect that dependent types will help ruling out even more classes of bugs at compile-time and encode even more invariants. Additionally by constructively implementing model specifications on the type level could allow the ABS community to reason about a model directly in code as it narrows the gap between model specification and implementation.

By definition, ABS is of constructive nature, as described by Epstein [27]: "If you can't grow it, you can't explain it" - thus an agent-based model and the simulated dynamics of it is itself a constructive proof which explain a real-world phenomenon sufficiently well. Although Epstein certainly wasn't talking about a constructive proof in any mathematical sense in this context (he was using the word *generative*), dependent types *might* be a perfect match and correspondence between the constructive nature of ABS and programs as proofs.

When we talk about dependently typed programs to be proofs, then we also must attribute the same to dependently typed agent-based simulations, which are then constructive proofs as well. The question is then: a constructive proof of what? It is not entirely clear *what we are proving* when we are constructing dependently typed agent-based simulations. Probably the answer might be that a dependently typed agent-based simulation is then indeed a constructive proof in a mathematical sense, explaining a real-world phenomenon sufficiently well - we have closed the gap between a rather informal constructivism as mentioned above when citing Epstein who certainly didn't mean it in a constructive

mathematical sense, and a formal constructivism, made possible by the use of dependent types.

In the following subsections we will discuss related work in this field (**??**), discuss general concepts where dependent types might be of benefit in ABS (**??**), present a dependently typed implementation of a 2D discrete environment (**??**) and finally discuss potential use of dependent types in the SIR model (**??**) and SugarScape model (**??**).

10.1 Related Work

In [11] the authors 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.

The paper [16] discusses depend types to implement correct-by-construction concurrency in the Idris language [12]. The authors introduce the concept of a Embedded Domain Specific Language (EDSL) for concurrently locking/unlocking and reading/writing of resources and show that an implementation and formalisation are the same thing when using dependent types. We can draw inspiration from it by taking into consideration that we might develop an EDSL in a similar fashion for specifying general commands which agents can execute. The interpreter of such a EDSL can be pure itself and doesn't have to run in the IO Monad as our previous research (see Appendix **??**) has shown that ABS can be implemented pure.

In [17] the authors discuss systems programming with focus on network packet parsing with full dependent types in the Idris language [12]. Although they use an older version of it where a few features are now deprecated, they follow the same approach as in the previous paper of constructing an EDSL and writing an interpreter for the EDSL. In a longer introduction of Idris the authors discuss its ability for termination checking in case that recursive calls have an argument which is structurally smaller than the input argument in the same position and that these arguments belong to a strictly positive data type. We are particularly interested in whether we can implement an agent-based simulation which termination can be checked at compile-time - it is total.

In [13] the author discusses programming and reasoning with algebraic effects and dependent types in the Idris language [12]. They claim that monads do not compose very well as monad transformer can quickly become unwieldy when there are lots of effects to manage. As a remedy they propose algebraic effects [7] and implement them in Idris and show how dependent types can be used to reason about states in effectful programs. In our previous research (see

Appendix **??**) we relied heavily on Monads and transformer stacks and we indeed also experienced the difficulty when using them. Algebraic effects might be a promising alternative for handling state as the global environment in which the agents live or threading of random-numbers through the simulation which is of fundamental importance in ABS. According to the authors of the paper, unfortunately, algebraic effects cannot express continuations which is but of fundamental importance for pure functional ABS as agents are on the lowest level built on continuations - synchronous agent interactions and time-stepping builds directly on continuations. Thus we need to find a different representation of agents - GADTs seem to be a natural choice as all examples build heavily on them and they are very flexible.

In [31] the authors apply dependent types to achieve safe and secure web programming. This paper shows how to implement dependent effects, which we might draw inspiration from of how to implement agent-interactions which, depending on their kind, are effectful e.g. agent-transactions or events.

In [14] the author introduces the ST library in Idris, which allows a new way of implementing dependently typed state machines and compose them vertically (implementing a state machine in terms of others) and horizontally (using multiple state machines within a function). In addition this approach allows to manage stateful resources e.g. create new ones, delete existing ones. We can draw further inspiration from that approach on how to implement dependently typed state machines, especially composing them hierarchically, which is a common use case in agent-based models where agents behaviour is modelled through hierarchical state-machines. As with the Algebraic Effects, this approach doesn't support continuations, so it is not really an option to build our architecture for our agents on it, but it may be used internally to implement agents or other parts of the system. What we definitely can draw inspiration from is the implementation of the indexed Monad *STrans* which is the main building block for the ST library.

The book [15] is a great source to learn pure functional dependently typed programming and in the advanced chapters introduces the fundamental concepts of dependent state machine and dependently typed concurrent programming on a simpler level than the papers above. One chapter discusses on how to implement a messaging protocol for concurrent programming, something we can draw inspiration from for implementing our synchronous agent interaction protocols.

In [82] the authors apply dependent types to FRP to avoid some run-time errors and implement a dependently typed version of the Yampa library in Agda.

The fundamental difference to all these real-world examples is that in our approach, the system evolves over time and agents act over time in a feedback loop. A fundamental question will be how we encode the monotonous increasing flow of time in types and how we can reflect in the types that agents act over time.

10.2 General Concepts

We came up with the following ideas of how and where to apply dependent types in the context of agent-based simulation:

Environment Access Accessing e.g. discrete 2D environments involves (almost always) indexed array access which is always potentially dangerous as the indices have to be checked at run-time.

Using dependent types it should be possible to encode the environment dimensions into the types. In combination with suitable data types (finite sets) for coordinates one should be able to ensure already at compile-time that access happens only within the bounds of the environment. We have implemented this already and describe it in detail in the section **??**.

State-Machines Often, Agent-Based Models define their agents in terms of state-machines. It is easy to make wrong state-transitions e.g. in the SIR model when an infected agent should recover, nothing prevents one from making the transition back to susceptible.

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

Flow Of Time State-Machines often have timed transitions e.g. in the SIR model, an infected agent recovers after a given time. Nothing prevents us from introducing a bug and *never* doing the transition at all.

With dependent types we might be able to encode the passing of time in the types and guarantee on a type level that an infected agent has to recover after a finite number of time steps. Also can dependent types be used to express the flow of time and that it is strongly monotonic increasing?

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

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

Agent-Agent Interactions Because we are lacking method-calls as in object-oriented programming, we need to come up with different mechanics for agent-agent interaction, which are basically based upon continuations. The main use-case are multi-step interactions which happen without a time-delay e.g trading

or resource exchange protocols as described in SugarScape. In these two agents interact over multiple steps, following a given protocol, which is a source of bugs when not following the required steps.

Using dependent types we might be able to encode a protocol for agent-agent interactions which e.g. ensures on the type-level that an agent has to reply to a request or that a more specific protocol has to be followed e.g. in auction- or trading-simulations.

Equilibrium and Totality For some agent-based simulations there exists equilibria, which means that from that point the dynamics won't change any more e.g. when a given type of agents vanishes from the simulation or resources are consumed. This means that at that point the dynamics won't change any more, thus one can safely terminate the simulation. Very often, despite such a global termination criterion exists, such simulations are stepped for a fixed number of time-steps or events or the termination criterion is checked at runtime in the feedback-loop.

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

A central question in tackling this is whether to follow a model- or an agent-centric approach. The former one looks at the model and its specifications as a whole and encodes them e.g. one tries to directly find a total implementation of an agent-based model. The latter one looks only at the agent level and encodes that as dependently typed as possible and hopes that model guarantees emerge on a meta-level - put otherwise: does the totality of an implementation emerge when we follow an agent-centric approach?

Specifications and properties Using dependent types it is possible to encode model specifications and properties directly in types as described above. Other examples are to guarantee that the number of agent stays constant.

Hypotheses Models which are exploratory in nature don't have a formal ground truth where one could derive equilibria or dynamics from and validate with. In such models the researchers work with informal hypotheses which they express before running the model and then compare them informally against the resulting dynamics.

It would be of interest if dependent types could be made of use in encoding hypotheses on a more constructive and formal level directly into the implementation code. So far we have no idea how this could be done but it might be a very interesting application as it allows for a more formal and automatic testable approach to hypothesis checking.

10.3 Dependently Typed Discrete 2D Environment

One of the main advantages of Agent-Based Simulation over other simulation methods e.g. System Dynamics is that agents can live within an environment. Many agent-based models place their agents within a 2D discrete NxM environment where agents either stay always on the same cell or can move freely within the environment where a cell has 0, 1 or many occupants. Ultimately this boils down to accessing a NxM matrix represented by arrays or a similar data structure. In imperative languages accessing memory always implies the danger of out-of-bounds exceptions *at run-time*. With dependent types we can represent such a 2D environment using vectors which carry their length in the type (see **??**) thus fixing the dimensions of such a 2D discrete environment in the types. This means that there is no need to drag those bounds around explicitly as data. Also by using dependent types like a finite set Fin, which depend on the dimensions we can enforce at compile-time that we can only access the data structure within bounds. If we want to we can also enforce in the types that the environment will never be an empty one where $N, M > 0$.

```
-- an environment has width w and height h and cells e and is never empty
-- adding Successor S to each dimension ensures that the environment is not empty
Disc2dEnv : (w : Nat) -> (h : Nat) -> (e : Type) -> Type
Disc2dEnv w h e = Vect (S w) (Vect (S h) e)

-- the coordinates for an environment are respresented by the (Fin k) datatype
-- which represents the natural numbers as a finite set from  0..k
-- need an additional S for ensuring that our bounds are strictly less than
data Disc2dCoords : (w : Nat) -> (h : Nat) -> Type where
  MkDisc2dCoords : Fin (S w) -> Fin (S h) -> Disc2dCoords w h

centreCoords : Disc2dEnv w h e -> Disc2dCoords w h
centreCoords {w} {h} _ =
    let x = halfNatToFin w
        y = halfNatToFin h
    in  mkDisc2dCoords x y
  where
    halfNatToFin : (x : Nat) -> Fin (S x)
    halfNatToFin x =
      let xh   = divNatNZ x 2 SIsNotZ
          mfin = natToFin xh (S x)
      in  fromMaybe FZ mfin

-- overriding the content of a cell: no boundary checks necessary
setCell :  Disc2dCoords w h
        -> (elem : e)
        -> Disc2dEnv w h e
        -> Disc2dEnv w h e
setCell (MkDisc2dCoords colIdx rowIdx) elem env
    = updateAt colIdx (\col => updateAt rowIdx (const elem) col) env

-- reading the content of a cell: no boundary checks necessary
getCell :  Disc2dCoords w h
```

```
        -> Disc2dEnv w h e
        -> e
getCell (MkDisc2dCoords colIdx rowIdx) env
    = index rowIdx (index colIdx env)

neumann : Vect 4 (Integer, Integer)
neumann = [         (0,  1),
           (-1,  0),         (1,  0),
                    (0, -1)]

moore : Vect 8 (Integer, Integer)
moore = [(-1,  1), (0,  1), (1,  1),
         (-1,  0),          (1,  0),
         (-1, -1), (0, -1), (1, -1)]

filterNeighbourhood :  Disc2dCoords w h
                    -> Vect len (Integer, Integer)
                    -> Disc2dEnv w h e
                    -> (n ** Vect n (Disc2dCoords w h, e))
filterNeighbourhood {w} {h} (MkDisc2dCoords x y) ns env =
    let xi = finToInteger x
        yi = finToInteger y
    in  filterNeighbourhood' xi yi ns env
  where
    filterNeighbourhood' :  (xi : Integer)
                         -> (yi : Integer)
                         -> Vect len (Integer, Integer)
                         -> Disc2dEnv w h e
                         -> (n ** Vect n (Disc2dCoords w h, e))
    filterNeighbourhood' _ _ [] env = (0 ** [])
    filterNeighbourhood' xi yi ((xDelta, yDelta) :: cs) env
      = let xd = xi - xDelta
            yd = yi - yDelta
            mx = integerToFin xd (S w)
            my = integerToFin yd (S h)
        in case mx of
            Nothing => filterNeighbourhood' xi yi cs env
            Just x  => (case my of
                        Nothing => filterNeighbourhood' xi yi cs env
                        Just y  => let coord      = MkDisc2dCoords x y
                                       c          = getCell coord env
                                       (_ ** ret) = filterNeighbourhood' xi yi cs env
                                   in  (_ ** ((coord, c) :: ret)))
```

10.4 Dependently Typed SIR

We plan to prototype the concepts of section **??** in a dependently typed SIR implementation. One can object that the SIR model [52] is a very simple model but despite its simplicity it has a number of advantages. There is a theory behind it with a formal ground-truth for the dynamics which can be generated by differential equations, which allows validation of the simulation. Also, it has already many concepts of ABS in it without making it too complex: agent-behaviour as a state-machine, local agent-state (current SIR state and duration of illness), feedback, very rudimentary interaction with other agents, 2D environment if

required and behaviour over time. We will also look into the SugarScape model (see **??**), which is of quite a different type and adds more complexity.

The general approach of using dependent types is to specify the general commands available for an agent, where we can follow the approach of an EDSL as described in [16] and write then an interpreter for it. It is of importance that the interpreter shall be pure itself and does not make use of any IO. Applying dependent types to the SIR model, we came up with the following use-cases:

Environment access We have already introduced an implementation for a dependently typed 2D environment in section **??**. This can be directly used to implement a SIR on a 2D environment as we have done in the paper in Appendix **??**.

State-Machine and Flow Of Time The transition through the Susceptible, Infected and Recovered states are a state-machine, thus we want to apply dependent types to restrict the valid transitions and ensure that they are enforced under the given circumstances. The transitions are restricted to: Susceptibles can only transition to Infected, Infected only to Recovered and Recovered stay in that state forever. A transition from Susceptible to Infected happens with a given probability in case the Susceptible makes contact with an Infected. The transition from Infected to Recover happens after a given number of time-steps.

The tricky thing is that all these transitions ultimately depend on stochastic events: Susceptible pick their contacts at random, uniformly distributed from all agents in the simulation, they get infected with a probability when the contact is Infected and the duration an Infected agent is ill is picked from an exponential distribution.

Equilibrium and totality The idea is to implement a total agent-based SIR simulation, where the termination does NOT depend on time (is not terminated after a finite number of time-steps, which would be trivial). We argue that the underlying SIR model actually has a steady state.

The dynamics of the System Dynamics SIR model are in equilibrium (won't change any more) when the infected stock is 0. This might be shown formally but intuitively it is clear because only infected agents can lead to infections of susceptible agents which then make the transition to recovered after having gone through the infection phase.

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

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

Unfortunately this criterion alone does not suffice because when we look at the SIR+S model, which adds a cycle from Recovered back to Susceptible, we have the same termination criterion, but we cannot guarantee that it will run out of infected. We need an additional criteria.

3. The source of infected agents is the pool of susceptible agents which is monotonous decreasing (not strictly though!) because recovered agents do NOT turn back into susceptibles.

Thus we can conclude that a SIR model must enter a steady state after finite steps / in finite time.

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

Dependent Types and Idris' ability for totality- and termination-checking should theoretically allow us to proof that an agent-based SIR implementation terminates after finite time: if an implementation of the agent-based SIR model in Idris is total it is a formal proof by construction. Note that such an implementation should not run for a limited virtual time but run unrestricted of the time and the simulation should terminate as soon as there are no more infected agents, returning the termination time as an output. Also if we find a total implementation of the SIR model and extend it to the SIR+S model, which adds a cycle from Recovered back to Susceptible, then the simulation should become again non-total as reasoned above.

The HOTT book [79] states that lists, trees,... are inductive types/inductively defined structures where each of them is characterized by a corresponding *induction principle*. Thus, for a constructive proof of the totality of the agent-based SIR model we need to find the induction principle of it. This leaves us with the question of what the inductive, defining structure of the agent-based SIR model is? Is it a tree where a path through the tree is one way through the simulation or is it something else? It seems that such a tree would grow and then shrink again e.g. infected agents. Can we then apply this further to (agent-based) simulation in general?

So far we have no clear idea and understanding how to implement such a total implementation - this will be subject to quite substantial research. One might object to that undertaking and ask what we gain from it. We argue that investigating the correspondence between the equilibrium of an agent-based model and the totality of its implementation for the first time is reason enough because we expect to gain new insights from this undertaking.

Specifications The number of agents stays constant in SIR, this means no agents are created / destroyed during simulation, they only might change their state. We could conceptually specify that in the types as:

```
sirAgentNumberConstant : Vect s (SIRAgent Susceptible) ->
                         Vect i (SIRAgent Infected) ->
```

```
                    Vect r (SIRAgent Recovered) ->
                    Vect (s + i + r) (SIRAgent st)
```

Another property of the SIR model is, that the number of susceptibles, infected and recovered might change in each step but the sum will be the same as before. We could conceptually specify that in the types as:

```
sirStep : Vect s (SIRAgent Susceptible) ->
          Vect i (SIRAgent Infected) ->
          Vect r (SIRAgent Recovered) ->
          (Vect s' (SIRAgent Susceptible),
           Vect i' (SIRAgent Infected),
           Vect r' (SIRAgent Recovered), (s'+i'+r') = (s+i+r))
```

10.5 Dependently Typed Sugarscape

The other model we will employ as a use-case for the concepts of section **??** is the SugarScape model [29]. It is an exploratory model by which social scientists tried to explain phenomena found in societies in the real world. The main complexity of this model lies in the much more complex local state of the agents and the agent-agent interactions e.g. in case of trade and mating and a pro-active environment. Opposed to the SIR model agents behaviour is not modelled as a state-machine and time-semantics is not of that much importance: the simulation is stepped in unit-steps of $\Delta t = 1.0$ and in every time-step, all agents act in random order. Although there are equilibria e.g. in case all agents die out or the carrying capacity of an environment, trading prices, we think that this model is too complex for a total implementation in the cases.

Environment We have already introduced an implementation for a dependently typed 2D environment in section **??**. This can be directly used to implement the pro-active environment of SugarScape.

Existence Of Agents In SugarScape agents can die and be born thus on a technical level agents are added and removed dynamically during the simulation. This means we can employ proofs of existence of an agent for establishing interactions with another one. Also a proof might become invalid after a time. Also one can construct a proof only from a given time on e.g. when one wants to prove that agent X exists but agent X is only created at time t then before time t the prove cannot be constructed and is uninhabited and only inhabited from time t on.

Agent-Agent interactions In SugarScape agents interact with each other on a much more complex way than in SIR due to the complex behaviour. The two main complex use-cases are mating and trading between agents where both require multiple interaction-steps happening instantaneous without delay (that is, within 1 time-step). Both use-cases implement a protocol which we might be able to enforce using dependent types.

Hypotheses SugarScape is an exploratory model and although it is based on theoretical concepts from sociology, economics and epidemiology, it has strictly speaking no analytical or theoretical ground truth. Thus there are no means to validate this model and the researcher works by formulating hypotheses about the emergent properties of the model. So the approach the creators of SugarScape took in [29] was that they started from real world phenomenon and modelled the agent-interactions for them and hypothesized that out of this the real-world phenomenon will emerge. An example is the carrying capacity of an environment, as described in the first chapter: they hypothesized that the size of the population will reach a state where it will fluctuate around some mean because the environment cannot sustain more than a given number so agents not finding enough resources will simply die. Maybe we can encode such hypotheses using dependent types.

Chapter 11

A Case-Study

Apply my developed techniques to the Gintis paper (and its follow ups: the Ionescu paper [11] and a Masterthesis [30] on it). The aim of this study is to see:

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)

11.1 A pure functional implementation

11.2 Exploiting property-based tests

11.3 A dependently typed implementation

11.4 Discussion

Part IV:

Reflections

Chapter 12

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. About 20% exists already.

12.1 Benefits

12.2 Drawbacks

12.3 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.

12.3.1 Simulation in general

We already showed in the paper [89], 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.

12.3.2 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.

12.3.3 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

12.3.4 Recursive Simulation

Inspired by [33], 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

12.3.5 Multi Agent Systems

The fields of Multi Agent Systems (MAS) and ABS are closely related where ABS has drawn much inspiration from MAS [103], [101]. 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.

12.4 Peers Framework

TODO: discusses if and how peers object-oriented agent-based modelling framework can be applied to our pure functional approach. TODO: i need to re-read peers framework specifications / paper from the simulation bible book. Although peers framework uses UML and OO techniques to create an agent-based model, we realised from a short case-study with him that most of the framework can be directly applied to our pure functional approach as well, which is not a huge surprise, after all the framework is more a modelling guide than an

implementation one. E.g. a class diagram identifies the main datastructures, their operations and relations, which can be expressed equally in our approach - though not that directly as in an oo language but at least the class diagram gives already a good outline and understanding of the required datafields and operations of the respective entities (e.g. agents, environment, actors,...). A state diagram expresses internal states of e.g. an agent, which we discussed how to do in both our time- and even-driven approach. A sequence diagram e.g. expresses the (synchronous) interactions between agents or with their environment, something for which we developed techniques in our event-driven approach and we discuss in depth there.

Chapter 13

Conclusions

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

13.1 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] Altenkirch, T., Danielsson, N. A., Loeh, A., and Oury, N. Pi Sigma: Dependent Types Without the Sugar. In *Proceedings of the 10th International Conference on Functional and Logic Programming* (Berlin, Heidelberg, 2010), FLOPS'10, Springer-Verlag, pp. 40–55.

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

[5] Asta, S., Özcan, E., and Peer-Olaf, S. An investigation on test driven discrete event simulation. In *Operational Research Society Simulation Workshop 2014 (SW14)* (Apr. 2014).

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

[7] Bauer, A., and Pretnar, M. Programming with Algebraic Effects and Handlers. *Journal of Logical and Algebraic Methods in Programming 84*, 1 (Jan. 2015), 108–123. arXiv: 1203.1539.

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

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

[10] 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.

[11] 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.

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

[13] Brady, E. Programming and Reasoning with Algebraic Effects and Dependent Types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2013), ICFP '13, ACM, pp. 133–144.

[14] Brady, E. State Machines All The Way Down - An Architecture for Dependently Typed Applications. Tech. rep., 2016.

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

[16] Brady, E., and Hammond, K. Correct-by-Construction Concurrency: Using Dependent Types to Verify Implementations of Effectful Resource Usage Protocols. *Fundam. Inf. 102*, 2 (Apr. 2010), 145–176.

[17] Brady, E. C. Idris — systems programming meets full dependent types. In *In Proc. 5th ACM workshop on Programming languages meets program verification, PLPV '11* (2011), ACM, pp. 43–54.

[18] 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.

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

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

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

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

[23] 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.

[24] 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.

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

[26] 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.

[27] Epstein, J. M. Chapter 34 Remarks on the Foundations of Agent-Based Generative Social Science. In *Handbook of Computational Economics*, L. T. a. K. L. Judd, Ed., vol. 2. Elsevier, 2006, pp. 1585–1604.

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

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

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

[31] Fowler, S., and Brady, E. Dependent Types for Safe and Secure Web Programming. In *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages* (New York, NY, USA, 2014), IFL '13, ACM, pp. 49:49–49:60.

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

[33] 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.

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

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

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

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

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

[47] 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.

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

[49] 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.

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

[51] 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.

[52] 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.

[53] Lipovaca, M. *Learn You a Haskell for Great Good!: A Beginner's Guide*, 1 edition ed. No Starch Press, San Francisco, CA, Apr. 2011.

[54] 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.

[55] 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.

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

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

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

[59] 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.

[60] McKinna, J. Why Dependent Types Matter. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2006), POPL '06, ACM, pp. 1–1.

[61] 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.

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

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

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

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

[70] 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.

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

[72] 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.

[73] 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.

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

[75] 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.

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

[77] Pierce, B. C., Amorim, A. A. d., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., Tolmach, A., and Yorgey, B. *Programming Language Foundations*. Software Foundations series, volume 2. Electronic textbook, May 2018.

[78] 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.

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

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

[81] 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.

[82] 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.

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

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

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

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

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

[88] 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.

[89] 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).

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

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

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

[93] 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.

[94] 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.

[95] 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.

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

[97] 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.

[98] 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.

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

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

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

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

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

[104] 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 [100] [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' [100] 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 [100] 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 [100] 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 [100], 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 [100]. 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 [100].

A.9 Combat

Unfortunately [100] 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 [100] 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 [100].

A.11 Trading

For trading we had a look at the NetLogo implementation of [100]: there an agent engages in trading with its neighbours *over multiple rounds* until either MRSs cross over or no trade has happened anymore. Because [100] 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 [100] 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.