2nd Year Report

Functional Agent-Based Simulation

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

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

June 19, 2018

Abstract

This Ph.D. investigates how Agent-Based Simulations (ABS) can be implemented using the functional programming paradigm and what the benefits and drawbacks are when doing so. Due to the nature of the functional paradigm we hypothesize that by using this approach we can increase confidence in the correctness of the simulation to an unprecedented level not possible with the established object-oriented approaches in the field. The correctness of a simulation and its results is of paramount interest in scientific computing thus giving our research fundamental importance.

So far we researched *how* to do ABS using the functional paradigm, where we implemented a highly promising approach by building on Functional Reactive Programming using the library Yampa and generalising it to Monadic Stream Functions. By this we could show that ABS is indeed very possible in functional programming as it allowed us to implement a number of different agent-based models, incorporating discrete time-semantics similar to Discrete Event Simulation and continuous time-flows like System Dynamics.

During the research conducted so far, it became apparent that this approach exhibits a few unique properties which indeed supports our initial hypothesis. We have started to systematically explore these properties through the additional use of dependent types and will commit our research of the next 8 months to fully develop this. We expect that dependent types allow us to narrow the gap between model specification and implementation on an unprecedented level, implying by definition, that a simulation is correct-by-construction.

Contents

Chapter 1

Introduction

In the first half of the Ph.D. we have investigated *how* to do agent-based simulation (ABS) in functional programming. This step was necessary because there didn't exist any research or implementation we could build ours on. Also it served to develop a deeper understanding of functional programming and its application to ABS.

In this process, it became apparent that there are a few unique benefits to the established object-oriented approaches which can be subsumed under the common category of *increasing the correctness of the simulation*. This insight didn't come as a surprise as this is what the functional programming paradigm is known for and also the hypothesis with which we started with: *Functional programming will allow us to write ABS which is more likely to be correct.* Note that, strictly speaking, a simulation is either correct or not but we cannot decide this generally for software unless we are willing to pour in an extreme amount of formalisms and tests, so when we say 'increasing the correctness' or 'more likely to be correct' we mean that we can guarantee less bugs and less sources of potential bugs.

We commit the next year of the Ph.D. to explore this insight in a more rigorous way and push it to new levels. We hypothesise that this will allow us to write simulations which are *very* likely to be correct and allow a much deeper level of formal and informal reasoning, something not possible with the established object-oriented approaches of the field yet. Because correctness is of paramount importance in scientific computing, our research is a valuable contribution to the field and can be regarded as high impact work.

In the next section of this chapter we will briefly discuss our approach of *how* agent-based simulation can be done (Section 1.1) with the functional paradigm and its benefits and drawbacks. Then we will give a short overview of *why* this is of benefit (Section 1.2) and introduce concepts which will outline the research for the next 10 months. Also we will shortly discuss what this research is *not* doing (Section 1.3) and we give a short overview of the (intended) contributions (Section 1.4) of this research.

In the following chapters we will give an in-depth introduction to the con-

cepts used in the research of the following year, namely Verification & Validation in Chapter 3 and Dependent Types in Chapter 4. We also believe that although our research is conducted in the field of ABS, part of it is transferable to simulation in general and possibly other fields, closely related to ABS. We discuss this shortly in Chapter 5. We also have updated the chapters on Aims & Objectives 6, Work to Date 7 and Future Work Plan 8.

1.1 How

The essence of *how* to do agent-based simulation using the functional programming paradigm, as in the language Haskell, is described in the paper in Appendix A. The approach we developed is based on Functional Reactive Programming which allows to express discrete- and continuous-time systems in functional programming.

Benefits Following the conclusions of the paper, we got the following benefits, which support directly our initial hypothesis:

1. Run-Time robustness by compile-time guarantees - by expressing stronger guarantees already at compile-time we can restrict the classes of bugs which occur at run-time by a substantial amount due to Haskell's strong and static type system. This implies the lack of dynamic types and dynamic casts [1] which removes a substantial source of bugs. Note that we can still have run-time bugs in Haskell when our functions are partial.

2. Purity - By being explicit and polymorphic in the types about side-effects and the ability to handle side-effects explicitly in a controlled way allows to rule out non-deterministic side-effects which guarantees reproducibility due to guaranteed same initial conditions and deterministic computation. Also by being explicit about side-effects e.g. Random-Numbers and State makes it easier to verify and test.

3. Explicit Data-Flow and Immutable Data - All data must be explicitly passed to functions thus we can rule out implicit data-dependencies because we are excluding IO. This makes reasoning of data-dependencies and data-flow much easier as compared to traditional object-oriented approaches which utilize pointers or references.

4. Declarative - describing *what* a system is, instead of *how* (imperative) it works. In this way it is should be easier to reason about a system and its (expected) behaviour because it is more natural to reason about the behaviour of a system instead of thinking of abstract operational details.

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

Drawbacks Following the conclusions of the paper, we also found drawbacks.

The most fundamental one is that agent-agent and agent-environment interactions work very different because method calls and mutable data are not available. We had to invent new techniques for these kind of interactions which makes functional agent-based simulation conceptually more difficult to understand and implement models with. Despite these difficulties, it also makes those interactions more expressive and explicit and separates them into different categories which are conceptually different but are almost always implemented the same way in object-oriented approaches.

Another drawback is performance. So far, performance is not comparable to object-oriented approaches, but that was not the main focus of the research. Also pure functional programming as in Haskell, can exploit parallelism nearly for free due to its explicit data-flow, lack of side-effects and immutable data. This should make speeding up of simulations very easy, and if it is just running multiple replications in parallel. We leave this for further research as investigating this is worth a Ph.D. on its own.

1.2 Why

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. All of them ultimately suffer from same problems when it comes to verifying the correctness of the simulation.

The established way to test software in traditional 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 it is popularity is surging and more and more people become aware of that problem, but so far due to the highly dynamic nature of the type-system, run-time errors are still possible both in Python and Java.

1.2.1 Types to the rescue

In general, Types guide us in program construction by restricting the operations we can perform on the data. This means that by choosing types this reveals already a lot of our program and data and prevents us from making mistakes e.g. interpreting some binary data as text instead of a number. In strongly statically typed languages the types can do this already at compile-time which allows to rule out certain bugs already at compile-time. In general, we can say that for all bugs which can be ruled out at compile-time, we don't need to write property- or unit-tests, because those bugs cannot - per definition - occur at run-time, so it won't make sense to test their absence at run-time. Also, as Dijkstra famously put it: "Testing shows the presence, not the absence of bugs" - thus by induction we can say that compile-time guarantees save us from a potentially infinite amount of testing.

In general it is well established, that pure functional programming as in Haskell, allows to express much stronger guarantees about the correctness of a program *already at compile-time*. This is in fundamental contrast to imperative object-oriented languages like Java or Python where only primitive guarantees about types - mostly relationships between type-hierarchies - can be expressed at compile-time which directly implies that one needs to perform much more testing (user testing or unit-testing) at *run-time* to check whether the model is sufficiently correct. Thus guaranteeing properties already at compile-time frees us from writing unit-tests which cover these cases or test them at run time because they are *guaranteed to be correct under all circumstances, for all inputs.* In this regards we see pure functional programming as truly superior to the traditional object oriented approaches: they lead to implementations of models which are more likely correct because we can express more guarantees already at compile-time which directly leads to less bugs which directly increases the probability of the software being a correct implementation of the model. Having established this was only the first step in our paper in Appendix A, as outlined in the benefits above (see Section 1.1).

Although pure functional ABS as in Haskell allows us to leverage on the concepts of functional and its benefits (and drawbacks) we still rely heavily on

(property-based) testing to ensure correctness of a simulation because our approach still can have run-time bugs. Thus, the next step, which follows directly, is towards even stronger guarantees at compile-time, by using dependent types. Generally speaking, dependent types allow to push compile-time guarantees to a new level where we can express nearly arbitrary complex guarantees at compile-time because we can *compute types at compile-time*. This means that types are first-class citizen of the language and go as far as being formal proofs of the correctness of an implementation, allowing to narrow the gap between specification and implementation substantially. We hypothesise that the use of dependent types allows us to push the judgement of the correctness of a simulation to new, unprecedented level, not possible with the established object-oriented approaches so far. This has the direct consequence that the development process is very different and can reduce the amount of testing (both unit-testing and manual testing) substantially. Because one is implementing a simulation which is (as much as possible) correct-by-construction, the correctness (of parts) can be guaranteed statically.

This is supported by a talk [28], Tim Sweeney CEO of Epic Games in which he discusses the use of main-stream imperative object-oriented programming languages (C++) in the context of Game Programming. He 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 and reports that 50% of all bugs in the Game Engine Middleware Unreal, can be traced back to such problems. He then presents dependent types as a potential rescue to those problems.

Summarizing, we expect the following benefits (additionally to the ones described in Section 1.1) from adding dependent types to ABS:

1. Narrowing the gap between the model specification and its implementation reduces the potential for conceptual errors in model-to-code translation.

2. Less number of tests required due to guarantees being expressed already at compile time.

3. Higher confidence in correctness due to formal guarantees in code.

1.3 What Not

Because our research focuses on new implementation techniques and paradigms in agent-based simulation, there is always the question of "Why would you do that, what is wrong with the established way?". This means new techniques and paradigms always have to justify themselves in terms of benefits and drawbacks and our research is no exception to that. The way we approach this is not on a low technical level but on very broad conceptual levels which have been already established for object-oriented and functional programming paradigms in general. Thus we will *not* compare object-oriented to functional implementations

of a same model and go into lengthy debates over Lines Of Code, maintainability, extensibility, readability,... because many of those properties are highly subjective and depend very much on experience. In short: we want to avoid discussions over "which programming language is better" under all circumstances because they lead nowhere (just take a look in any technical forum / Reddit / Youtube and you will see the emotions going high like in religious debates). This also means that we are *not* comparing the general approach of pure functional programming in the instance of Haskell and object-oriented programming in the instance of Java and Python to implement ABS.

We claim from experience, that one has to pick the *right* programming language for the *right* task e.g. for web-development one would probably never pick Assembly and for embedded systems programming probably not Haskell. The reason for that is, that in web-development we prefer the higher abstraction which Assembly wouldn't provide because it is extremely close to hardware. This property in turn would be required in embedded systems programming which in turn Haskell is lacking. So when we pick a language we need to know its strengths which directly influences how we solve the problems at hand, thus it might make sense programming a web-application in Assembly if we need speed under all circumstances or use Haskell for implementing embedded programs when we need stronger guarantees about correctness.

Thus our research is to find out what the *right* task for functional programming and dependent types is in agent-based simulation - we believe it is correctness. So despite challenging the metaphor that *agents map naturally to objects*, we still think object-oriented programming is great for implementing ABS because we think it is the *easiest* way to go for teaching and low-impact models, not used in far reaching policy decisions because for those, correctness is not the primary objective.

1.4 Contributions

Although I am just half-way through the Ph.D. I anticipate and project the contributions of it:

1. This research is the first to *systematically* investigate the application of the functional programming paradigm, as in Haskell, to Agent-Based Simulation, identifying its benefits and drawbacks.

2. This research is the first to apply *dependent types* to Agent-Based Simulation to investigate its usefulness for increasing the correctness of a simulation.

3. The developed methods allow to implement Agent-Based Simulation which are guaranteed to be reproducible, have less sources of bugs, are easier to verify and thus more likely to be correct which is of paramount importance in high-impact scientific computing.

Chapter 2

Going Large-Scale with Softare Transactional Memory

Chapter 3

Verification & Validation

In this chapter we will have a closer look on the topic of Verification & Validation. It is of most importance to understand the ideas and concepts behind it, when we are referring to the *correctness of a simulation*, as in our initial hypothesis.

We first will give a short and concise introduction (3.1) on the topic [1] and then briefly discuss verification & validation in the context of agent-based simulation (3.2).

3.1 Introduction

Validation is the process of ensuring that a model or specification is sufficiently accurate for the purpose at hand.

Verification is the process of ensuring that the model design has been transformed into a computer model with sufficient accuracy.

[2] define validation as "are we building the right model?" and verification as "are we building the model right?" and.

In verification and validation the aim is to ensure, that the model is sufficiently accurate, which always implies its purpose. Therefore the purpose and objectives must be known before it is validated.

In this research we will primarily focus on verification, because its there where one ensures that the model is programmed correctly, the algorithms have been implemented properly, and the model does not contain errors, oversights, or bugs. Note that verification has a narrow definition and can be seen as a subset of the wider issue of validation. One distinguishes between:

- White-box Validation: detailed, micro check if each part of the model represent the real world with sufficient accuracy. It is therefore intrinsic to model coding. The ways to do it is checking the code, visual checks, inspecting output reports.

[1] In this short introduction we closely follow the book [25]

- Black-box Validation: overall macro check whether the model provides a sufficiently accurate representation of the real world system. It can only be performed once model code is complete. Ways to do this is comparison with the real system or with other (simpler) models.
- White-box Verification: compares the content of the model to the *conceptual* model. This is different to white-box validation which compares the content of the model to the *real world*
- Black-box Verification: treating the functionality to test as a black-box with inputs and outputs and comparing controlled inputs to expected outputs.

So in general one can see verification as a test of the fidelity with which the conceptual model is converted into the computer model. Verification (and validation) is a continuous process and if it is already there in the programming language / supported by then this is much easier to do. This is the fundamental basis of our hypothesis where we claim that by choosing a programming language which supports this continuous verification and validation process, then the result is an implementation of a model which is more likely to be correct.

Unfortunately, there is no such thing as general validity: a model should be built for one purpose as simple as possible and not be too general, otherwise it becomes too bloated and too difficult or impossible to analyse. Also, there may be no real world to compare against: simulations are developed for proposed systems, new production or facilities which don't exist yet. Further, it is questionable which real world one is speaking of: the real world can be interpreted in different ways, therefore a model valid to one person might not be valid to another. Sometimes validation struggles because the real world data are inaccurate or there is not enough time to verify and validate everything.

In general this implies that we can only *raise the confidence* in the correctness of the simulation: it is not possible to prove that a model is valid, instead one should think of confidence in its validity. Therefore, the process of verification and validation is not the proof that a model is correct but trying to prove that the model is incorrect! The more tests/checks one carries out which show that it is not incorrect, the more confidence we can place on the models validity.

3.2 Verification & Validation in Agent-Based Simulation

In our research we focus primarily on the *Verification* aspect of agent-based simulation: ensuring that the implementation reflects the specifications of the *conceptual* model - have we built the model right? Thus we are not interested in our research into making connections to the real world and always see the model specifications as our "last resort", our ground truth beyond nothing else exists. When there are hypotheses formulated, we always treat and interpret them in respect of the conceptual model.

In [21] the authors clarify on verification in ABS. Verification "... is essentially the question: does the model do what we think it is supposed to do? Whenever a model has an analytical solution, a condition which embraces almost all conventional economic theory, verification is a matter of checking the mathematics.". They say about validation that "In an important sense, the current process of building ABMs is a discovery process, of discovering the types of behavioural rules for agents which appear to be consistent with phenomena we observe.". Further they claim that "Because such models are based on simulation, the lack of an analytical solution (in general) means that verification is harder, since there is no single result the model must match. Moreover, testing the range of model outcomes provides a test only in respect to a prior judgment on the plausibility of the potential range of outcomes. In this sense, verification blends into validation."

So the baseline is that either one has an analytical model as the basis of an agent-based model or one does not. In the former case, e.g. the SIR model, one can very easily validate the dynamics generated by the simulation to the one generated by the analytical solution (e.g. through System Dynamics). In the latter case one has basically no idea or description of the emergent behaviour of the system prior to its execution e.g. SugarScape. It is important to have some hypothesis about the emergent property / dynamics. The question is how verification / validation works in this setting as there is no formal description of the expected behaviour: we don't have a ground-truth against which we can compare our simulation dynamics.

General there are the following basic verification & validation requirements to ABS [25], which all can be addressed in our *pure* functional approach as described in the paper in Appendix A:

- 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

3.2.1 Related Work

The work of [18] suggests good programming practice which is extremely important for high quality code and reduces bugs but real world practice and

experience shows that this alone is not enough, even the best programmers make mistakes which often can be prevented through a strong static or a dependent type system already at compile-time. What we can guarantee already at compile-time, doesn't need to be checked at run-time which saves substantial amount of time as at run-time there may be a large number of execution paths through the simulation which is almost always simply not feasible to check (note that we also need to check all combinations). This paper also cites modularity as very important for verification: divide and conquer and test all modules separately. We claim that this is especially easy in functional programming as code composes better than in traditional object-oriented programming due to the lack of interdependence between data and code as in objects and the lack of global mutable state (e.g. class variables or global variables) - this makes code extremely convenient to test. The paper also discusses statistical tests (the t test) to check if the outcome of a simulation is sufficiently close to real-world dynamics - we explicitly omit this as it part of validation and not the focus of this research.

[23]: "For some time now, Agent Based Modelling has been used to simulate and explore complex systems, which have proved intractable to other modelling approaches such as mathematical modelling. More generally, computer modelling offers a greater flexibility and scope to represent phenomena that do not naturally translate into an analytical framework. Agent Based Models however, by their very nature, require more rigorous programming standards than other computer simulations. This is because researchers are cued to expect the unexpected in the output of their simulations: they are looking for the 'surprise' that shows an interesting emergent effect in the complex system. It is important, then, to be absolutely clear that the model running in the computer is behaving exactly as specified in the design. It is very easy, in the several thousand lines of code that are involved in programming an Agent Based Model, for bugs to creep in. Unlike mathematical models, where the derivations are open to scrutiny in the publication of the work, the code used for an Agent Based Model is not checked as part of the peer-review process, and there may even be Intellectual Property Rights issues with providing the source code in an accompanying web page."

[16]: "a prerequisite to understanding a simulation is to make sure that there is no significant disparity between what we think the computer code is doing and what is actually doing. One could be tempted to think that, given that the code has been programmed by someone, surely there is always at least one person - the programmer - who knows precisely what the code does. Unfortunately, the truth tends to be quite different, as the leading figures in the field report, including the following: You should assume that, no matter how carefully you have designed and built your simulation, it will contain bugs (code that does something different to what you wanted and expected), "Achieving internal validity is harder than it might seem. The problem is knowing whether an unexpected result is a reflection of a mistake in the programming, or a surprising consequence of the model itself. [] As is often the case, confirming that the model was correctly programmed was substantially more work than pro-

gramming the model in the first place. This problem is particularly acute in the case of agent-based simulation. The complex and exploratory nature of most agent-based models implies that, before running a model, there is some uncertainty about what the model will produce. Not knowing a priori what to expect makes it difficult to discern whether an unexpected outcome has been generated as a legitimate result of the assumptions embedded in the model or, on the contrary, it is due to an error or an artefact created in the model design, its implementation, or its execution."

3.2.2 Testing

Although (pure) functional programming allows us to have stronger guarantees about the behaviour and absence of bugs of the simulation already at compile-time, we still need to test all the properties of our simulation which we cannot guarantee at compile-time.

We found property-based testing particularly well suited for ABS. Although it is now available in a wide range of programming languages and paradigms, propert-based testing has its origins in Haskell [11, 12] and we argue that for that reason it really shines in pure functional programming. Property-based testing allows to formulate *functional specifications* in code which then the property-testing library (e.g. QuickCheck [11]) tries to falsify by automatically generating random test-data covering as much cases as possible. When an input is found for which the property fails, the library then reduces it to the most simple one. It is clear to see that this kind of testing is especially suited to ABS, because we can formulate specifications, meaning we describe *what* to test instead of *how* to test (again the declarative nature of functional programming shines through). Also the deductive nature of falsification in property-based testing suits very well the constructive nature of ABS.

Generally we need to distinguish between two types of testing/verification: 1. testing/verification of models for which we have real-world data or an analytical solution which can act as a ground-truth - examples for such models are the SIR model, stock-market simulations, social simulations of all kind and 2. testing/verification of models which are just exploratory and which are only be inspired by real-world phenomena - examples for such models are Epsteins Sugarscape and Agent_Zero.

In both cases this leaves us with black-box and white-box Verification:

3.2.2.1 Black-Box Verification

In black-box Verification one generally feeds input and compares it to expected output. In the case of ABS we have the following examples of black-box test:

1. Isolated Agent Behaviour - test isolated agent behaviour under given inputs using and property-based testing.

2. Interacting Agent Behaviour - test if interaction between agents are correct .

3. Simulation Dynamics - compare emergent dynamics of the ABS as a whole under given inputs to an analytical solution or real-world dynamics in case there exists some using statistical tests.

4. Hypotheses- test whether hypotheses are valid or invalid using and property-based testing.

Using black-box verification and property-based testing we can apply for the following use cases for testing ABS in FRP:

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.

3.2.2.2 White-Box Verification

White-Box verification is necessary when we need to reason about properties like *forever*, *never*, which cannot be guaranteed from black-box tests. Additional help can be coverage tests with which we can show that all code paths have been covered in our tests.

3.2.3 Example: Property-Based Testing of SIR

As an example we discuss the black-box testing for the SIR model using property-testing. We test if the *isolated* behaviour of an agent in all three states Susceptible, Infected and Recovered, corresponds to model specifications. The crucial

thing though is that we are dealing with a stochastic system where the agents act *on averages*, which means we need to average our tests as well. We conducted the tests on the implementation found in the paper of Appendix A.

3.2.3.1 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
```

Susceptible Behaviour A susceptible agent *may* become infected, depending on the number of infected agents in relation to non-infected the susceptible agent has contact to. To make this property testable we run a susceptible agent for 1.0 time-unit (note that we are sampling the system with a smaller $\Delta t = 0.1$) and then check if it is infected - that is it returns infected as its current state.

Obviously we need to pay attention to the fact that we are dealing with a stochastic system thus we can only talk about averages and thus it does not suffice to only run a single agent but we are repeating this for e.g. $N = 10.000$ agents (all with different RNGs). We then need a formula for the required fraction of the N agents which should have become infected on average. Per 1.0 time-unit, a susceptible agent makes *on average* contact with β other agents where in the case of a contact with an infected agent the susceptible agent becomes infected with a given probability γ. In this description there is another probability hidden, which is the probability of making contact with an infected agent which is simply the ratio of number of infected agents to number not infected agents. The formula for the target fraction of agents which become infected is then: $\beta * \gamma * \frac{numberofinfected}{numberofnon-infected}$. To check whether this test has passed we compare the required amount of agents which on average should become infected to the one from our tests (simply count the agents which got infected and divide by N) and if the value lies within some small ϵ then we accept the test as passed.

Obviously the input to the susceptible agents which we can vary is the set of agents with which the susceptible agents make contact with. To save us from constructing all possible edge-cases and combinations and testing them with unit-tests we use property-testing with QuickCheck which creates them randomly for us and reduces them also to all relevant edge-cases. This is an

example for how to use property-based testing in ABS where QuickCheck can be of immense help generating random test-data to cover all cases.

Infected Behaviour An infected agent *will always* recover after a finite time, which is *on average* after δ time-units. Note that this property involves stochastics too, so to test this property we run a large number of infected agents e.g. $N = 10.000$ (all with different RNGs) until they recover, record the time of each agents recovery and then average over all recovery times. To check whether this test has passed we compare the average recovery times to δ and if they lie within some small ϵ then we accept the test as passed.

We use property-testing with QuickCheck in this case as well to generate the set of other agents as input for the infected agents. Strictly speaking this would not be necessary as an infected agent never makes contact with other agents and simply ignores them - we could as well just feed in an empty list. We opted for using QuickCheck for the following reasons:

- We wanted to stick to the interface specification of the agent-implementation as close as possible which asks to pass the states of all agents as input.
- We shouldn't make any assumptions about the actual implementation and if it REALLY ignores the other agents, so we strictly stick to the interface which requires us to input the states of all the other agents.
- The set of other agents is ignored when determining whether the test has failed or not which indicates by construction that the behaviour of an infected agent does not depend on other agents.
- We are not just running a single replication over 10.000 agents but 100 of them which should give black-box verification more strength.

Recovered Behaviour A recovered agent will stay in the recovered state *forever*. Obviously we cannot write a black-box test that truly verifies that because it had to run in fact forever. In this case we need to resort to white-box verification (see below).

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 [19]. 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.

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.

3.2.3.2 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.

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.

Chapter 4

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 [30, 20, 1].

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 [5] 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 4.1, 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 4.2 we briefly discuss ideas of how the concepts of dependent types could be applied to agent-based simulation and in Section 4.3 we very shortly discuss the connection between Verification & Validation and dependent types.

4.1 Introduction

There exist a number of excellent introduction to dependent types which we use as main ressources for this section: [30, 24, 27, 8, 22].

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.

2. Totality and termination - A total function is defined in [8] as: it terminates with a well-typed result or produces a non-empty finite prefix of a well-typed infinite result in finite 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.

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 4.1.3) 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.

4.1.1 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 4.1.3). 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
```

4.1.2 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
```

4.1.2.1 Kinds of Equality

In type theory there are different kinds of equality [1], 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.

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 4.1.3).

[1] 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`.

Still it is not very clear what *intensional* and *extensional* type theory means. The HOTT Book [24] 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).

4.1.3 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. constructive [2]. In general, the constructive position has been identified with realism and empirical computational content where the classical one with idealism and pragmatism.

In the classical view, the position is that to prove $\exists x.P(x)$ it is sufficient to prove that $\forall x.\neg P(x)$ leads to a contradiction. The constructive view would claim that only the contradiction is established but that a proof of existence has to supply an evidence of an x and show that $P(x)$ is provable. In the end this boils down whether to use proof by contradiction or not, which is sanctioned by the law of the excluded middle which says that $A \vee \neg A$ must hold. The classic position accepts that it does and such proofs of existential statements as above, which follow directly out of the law of the excluded middle, abound in mathematics [3]. The constructive view rejects the law of the excluded middle and thus the position that every statement is seen as true or false, independently of any evidence either way. [30] (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.*

[2] We follow the excellent introduction on constructive mathematics [30], chapter 3.

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

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

4.2 Dependent Types in Agent-Based Simulation

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 [13]: "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 (4.2.1), discuss general concepts where dependent types might be of benefit in ABS (4.2.2), present a dependently typed implementation of a 2D discrete environment (4.2.3) and finally discuss potential use of dependent types in the SIR model (4.2.4) and SugarScape model (4.2.5).

4.2.1 Related Work

In [4] 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 [9] discusses depend types to implement correct-by-construction concurrency in the Idris language [5]. 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 A) has shown that ABS can be implemented pure.

In [10] the authors discuss systems programming with focus on network packet parsing with full dependent types in the Idris language [5]. 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 [6] the author discusses programming and reasoning with algebraic effects and dependent types in the Idris language [5]. 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 [3] 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 A) 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 [15] 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 [7] 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 [8] 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 [26] 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.

4.2.2 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 4.2.3.

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 run-time 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.

4.2.3 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 4.1) 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)))
```

4.2.4 Dependently Typed SIR

We plan to prototype the concepts of section 4.2.2 in a dependently typed SIR implementation. One can object that the SIR model [17] 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 4.2.5), 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 [9] 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 4.2.3. This can be directly used to implement a SIR on a 2D environment as we have done in the paper in Appendix A.

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 [24] 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))
```

4.2.5 Dependently Typed Sugarscape

The other model we will employ as a use-case for the concepts of section 4.2.2 is the SugarScape model [14]. 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 4.2.3. 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 [14] 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.

4.3 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 [8]). 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.

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

When we implement our simulation things change a bit as we add another layer: the conceptual model, describing the phenomenon, which is an abstraction of reality. This description can be of many forms but can be regarded on a line between completely formal (economic models) to informal (sociology) but the implementation will follow that description. The fundamental difference here is that in this case we want our implementation to be exactly the same as the conceptual model. Contrary to the real-world, where it is not possible to find a *true* model (as was argued by Popper), on this level we actually can construct an implementation which matches the conceptual model exactly because we have a description of the conceptual model. In the end we transform the con-

ceptual model description in code, which is itself a formal description. In this translation process (speak: implementation / programming), one can make an endless number of mistakes. Generally we can distinguish between two classes of mistakes: 1) conceptual mistakes - wrong translation of the model specifications into code due to various reasons e.g. imprecise description, human error. The more precise an unambiguous a model description is, the less probable conceptual mistakes will be. 2) internal mistakes - normal programming mistakes e.g. access of arrays out of bounds, ... also using correlated Random Number generators.

Level 0: Real-World phenomenon Level 1: Conceptual model of the real-world phenomenon Level 2: Implementation of the conceptual model

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

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

Chapter 5

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.

5.1 Simulation

We already showed in the paper found in Appendix A, 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 section 4.2.2.

5.2 Multi Agent Systems

The fields of Multi Agent Systems (MAS) and ABS are closely related where ABS has drawn much inspiration from MAS [32], [31]. 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 reason-

ing about a system and their agents, to show that a system follows a formal specifications.

5.3 Distributed Search Algorithms

TODO: Julie Greensmith mentioned distributed search algorithms as an example where localized entities (speak: agents) are interacting with each other.

Chapter 6

Aims and Objectives

This chapter gives a compact and concise overview of the aims and objectives. Chapter 8 gives a more in-depth plan and details in how we will approach the aim in general and the objectives in particular.

6.1 Aim

The aim of this Ph.D. is to investigate how the pure functional programming paradigm without dependent types, as in Haskell, and with dependent types, as in Idris, can be used to implement Agent-Based Simulation and what the benefits are in doing so.

6.1.1 Hypotheses

Hypothesis 1: Using pure functional programming without dependent types to implement Agent-Based Simulations leads to simulation software which is easier to verify and more likely to be correct.

Hypothesis 2: Using pure functional programming with dependent types to implement Agent-Based Simulations allows to narrow the gap between model-specification and its implementation substantially up to a correct-by-construction level. By definition, this leads to a simulation which is even more likely to be correct.

6.2 Objectives

1. Implement the SIR and SugarScape model in pure functional Haskell. This will allow us to develop the fundamental concepts of pure functional programming in Agent-Based Simulation in general.

2. Explore the benefits and drawbacks of using *pure* functional programming (without dependent types) in Agent-Based Simulations.

3. Implement a dependently typed agent-based simulation of the SIR and SugarScape model. This will allow us to develop the fundamental concepts of dependent types in Agent-Based Simulation in general.

4. Explore how far we can narrow the gap between model-specification and implementation using dependent types in Agent-Based Simulation.

Chapter 7

Work To Date

In the first half of my PhD (October 2016 - March 2018) I have learned the underlying foundations of pure functional programming, did lots of prototyping and ultimately developed a way of implementing ABS in this approach. This resulted in a paper (see Appendix A) which discusses *how* to do agent-based simulation with pure functional programming as foundation and how to solve the fundamental problems of encapsulating agent-state, doing agent-interactions and bringing in environments in this setting.

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

7.1 Social Simulation Conference 2017

I participated in the Social Simulation Conference 2017 (SSC2017) in Dublin from 24th - 29th September. I gave a 30 minutes talk on our submitted paper [29] which was very well received and discussed. The paper got selected to be published in the Conference Proceedings.

7.2 Papers submitted: Pure Functional Epidemics

The paper as attached in Appendix A.

7.3 Haskell Communities and Activities Report (HCAR) May 2018

We wrote a new entry for the HCAR May 2018, which tries to compile and publish novel and on-going ideas in the Haskell community. It is freely available under `https://www.haskell.org/communities/05-2018/html/report.html`. We hope that our idea and the work of our PhD gets a bit more attention and may start some discussions with people interested in this work.

7.4 2nd Year Report

This document.

7.5 Talks

So far three talks were given:

1. Presenting our paper [29] at the SSC2017 .
2. Presentation of pure functional programming concepts in ABS to Master Students of my 1st Supervisor.
3. Presenting the ideas on dependent types in ABS as outlined in section 4.2 to the Functional Programming Lab at the FP Away Day 28/29th June 2018.

Chapter 8

Future Work Plan

We will be researching the *why* of our approach for the next 10 months. In this time we will investigate the use of dependent types in our pure functional approach to agent-based simulation which we hypothesise should allow an unprecedented level of verification and validation, not possible (even not on a theoretical level) with imperative, traditional object-oriented approaches. There exists literally no research on this topic thus it will form the unique and sufficiently advanced, novel contribution of our PhD to the field. We will also write an additional paper which will investigate how dependent types can be made of use in ABS. Around December 2018 I will start writing another paper which is targeted for an agent-based simulation journal and is written as a conceptual paper, describing the approach and benefits of purely and dependently typed agent-based simulation. While writing this paper I will start constructing the main argument structure of my thesis so I have structure already when I start writing the thesis in April 2019. The last 6 months of the PhD (April 2019 - September 2019) will be dedicated to writing up the thesis and conducting additional research if still necessary.

1. Researching the HOW: September 2016 - March 2018
2. Researching the WHY: April 2018 - March 2019
3. Writing the Thesis: April 2019 - September 2019

In this chapter we discuss in more detail how we plan to approach the aim and objectives stated in Chapter 6. Further we give a short overview of planned papers and present a Gantt-Chart 8.3 reflecting the most important activities and relevant milestones.

8.1 Approaching the Objectives: Dependent Types

We have already begun working on dependent types and the plan is to do an in-depth research and examination of the ideas outlined in the section 4.2 on

the concepts of dependent types in Agent-Based Simulation.

Although there exists research on bringing dependent types to FRP [26], we follow a fundamentally different approach in implementing ABS with dependent types. Instead of using FRP as described in section 1.1, we follow the approach taken in the papers described in related work on dependent types 4.2.1. The approach there is to implement a Monad, *indexed* over additional parameters - in our case pre & post state, time, environment boundaries,... - and write an interpreter for the operations the Monad supports.

The reason why we chose to follow a different approach was that we came to the conclusion that we would have to invest very much additional work to make dependently typed FRP work and that it would not allow us to encode the properties we want in types. Also it allows us to explore the other fundamental approach to program design in functional programming: implementing a *shallow* encoded EDSL. Although it might seem that we are throwing away a lot of work on *how* to do ABS in pure functional programming, this is not so: the work described in section 1.1 was a very important step, getting us familiar with the concepts and approaches which will allow us to transfer a lot of insights.

Implement SIR We implement an agent-based SIR model with an indexed monad, to explore the concepts described in section 4.2 in the context of an explanatory model, which has an analytical background.

Implement SugarScape We implement the SugarScape model with an indexed monad, to explore the concepts described in section 4.2 in the context of an exploratory model, which has no analytical background but follows hypotheses.

Extract Indexed Agent Monad After having implemented the dependently typed SIR and SugarScape we try to derive the common concepts into an indexed Agent Monad.

Total SIR The research on equilibrium and totality in the agent-based SIR model is a separate activity as we think that using indexed monads will probably not work for that approach and that we rather need to express relations in types from which the implementation will follow .

8.2 Planned Papers

For the remainder of the Ph.D. we planned for two more papers.

1. The first one will be about applying dependent types to agent-based simulation and targeted towards a functional programming journal. We plan on start writing it end of the year around November / December 2018.

2. The second paper will be conceptual *towards* paper which is basically a condensed version of all my research on pure functional programming and dependent types in Agent-Based Simulation and is targeted towards an agent-based simulation journal. With that paper we want to try to sell the concepts found in our research to the ABS community. We plan on start writing it around February 2019 as a precursor to start writing the thesis.

8.3 Writing the Thesis

I plan on start writing the thesis in April 2019 and hope to finish it at the end of the Ph.D. programme in September 2019. A first draft of a structure outline is attached in Appendix

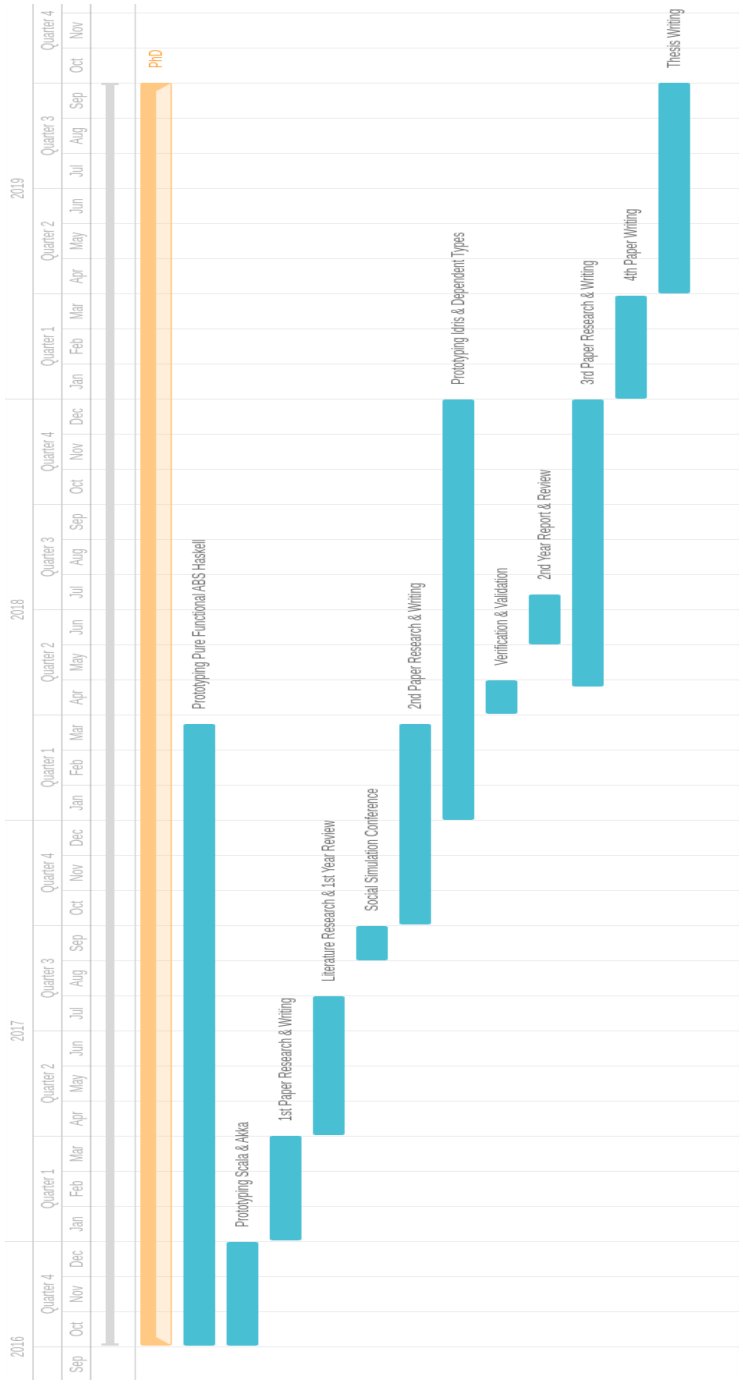


Figure 8.1: Gantt Chart of the Ph.D.

References

[1] 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.

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

[3] 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.

[4] 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.

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

[6] 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.

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

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

[9] 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.

[10] 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.

[11] Claessen, K., and Hughes, J. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2000), ICFP '00, ACM, pp. 268–279.

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

[13] 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.

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

[15] 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.

[16] Galn, J. M., Izquierdo, L. R., Izquierdo, S. S., Santos, J. I., del Olmo, R., Lpez-Paredes, A., and Edmonds, B. Errors and Artefacts in Agent-Based Modelling. *Journal of Artificial Societies and Social Simulation 12*, 1 (2009), 1.

[17] 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.

[18] Kleijnen, J. P. C. Verification and validation of simulation models. *European Journal of Operational Research 82*, 1 (Apr. 1995), 145–162.

[19] 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.

[20] 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.

[21] Ormerod, P., and Rosewell, B. Validation and Verification of Agent-Based Models in the Social Sciences. In *Epistemological Aspects of Computer Simulation in the Social Sciences*, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Oct. 2006, pp. 130–140.

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

[23] Polhill, J. G., Izquierdo, L. R., and Gotts, N. M. The Ghost in the Model (and Other Effects of Floating Point Arithmetic). *Journal of Artificial Societies and Social Simulation 8*, 1 (2005), 1.

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

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

[26] 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.

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

[28] 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.

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

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

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

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

Appendices

Appendix A

Pure Functional Epidemics

Submission history:

1. Submitted to Haskell Symposium 2018 on 30th March
   REJECTED on 18th May
2. Submitted to IFL 2018 on 25th May
   NOTIFICATION PENDING until 20th July

Pure Functional Epidemics

An Agent-Based Approach

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

ABSTRACT

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

So far mainly object-oriented techniques and languages have been used in ABS. Using the SIR model of epidemiology, which simulates the spreading of an infectious disease through a population, we show how to use pure Functional Reactive Programming to implement ABS. With our approach we can guarantee the reproducibility of the simulation at compile time and rule out specific classes of run-time bugs, something that is not possible with traditional object-oriented languages. Also, we found that the representation in a purely functional format is conceptually quite elegant and opens the way to formally reason about ABS.

KEYWORDS

Functional Reactive Programming, Monadic Stream Functions, Agent-Based Simulation

ACM Reference Format:
Jonathan Thaler, Thorsten Altenkirch, and Peer-Olaf Siebers. 2019. Pure Functional Epidemics: An Agent-Based Approach. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The traditional approach to Agent-Based Simulation (ABS) has so far always been object-oriented techniques, due to the influence of the seminal work of Epstein et al [9] 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* [22] which still holds up today.

In this paper we challenge this metaphor and explore ways of approaching ABS in a pure (lack of implicit side-effects) functional way using Haskell. By doing this we expect to leverage the benefits of pure functional programming [12]: higher expressivity through declarative code, being polymorph and explicit about side-effects through monads, more robust and less susceptible to bugs due to explicit data flow and lack of implicit side-effects.

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

Over the course of four steps, we derive all necessary concepts required for a full agent-based implementation. We start from a very simple solution running in the Random Monad which has all general concepts already there and then refine it in various ways, making the transition to Functional Reactive Programming (FRP) [36] and to Monadic Stream Functions (MSF) [26].

The aim of this paper is to show how ABS can be implemented in *pure* Haskell and what the benefits and drawbacks are. By doing this we give the reader a good understanding of what ABS is, what the challenges are when implementing it and how we solve these in our approach.

The contributions of this paper are:

- We present an approach to agent-based simulation using *declarative* analysis with FRP in which we systematically introduce the concepts of ABS to *pure* functional programming in a step-by-step approach. Also this work presents a new field of application to FRP as to the best of our knowledge the application of FRP to ABS (on a technical level) has not been addressed before. 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. This only becomes possible in pure functional programming where we can control the side-effects and can program side-effect polymorph, 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

IFL'18, August 2019, Lowell, MA, USA
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

e.g. related to dynamic typing, and the lack of implicit data-dependencies which are common in traditional imperative object-oriented approaches.

In Section 2 we define agent-based simulation, introduce functional reactive programming, arrowized programming and monadic stream functions, because our approach builds heavily on these concepts. In Section 3 we introduce the SIR model of epidemiology as an example model to explain the concepts of ABS. The heart of the paper is Section 4 in which we derive the concepts of a pure functional approach to ABS in four steps, using the SIR model. Section 5 discusses related work. Finally, we draw conclusions and discuss issues in Section 6 and point to further research in Section 7.

2 BACKGROUND

2.1 Agent-Based Simulation

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 [18, 30, 37]:

- 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 [8] 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 norm in ABS.
- Heterogeneity - although agents can have same properties like height, sex,... 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.

2.2 Functional Reactive Programming

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 [13, 14] as implemented in the library Yampa [4, 11, 21].

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

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

Yampa provides a number of combinators for expressing time-semantics, events and state-changes of the system. They allow to change system behaviour in case of events, run signal functions and generate stochastic events and random-number streams. We shortly discuss the relevant combinators and concepts we use throughout the paper. For a more in-depth discussion we refer to [4, 11, 21].

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, 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.

Randomness. In ABS one often needs to generate stochastic events which occur based on e.g. an exponential distribution. Yampa provides the combinator *occasionally :: RandomGen g => g -> Time -> b -> SF a (Event b)* for this. It takes a random-number generator, a rate and a value the stochastic event will carry. It generates events on average with the given rate. Note that at most one event will be generated and no 'backlog' is kept. This means that when this function is not sampled with a sufficiently high frequency, depending on the rate, it will lose events.

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

Running signal functions. To *purely* run a signal function Yampa provides the function *embed :: SF a b -> (a, [(DTime, Maybe a)]) -> [b]* which allows to run an SF for a given number of steps where in each step one provides the Δt and an input *a*. The function then returns the output of the signal function for each step. Note that the

input is optional, indicated by *Maybe*. In the first step at $t = 0$, the initial a is applied and whenever the input is *Nothing* in subsequent steps, the last a which was not *Nothing* is re-used.

2.3 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 [13, 14].

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 [23] 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 [27].

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

2.4 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 [25, 26]:

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

MSFs are also arrows which means we can apply arrowized programming with Patersons do-notation as well. MSFs are implemented in Dunai, which is available on Hackage. Dunai allows us to apply monadic transformations to every sample by means of combinators like *arrM :: Monad m => (a -> m b) -> MSF m a b* and *arrM_ :: Monad m => m b -> MSF m a b*.

3 THE SIR MODEL

To explain the concepts of ABS and of our pure functional approach to it, we introduce the SIR model as a motivating example and use-case for our implementation. It is a very well studied and understood compartment model from epidemiology [16] which allows to simulate the dynamics of an infectious disease like influenza, tuberculosis, chicken pox, rubella and measles spreading through a population [7].

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

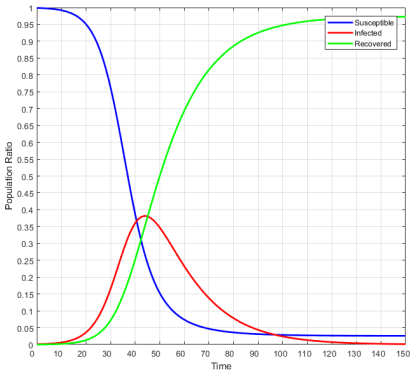


Figure 2: Dynamics of the SIR compartment model using the System Dynamics approach. Population Size N = 1,000, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time-steps.

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

This model was also formalized using System Dynamics (SD) [28]. 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. We won't go into detail here and provide the dynamics of such a solution for reference purposes, shown in Figure 2.

An Agent-Based approach

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 4.4 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 (DES).

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 [2]. Note that an agent does not know the other agents' state when making contact with it, thus we need a mechanism in which agents reveal their state in which they are in *at the moment of making contact*. This mechanism is an implementation detail which we will derive in our implementation steps. For now we only assume that agents can make contact with each other somehow.

This results in the following agent behaviour:

- *Susceptible*: A susceptible agent makes contact *on average* with β other random agents. For every *infected* agent it gets into contact with, it becomes infected with a probability of γ. If an infection happens, it makes the transition to the *Infected* state.
- *Infected*: An infected agent recovers *on average* after δ time units. This is implemented by drawing the duration from an exponential distribution [2] with $\lambda = \frac{1}{\delta}$ and making the transition to the *Recovered* state after this duration.
- *Recovered*: These agents do nothing because this state is a consuming state from which there is no escape: recovered agents stay immune and can not get infected again in this model.

4 DERIVING A PURE FUNCTIONAL APPROACH

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

In [33] two fundamental problems of implementing an agent-based simulation from a programming-language agnostic point of view is discussed. The first problem is how agents can be pro-active and the second how interactions and communication between agents can happen. For agents to be pro-active, they must be able to perceive the passing of time, which means there must be a concept of an agent-process which executes over time. Interactions between agents can be reduced to the problem of how an agent can expose information about its internal state which can be perceived by other agents. Further the authors have shown the influence of different deterministic and non-deterministic elements in agent-based simulation on the dynamics and how the influence of non-determinism can completely break them down or result in different dynamics despite same initial conditions. This means that we want to rule out any potential source of non-determinism.

In this section we will derive a pure functional approach for an agent-based simulation of the SIR model in which we will pose solutions to the previously mentioned problems. We will start out with a very naive approach and show its limitations which we overcome by adding FRP. Then in further steps we will add more concepts and generalisations, ending up at the final approach which utilises Monadic Stream Functions, a generalisation of FRP.

Of paramount importance is to keep our implementations pure which rules out the use of the IO Monad and thus any potential source of non-determinism under all circumstances because we would loose all compile time guarantees about reproducibility. Still we will make use of the Random and State Monad which indeed allow side-effects but the crucial point here is that we restrict side-effects only to these types in a controlled way without allowing general unrestricted effects [1].

4.1 Naive beginnings

We start by modelling the states of the agents. Infected agents are ill for some duration, meaning we need to keep track when an infected agent recovers. Also a simulation is stepped in discrete or continuous time-steps thus we introduce a notion of *time* and Δt by defining:

```
type Time      = Double
type TimeDelta = Double

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

Now we can represent every agent simply as its SIR state. We hold all our agents in a list:

```
type SIRAgent = SIRState
type Agents   = [SIRAgent]
```

Next we need to think about how to actually step our simulation. For this we define a function which advances our simulation with a fixed Δt until a given time t where in each step the agents are processed and the output is fed back into the next step. This is the source of pro-activity as agents are executed in every time step and can thus initiate actions based on the passing of time. Note that we step the simulation with a hard-coded $\Delta t = 1.0$ for reasons which become apparent when implementing the *susceptible* behaviour. As already mentioned, the agent-based implementation of the SIR model is inherently stochastic which means we need access to a random-number generator. We decided to use the Random Monad at this point as threading a generator through the simulation and the agents would be very cumbersome. Thus our simulation stepping runs in the Random Monad:

```
runSimulation :: RandomGen g => Time -> Agents -> Rand g [Agents]
runSimulation tEnd as = runSimulationAux 0 as []
  where
    runSimulationAux :: RandomGen g
      => Time -> Agents -> [Agents] -> Rand g [Agents]
    runSimulationAux t as acc
      | t >= tEnd = return (reverse (as : acc))
      | otherwise = do
        as' <- stepSimulation as
        runSimulationAux (t + 1.0) as' (as : acc)

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

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

```
processAgent :: RandomGen g => Agents -> SIRAgent -> Rand g SIRAgent
processAgent as Susceptible    = susceptibleAgent as
processAgent _  (Infected dur) = return (infectedAgent dur)
processAgent _  Recovered      = return Recovered
```

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

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

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

Lets look how we can implement the behaviour of a susceptible agent. It simply makes contact on average with a number of other agents and gets infected with a given probability if an agent it has contact with is infected. Here it becomes apparent that we implicitly assume that the simulation is stepped with a $\Delta t = 1.0$. If we would use a different Δt then we need to adjust the contact rate accordingly because it is defined *per time-unit*. This would amount to multiplying with the Δt which in combination with the discretisation using *floor* would lead to too few contacts being made, ultimately resulting in completely wrong dynamics.

When the agent gets infected, it calculates also its time of recovery by drawing a random number from the exponential distribution, meaning it is ill on average for *illnessDuration*.

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

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

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

```
infectedAgent :: TimeDelta -> TimeDelta -> SIRAgent
infectedAgent dur
    | dur' <= 0 = Recovered
    | otherwise = Infected dur'
  where
    dur' = dur - 1.0
```

Again note the hard-coded $\Delta t = 1.0$.

4.1.1 Results. When running our naive implementation with a population size of 1,000 we get the dynamics as seen in Figure 3. When comparing it to the dynamics of the reference in Figure 2, the agent-based dynamics are not as smooth which stems from the fact that the agent-based approach is inherently discrete and stochastic [17].

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

- Time - the simulation occurs over virtual time which is modelled explicitly divided into *fixed* Δt where at each step all agents are executed.
- Agents - we implement each agent as an individual, with the behaviour depending on its state.

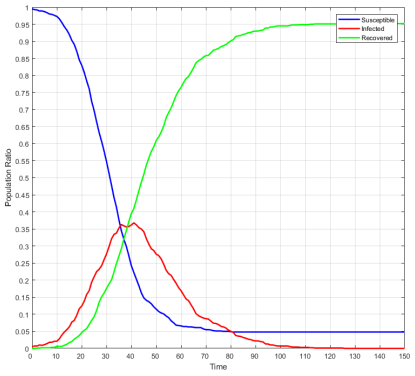


Figure 3: Naive simulation of SIR using the agent-based approach. Population of 1,000, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$, illness duration $\delta = 15$ with initially 1 infected agent. Simulation run for 150 time-steps with fixed $\Delta t = 1.0$.

- Feedback - the output state of the agent in the current time-step t is the input state for the next time-step $t + \Delta t$.
- Environment - as environment we implicitly assume a fully-connected network (complete graph) where every agent 'knows' every other agent, including itself and thus can make contact with all of them.
- Stochasticity - it is an inherently stochastic simulation, which is indicated by the Random Monad type and the usage of *randomBoolM* and *randomExpM*.
- Deterministic - repeated runs with the same initial random-number generator result in same dynamics. This may not come as a surprise but in Haskell we can guarantee that property statically already at compile time because our simulation runs in the Random Monad and *not* in the IO Monad. This guarantees that no external, uncontrollable sources of non-determinism can interfere with the simulation.

Nonetheless our approach has also weaknesses and dangers:

(1) We are using a hard-coded $\Delta t = 1.0$ because of the way our susceptible behaviour is implemented. Also Δt is dealt with explicitly in the infected behaviour where it is hard-coded to $\Delta t = 1.0$. This is not very modular and elegant and a potential source of errors - can we do better and find a more elegant solution?

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

We now move on to the next section in which we will address these points.

4.2 Adding Functional Reactive Programming

As shown in the first step, the need to handle Δt explicitly can be quite messy, is inelegant and a potential source of errors, also the explicit handling of the state of an agent and its behavioural function is not very modular. We can solve both these weaknesses

by switching to the Functional Reactive Programming paradigm, because it allows to express systems with discrete and continuous time-semantics.

In this step we are focusing on Arrowized FRP [13] using the library Yampa [11]. In it, time is handled implicitly, meaning it cannot be messed with, which is achieved by building the whole system on the concept of signal functions (SF). An SF can be understood as a process over time and is technically a continuation which allows to capture state using closures. Both these fundamental features allow us to tackle the weaknesses of our first step and push our approach further towards a truly elegant functional approach.

4.2.1 Implementation. We start by re-defining the SIR states, whereas now the illness duration is not needed any more. Also we re-define an agent to be an SF which receives the SIR states of all agents as input and outputs the SIR state of the agent:

```
data SIRState = Susceptible | Infected | Recovered

type SIRAgent = SF [SIRState] SIRState
```

Now we can define the behaviour of an agent to be the following:

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

Depending on the initial state we return the corresponding behaviour. Most notably is the difference that we are now passing a random-number generator instead of running in the Random Monad because signal functions as implemented in Yampa are not capable of being monadic. We see that the recovered agent ignores the random-number generator which is in accordance with the implementation in the previous step where it acts as a sink which returns constantly the same state:

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

When an event occurs we can change the behaviour of an agent using the Yampa combinator *switch*, which is much more elegant and expressive than the initial approach as it makes the change of behaviour at the occurrence of an event explicit. Thus a susceptible agent behaves as susceptible until it becomes infected. Upon infection an *Event* is returned which results in switching into the *infectedAgent* SF, which causes the agent to behave as an infected agent from that moment on. Instead of randomly drawing the number of contacts to make, we now follow a fundamentally different approach by using Yampas *occasionally* function. This requires us to carefully select the right Δt for sampling the system as will be shown in results.

```
susceptibleAgent :: RandomGen g => g -> SIRAgent
susceptibleAgent g =
    switch (susceptible g) (const (infectedAgent g))
  where
    susceptible :: RandomGen g
      => g -> SF [SIRState] (SIRState, Event ())
    susceptible g = proc as -> do
      makeContact <- occasionally g (1 / contactRate) () -< ()
      if isEvent makeContact
        then (do
          a <- drawRandomElemSF g -< as
          case a of
            Infected -> do
              i <- randomBoolSF g infectivity -< ()
              if i
                then returnA -< (Infected, Event ())
                else returnA -< (Susceptible, NoEvent)
            _        -> returnA -< (Susceptible, NoEvent))
        else returnA -< (Susceptible, NoEvent)
```

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

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

The infected agent behaves as infected until it recovers, on average after the illness duration, after which it behaves as a recovered agent by switching into *recoveredAgent*. As in the case of the susceptible agent, we use the *occasionally* function to generate the event when the agent recovers. Note that the infected agent ignores the states of the other agents as its behaviour is completely independent of them.

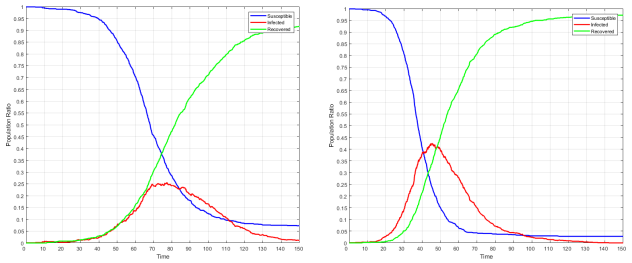
```
infectedAgent :: RandomGen g => g -> SIRAgent
infectedAgent g = switch infected (const recoveredAgent)
  where
    infected :: SF [SIRState] (SIRState, Event ())
    infected = proc _ -> do
      recEvt <- occasionally g illnessDuration () -< ()
      let a = event Infected (const Recovered) recEvt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

Running and stepping the simulation works now a bit differently, using Yampas function *embed*:

```
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, [4, 21] 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)
```



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

Figure 4: FRP simulation of agent-based SIR showing the influence of different Δt. Population size of 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.

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

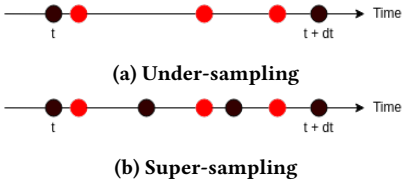
```
dpSwitch :: Functor col
         -- routing function
         => (forall sf. a -> col sf -> col (b, sf))
         -- SF collection
         -> col (SF b c)
         -- SF generating switching event
         -> SF (a, col c) (Event d)
         -- continuation to invoke upon event
         -> (col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

4.2.2 Results. The dynamics generated by this step can be seen in Figure 4.

In this step we followed the FRP approach which is fundamentally different from the previous step in Section 4.1 because in FRP we assume a continuous flow of time as opposed to discrete time-steps of $\Delta t = 1.0$ previously. This 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



(a) Under-sampling

(b) Super-sampling

Figure 5: 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.

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 4a. This issue is known as under-sampling and is described in Figure 5.

For tackling this issue we have two options. The first one is to use a smaller Δt as can be seen 4b, 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.

An approach to super-sampling would be to introduce a new combinator to Yampa which allows us to super-sample other signal functions.

```
superSampling :: Int -> SF a b -> SF a [b]
```

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. We have investigated super-sampling more in-depth but have to omit this due to lack of space.

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

A very severe problem, very 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 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*. Note that this is not an issue in Sections 4.1 and 4.3 as we are using the Random Monad, which never uses the same random-number generator twice thus resulting in guaranteed uncorrelated stochastics.

So far we have an acceptable implementation of an agent-based SIR approach. What we are lacking at the moment is a general treatment of an environment. To conveniently introduce it we want to make use of monads which is not possible using Yampa. In the next step we make the transition to Monadic Stream Functions as introduced in Dunai [26] which allows FRP within a monadic context.

4.3 Generalising to Monadic Stream Functions

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

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

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

4.3.2 Random Monad. Using the Identity Monad does not gain us anything but it is a first step towards a more general solution. Our next step is to replace the Identity Monad by the Random Monad which will allow us to get rid of the RandomGen arguments to our functions and run the whole simulation within the Random Monad *again* just as we started but now with the full features functional reactive programming. We start by re-defining the SIRMonad and SIRAgent:

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

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

```
occasionallyM :: MonadRandom m => Time -> b -> SF m a (Event b)
```

4.3.3 Discussion. So far making the transition to MSFs does not seem as compelling as making the move from the Random Monad to FRP in the beginning. Running in the Random Monad within FRP is convenient but we could achieve the same by passing RandomGen around as we already demonstrated. Still it guarantees us that we won't have correlated stochastics as discussed in the previous section. In the next step we introduce the concept of a read/write environment which we realise using a StateT monad. This will show the real benefit of the transition to MSFs.

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

Figure 6: Common neighbourhoods in discrete 2D environments of Agent-Based Simulation.

4.4 Adding an environment

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

In ABS agents are often situated within a discrete 2D environment [9] which is simply a finite NxM grid with either a Moore or von Neumann neighbourhood (Figure 6). Agents are either static or can move freely around with cells allowing either single or multiple occupants.

We can directly map the SIR model to a discrete 2D environment by placing the agents on a corresponding 2D grid with an unrestricted neighbourhood. The behaviour of the agents is the same but they select their interactions directly from the environment. Also instead of feeding back the states of all agents as inputs, agents now communicate through the environment by revealing their current state to their neighbours by placing it on their cell. Agents can read the states of all their neighbours which tells them if a neighbour is infected or not. For purposes of a more interesting approach, we restrict the neighbourhood to Moore (Figure 6b).

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

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

Next we redefine our monad stack and agent signal function. We use a StateT transformer on top of our Random Monad from the previous step with *SIREnv* as type for the state. Our agent signal function now has unit input and output type, which indicates that the actions of the agents are only visible through side-effects in the monad stack they are running in.

```
type SIRMonad g = StateT SIREnv (Rand g)
type SIRAgent g = SF (SIRMonad g) () ()
```

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

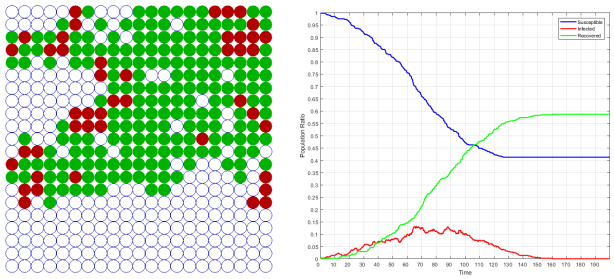
```
susceptibleAgent :: RandomGen g => Disc2dCoord -> SIRAgent g
susceptibleAgent coord
    = switch susceptible (const (infectedAgent coord))
  where
    susceptible :: RandomGen g
      => SF (SIRMonad g) () ((), Event ())
    susceptible = proc _ -> do
      makeContact <- occasionallyM (1 / contactRate)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

neighbours :: SIREnv -> Disc2dCoord -> Disc2dCoord
           -> [Disc2dCoord] -> [SIRState]

moore :: [Disc2dCoord]
moore = [ topLeftDelta,    topDelta,     topRightDelta,
          leftDelta,                     rightDelta,
          bottomLeftDelta, bottomDelta,  bottomRightDelta ]

topLeftDelta :: Disc2dCoord
topLeftDelta      = (-1, -1)
topDelta :: Disc2dCoord
topDelta          = ( 0, -1)
...
```

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

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

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

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



(a) $t = 100$ **(b) Dynamics over time**

Figure 7: Simulating the agent-based SIR model on a 21x21 2D grid with Moore neighbourhood (Figure 6b), a single infected agent at the center and same SIR parameters as in Figure 2. Simulation run until $t = 200$ with fixed $\Delta t = 0.1$. Last infected agent recovers shortly after $t = 160$. The susceptible agents are rendered as blue hollow circles for better contrast.

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

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

4.4.2 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 7.

Note that the dynamics of the spatial SIR simulation which are seen in Figure 7b look quite different from the reference dynamics of Figure 2. This is due to a much more restricted neighbourhood which results in far fewer infected agents at a time and a lower number of recovered agents at the end of the epidemic, meaning that fewer agents got infected overall.

4.4.3 Discussion. At first the environment approach might seem a bit overcomplicated and one might ask what we have gained by using an unrestricted neighbourhood where all agents can contact all others. The real advantage is that we can introduce arbitrary restrictions on the neighbourhood as shown with the Moore neighbourhood.

Of course an environment is not restricted to be a discrete 2D grid and can be anything from a continuous N-dimensional space

to a complex network - one only needs to change the type of the StateT monad and provide corresponding neighbourhood querying functions. The ability to place the heterogeneous agents in a generic environment is also the fundamental advantage of an agent-based over other simulation approaches and allows us to simulate much more realistic scenarios.

4.5 Additional Steps

ABS involves a few more advanced concepts which we don't fully explore in this paper due to lack of space. Instead we give a short overview and discuss them without presenting code or going into technical details.

4.5.1 Agent-Transactions. Agent-transactions are necessary when an arbitrary number of interactions between two agents need to happen instantaneously without time-lag. The use-case for this are price negotiations between multiple agents where each pair of agents needs to come to an agreement in the same time-step [9]. In object-oriented programming, the concept of synchronous communication between agents is implemented directly with method calls.

We have implemented synchronous interactions, which we termed agent-transactions in an additional step. We solved it pure functionally by running the signal functions of the transacting agent pair as often as their protocol requires but with $\Delta t = 0$, which indicates the instantaneous character of agent-transactions.

4.5.2 Event Scheduling. Our approach is inherently time-driven where the system is sampled with fixed Δt. The other fundamental way to implement an ABS in general, is to follow an event-driven approach [20] which is based on the theory of Discrete Event Simulation [38]. In such an approach the system is not sampled in fixed Δt but advanced as events occur where the system stays constant in between. Depending on the model, in an event-driven approach it may be more natural to express the requirements of the model.

In an additional step we have implemented a rudimentary event-driven approach which allows the scheduling of events but had to omit it due to lack of space. Using the flexibility of MSFs we added a State transformer to the monad stack which allows enqueuing of events into a priority queue. The simulation is advanced by processing the next event at the top of the queue which means running the MSF of the agent which receives the event. The simulation terminates if there are either no more events in the queue or after a given number of events, or if the simulation time has advanced to some limit. Having made the transition to MSFs, implementing this feature was quite straight forward which shows the power and strength of the generalised approach to FRP using MSFs.

4.5.3 Dynamic Agent creation. In the SIR model, the agent population stays constant - agents don't die and no agents are created during simulation - but some simulations [9] require dynamic agent creation and destruction. We can easily add and remove agents signal functions in the recursive switch after each time-step. The only problem is that creating new agents requires unique agent ids but with the transition to MSFs we can add a monadic context which allows agents to draw the next unique agent id when they create a new agent.

5 RELATED WORK

The amount of research on using pure functional programming with Haskell in the field of ABS has been moderate so far. Most of the papers are related to the field of Multi Agent Systems and look into how agents can be specified using the belief-desire-intention paradigm [6, 15, 32].

A library for DES and SD in Haskell called *Aivika 3* is described in the technical report [31]. It is not pure, as it uses the IO Monad under the hood and comes only with very basic features for event-driven ABS, which allows to specify simple state-based agents with timed transitions.

Using functional programming for DES was discussed in [15] where the authors explicitly mention the paradigm of FRP to be very suitable to DES.

A domain-specific language for developing functional reactive agent-based simulations was presented in [35]. This language called FRABJOUS is human readable and easily understandable by domain-experts. It is not directly implemented in FRP/Haskell but is compiled to Yampa code which they claim is also readable. This supports that FRP is a suitable approach to implement ABS in Haskell. Unfortunately, the authors do not discuss their mapping of ABS to FRP on a technical level, which would be of most interest to functional programmers.

Object-oriented programming and simulation have a long history together as the former one emereged out of Simula 67 [5] 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 [10]. 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 our approach in Section 4.4. 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.

6 CONCLUSIONS

Our approach is radically different from traditional approaches in the ABS community. First it builds on the already quite powerful FRP paradigm. Second, due to our 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 and State 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 [27]. Also by using FRP we gain all the benefits from it and can use research on testing, debugging and exploring FRP systems [24, 27].

Issues

Currently, the performance of the system is not comparable to imperative implementations but our research was not focusing on this aspect. We leave the investigation and optimization of the performance aspect of our approach for further research.

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 [29] 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 an agent can be hidden behind a polymorphic interface which is much more abstract than in our approach. Also the identity of an agent is much clearer in object-oriented programming due to the concept of object-identity and the encapsulation of data and methods.

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 straight-forward in object-oriented programming, where it is achieved using method-calls mutating the internal state of the agent, but that comes at the cost of a new class of bugs due to implicit data flow. In pure functional programming these data flows are explicit but our current approach of feeding back the states of all agents as inputs is not very general. We have added further mechanisms of agent interaction which we had to omit due to lack of space.

7 FURTHER RESEARCH

We see this paper as an intermediary and necessary step towards dependent types for which we first needed to understand the potential and limitations of a non-dependently typed pure functional approach in Haskell. Dependent types are extremely promising in functional programming as they allow us to express stronger guarantees about the correctness of programs and go as far as allowing to formulate programs and types as constructive proofs which must be total by definition [1, 19, 34].

So far no research using dependent types in agent-based simulation exists at all. In our next paper we want to explore this for the first time and ask more specifically how we can add dependent types to our pure functional approach, which conceptual implications this has for ABS and what we gain from doing so. We plan on using Idris [3] 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 hypothesize that dependent types could help ruling out even more classes of bugs at compile time and encode invariants and model specifications on the type level, which implies that we don't need to test them using e.g. property-testing with QuickCheck. This would allow the ABS community to reason about a model directly in code:

- Accessing the environment in section 4.4 involves 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 for coordinates one should be able to ensure already at compile time that access happens only within the bounds of the environment.
- In the SIR implementation one could make wrong state-transitions e.g. 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 many agent-based models define their agents in terms of state-machines.
- An infected agent recovers after a given time - the transition of infected to recovered is a timed transition. Nothing prevents us from *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.
- In more sophisticated models agents interact in more complex ways with each other e.g. through message exchange using agent IDs to identify target agents. The existence of an agent is not guaranteed and depends on the simulation time because agents can be created or terminated at any point during simulation.
  Dependent types could be used to implement agent IDs as a proof that an agent with the given id exists *at the current time-step*. This also implies that such a proof cannot be used in the future, which is prevented by the type system as it is not safe to assume that the agent will still exist in the next step.
- In our implementation, we terminate the SIR model always after a fixed number of time-steps. We can informally reason that restricting the simulation to a fixed number of time-steps is not necessary because the SIR model *has to* reach a steady state after a finite number of steps. This means that at that point the dynamics won't change any more, thus one can safely terminate the simulation. Informally speaking,

the reason for that is that eventually the system will run out of infected agents, which are the drivers of the dynamic. We know that all infected agents will recover after a finite number of time-steps *and* that there is only a finite source for infected agents which is monotonously decreasing.
Using dependent types it might be possible to encode this in the types, resulting 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.

ACKNOWLEDGMENTS

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

REFERENCES

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

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

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

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

[5] Ole-johan Dahl. 2002. The birth of object orientation: the simula languages. In *Software Pioneers: Contributions to Software Engineering, Programming, Software Engineering and Operating Systems Series*. Springer, 79–90.

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

[7] Richard H. Enns. 2010. *It's a Nonlinear World* (1st ed.). Springer Publishing Company, Incorporated.

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

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

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (Sept. 1991), 1305–1320. https://doi.org/10.1109/5.97300

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

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

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

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

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

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

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

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

[19] James McKinna. 2006. Why Dependent Types Matter. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, New York, NY, USA, 1–1. https://doi.org/10.1145/1111037.1111038

[20] Ruth Meyer. 2014. Event-Driven Multi-agent Simulation. In *Multi-Agent-Based Simulation XV (Lecture Notes in Computer Science)*. Springer, Cham, 3–16. https://doi.org/10.1007/978-3-319-14627-0_1

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

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

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

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

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

[26] Ivan Perez, Manuel Baerenz, and Henrik Nilsson. 2016. Functional Reactive Programming, Refactored. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, New York, NY, USA, 33–44. https://doi.org/10.1145/2976002.2976010

[27] Ivan Perez and Henrik Nilsson. 2017. Testing and Debugging Functional Reactive Programming. *Proc. ACM Program. Lang.* 1, ICFP (Aug. 2017), 2:1–2:27. https://doi.org/10.1145/3110246

[28] Donald E. Porter. 1962. Industrial Dynamics. Jay Forrester. M.I.T. Press, Cambridge, Mass.; Wiley, New York, 1961. xv + 464 pp. Illus. $18. *Science* 135, 3502 (Feb. 1962), 426–427. https://doi.org/10.1126/science.135.3502.426-a

[29] Neil Sculthorpe and Henrik Nilsson. 2009. Safe Functional Reactive Programming Through Dependent Types. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 23–34. https://doi.org/10.1145/1596550.1596558

[30] Peer-Olaf Siebers and Uwe Aickelin. 2008. Introduction to Multi-Agent Simulation. *arXiv:0803.3905 [cs]* (March 2008). http://arxiv.org/abs/0803.3905 arXiv: 0803.3905.

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

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

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

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

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

[36] Zhanyong Wan and Paul Hudak. 2000. Functional Reactive Programming from First Principles. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 242–252. https://doi.org/10.1145/349299.349331

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

[38] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. 2000. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press. Google-Books-ID: REzmYOQmHuQC.

Received May 2018

Appendix B

Thesis Structure

This appendix gives a first draft of the structure outline of the thesis which I plan on start writing in April 2019.

B.1 Introduction

This chapter is the introduction to the thesis and motivates it, describes the aim and scope of the Ph.D. Further it states the hypotheses and contributions.

- Main Argument: Defining the problem, motivation, aim and scope of the Ph.D.
- Hypotheses: Precisely stating the hypotheses which will form the points of reference for the whole research.
- Contributions: Precisely list the contribution to knowledge this Ph.D. makes and list all papers which were written (and published) during this Ph.D.
- Structure of the Thesis: describe the thesis structure.

B.2 Part I: General Concepts

In this part the general concepts for the Ph.D. are presented: they define the domain, context and background required to understand the central research of the Ph.D. in the following parts.

B.2.1 Introduction to Agent-Based Simulation

This chapter gives an introduction to ABS and gives a precise definition of the understanding of ABS within this ABS.

- History

- ABS vs. MAS
- ABS defined
- ABS examples e.g. SIR & SugarScape
- Event- vs. Time-Driven

B.2.2 Established Implementation Techniques

This chapter discusses the most widely used, established techniques to implement ABS and their approach to ensuring the correctness of a simulation.

- Frameworks: NetLogo, Anylogic
- Libraries: RePast, DesmoJ
- Programming: Java, Python
- Correctness: ad-hoc, manual testing, test-driven development

B.2.3 Validation & Verification

This chapter introduces the important concepts of validation & verification - follows mostly the introduction given in the 2nd annual report.

- Validation
- Verification
- V & V in the context of ABS.
- Verification is the central interest in this Ph.D.

B.3 Part II: Pure functional ABS

This part shows *how* agent-based simulation can be done with pure functional programming as in Haskell. It is based on the paper of Appendix **??** with much more detail and in-depth added from notes and reports written so far. Due to the different programming paradigm testing and verification works a bit different, which is described as well. Also modelling within a pure functional paradigm is discussed by putting it into a context of Peers framework. It shows that by just using pure functional programming we arrive at a number of benefits, which are discussed together with the drawbacks.

B.3.1 Introduction to Pure Functional Programming

This chapter shortly introduces to the functional programming paradigm as in Haskell. It further describes the concepts of side-effects and purity.

- Introduce basic concepts of functional programming in Haskell: functions, types, recursion, algebraic data-types, higher-order functions, continuations
- Define and explain side-effects and purity: monads, different types of effects, explain IO and that it is of fundamental importance to avoid it in our research.

B.3.2 Concepts of Functional ABS

This chapter derives the concepts of doing ABS in pure functional programming. It does that step-by-step by introducing it through functional reactive programming.

- Introduce Functional Reactive Programming
- Agent representation.
- Environment representation.
- Agent-Interactions - This is the central problem of the FP approach as basically the agent-interactions define the level of abstractions over the agents. Unfortunately this is easier and more elegant in object-oriented programming. Still, by using a strong static type system we are more explicit about agent-interactions and we can have advantages which OOP doesn't have. Also we show that there are multiple different kinds of agent-interactions, depending on whether it is a time- or event-driven ABS.
- Agent-Environment interaction.

B.3.3 Testing & Verification

This chapter Describes how testing & verification works in pure functional programming by looking on what and how to test in functional programming.

- Testing in functional programming
- Strong Static Types rule out some classes of bugs and make some tests obsolete.
- Property-Based testing: QuickCheck.
- Using Property-Based testing in ABS for specification testing.
- Reasoning about code

B.3.4 Going Large-Scale

Using Software Transactional Memory (STM) within Monadic Stream Functions it is possible to scale the pure functional ABS approach of FRP up, potentially running hundreds of thousands of agents.

- STM explained
- Shared Environment: TVars
- Messaging: TQueues
- Performance comparison

B.3.5 Discussion

This chapter gives an in-depth discussion of the pure functional approach to ABS, its benefits and drawbacks and outlines further research.

- Benefits - guaranteed reproducibility, less sources of run-time bugs, easier verification
- Drawbacks - performance, higher initial complexity
- Further Research - parallelise using Cloud-Haskell, recursive ABS

B.4 Part III: Dependent types in ABS

This part describes how dependent types can be applied to pure functional ABS and what we gain from doing so.

B.4.1 Introduction to Dependent Types

A brief introduction to dependent types inspired by my 2nd annual report introduction.

- Example
- Equality as Type
- Philosophical Foundations: Constructive mathematics

B.4.2 Concepts of Dependently Typed ABS

This chapter gives an in-depth discussion on how dependent types can be made of use in pure functional ABS.

- Environment Access
- State-Machines

- Flow Of Time
- Existence Of Agents
- Agent-Agent Interactions
- Specification and properties
- Hypotheses
- Equlibrium-Totality correspondence

B.4.3 Testing & Verificatoin

This chapter discusses the fundamentally different approach to implementation using dependent types in ABS which now follows a much more constructive, type-driven approach.

- Test-Driven (deductive) vs. Type-Driven (constructive) approach: in established oo (and to an extent, pure functional ABS because cannot make as strong guarantees) the approach is test-driven, in dependent types it is type-driven
- Correct-By-Construction

B.4.4 Discussion

This chapter gives an in-depth discussion of using dependent types in ABS and its benefits and drawbacks and shortly presents further research.

- Benefits: correct-by-construction
- Drawbacks: ?
- Further Research: ?

B.5 Part IV: Conclusions

An in-depth discussion of the whole thesis in the light of the initial hypotheses, scope and aims.

- Hypotheses revisited
- Generalising Research: Simulation in general, Multi Agent System, Distributed Search Algorithms
- Further Research

B.6 Appendix A: Philosophical Aspects

This appendix investigates the connection between the constructiveness of ABS and dependent types (and the impedance mismatch between test-driven development and ABS).