THE AGENTS NEW CLOTHS?
TOWARDS PURE FUNCTIONAL PROGRAMMING IN ABS

Jonathan Thaler
Peer Olaf Siebers

School Of Computer Science
University of Nottingham
7301 Wollaton Rd
Nottingham, United Kingdom
{jonathan.thaler,peer-olaf.siebers}@nottingham.ac.uk

ABSTRACT

The established, traditional approach to implement and engineer Agent-Based Simulations (ABS) so far has primarily been object-oriented with Python and Java being the most popular languages. In this paper we explore an orthogonal route to this problem and investigate the pure functional programming paradigm, using the language Haskell, for implementing ABS. We give a short, high level introduction into core pure functional programming paradigm features and how they can be made of use to implement ABS. To put our approach to a practical test we present as a case-study a *full and verified* implementation of the seminal Sugarscape model. With this case-study we are able to show that pure functional programming as in Haskell has a valid place in engineering clean, robust and maintainable ABS implementations. Further we show that we can directly leverage the benefits of pure functional programming to ABS: we have very few potential bugs at runtime, can easily exploit data-parallelism, concurrency is much less painful to add, code-testing in general is very convenient and powerful and with property-based testing in particular we present a new code-testing technique to ABS which hasn't been discussed before. The main drawback of using the pure functional programming paradigm we identified its lack of performance, which is clearly behind object-oriented approaches.

Keywords: Agent-Based Simulation, Functional Programming, Haskell, Concurrency, Parallelism, Property-Based Testing, Validation & Verification.

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 (Epstein and Axtell 1996) 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* (North and Macal 2007) which still holds up today.

In this paper we challenge this metaphor and present ways of implementing ABS with the functional programming paradigm using the language Haskell (Hudak, Hughes, Peyton Jones, and Wadler 2007). We

claim that functional programming has its place in ABS because of ABS' *scientific computing* nature where results need to be reproducible and correct while simulations should be able to be exploit parallelism and concurrency as well. The established object-oriented approaches need considerably high effort and might even fail to deliver these objectives due to its conceptually different approach to programming. In contrast, we claim that by using functional programming for implementing ABS it is less difficult to add parallelism and correct concurrency, the resulting simulations are easier to test and verify, guaranteed to be reproducible already at compile-time, have fewer potential sources of bugs and are ultimately more likely to be correct.

To substantiate our claims we present fundamental concepts and advanced features of functional programming and how they can be used to engineer clean, maintainable and reusable ABS implementations. Further we discuss how the well known benefits of functional programming in general are applicable in ABS. To put our claims to test we conducted a practical case-study in which we implemented the *full* SugarScape model (Epstein and Axtell 1996) in Haskell. In this case study:

- We developed techniques for engineering a clean, maintainable, general and reusable *sequential* implementation in Haskell. Those techniques are directly applicable to other ABS implementations as well due to the complex nature of the Sugarscape model.
- We explored ways of exploiting data-parallelism to speed up the execution in our sequential implementation.
- We explored techniques for implementing a *concurrent* implementation using Software Transactional Memory (STM).
- We conducted performance measurements of our sequential and concurrent implementation.
- We explored ways of code-testing our implementation. We show how to use property-based testing for ensuring the correctness of individual agent parts, and unit-testing of the whole simulation which serves both as regression test and to check model hypotheses.
- Our Sugarscape implementation is fully validated against the dynamics reported in the book (Epstein and Axtell 1996) and an already existing NetLogo replication (Weaver). Due to lack of space we discuss the validation process in Appendix A.

We present the challenges encountered in this case-study and discuss benefits and drawbacks. As will become apparent, our results support our claims that pure functional programming has indeed its place in ABS and that it has the mentioned benefits. On the other hand, due to its heavier approach in terms of engineering, its approach pays only off in cases of high-impact and large-scale simulations which results might have far-reaching consequences e.g. influence policy decisions. Because its only those models which require high confidence in correctness and stability of the software this heavier approach is almost never necessary for quick prototyping and small case-studies of ABS models for which NetLogo and the established object-oriented approaches using Python, Java, C++ are perfectly suitable. Still, we hope to distil the developed techniques into a general purpose ABS Haskell library so implementing models becomes much easier and quicker and makes using Haskell attractive for prototyping models as well.

The aim and contribution of this paper is to introduce the functional programming paradigm using Haskell to ABS on a *conceptual* level, identifying benefits, difficulties and drawbacks. This is done by presenting the above mentioned case-study which introduces general implementation techniques applicable to ABS. To the best of our knowledge, we are the first to do so.

The structure of the paper is as follows. In Section 2 we present related work on functional programming in ABS. In Section 3 we introduce the functional programming paradigm, establish key features, motivate why we chose Haskell, discuss its type system, side-effects, parallelism and concurrency and how ABS can be implemented on top of them building on Monads, Continuations and FRP. In Section 4 we present our Sugarscape case-study and with its challenges encountered and benefits and drawbacks. In Section

5 we discuss our initial claims in the light of the case-study. In Section 6 we conclude and point out further research. We added an Appendix A in which we give a deeper insight into our process of validating our Sugarscape model against the book and (Epstein and Axtell 1996) and an already existing NetLogo replication (Weaver).

2 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 (De Jong 2014, Sulzmann and Lam 2007, Jankovic and Such 2007).

A multi-method simulation library in Haskell called *Aivika 3* is described in the technical report (Sorokin 2015). It supports implementing Discrete Event Simulations (DES), System Dynamics and comes with basic features for event-driven ABS which is realised using DES under the hood. Further it provides functionality for adding GPSS to models and supports parallel and distributed simulations. It runs in IO for realising parallel and distributed simulation but also discusses generalising their approach to avoid running in IO.

In his masterthesis (Bezirgiannis 2013) the author investigated Haskells parallel and concurrency features to implement (amongst others) *HLogo*, a Haskell clone of the NetLogo simulation package, focusing on using Software Transactional Memory for a limited form of agent-interactions. *HLogo* is basically a re-implementation of NetLogos API in Haskell where agents run within IO and thus can also make use of STM functionality.

There exists some research (Di Stefano and Santoro 2005, Varela, Abalde, Castro, and Gulías 2004, Sher 2013) of using the functional programming language Erlang (Armstrong 2010) to implement ABS. The language is inspired by the actor model (Agha 1986) and was created in the 1986 by Joe Armstrong for Eriksson for developing distributed high reliability software in telecommunications. The actor model can be seen as quite influential to the development of the concept of agents in ABS which borrowed it from Multi Agent Systems (Wooldridge 2009). It emphasises message-passing concurrency with share-nothing semantics (no shared state between agents) which maps nicely to functional programming concepts. The mentioned papers investigate how the actor model can be used to close the conceptual gap between agent-specifications which focus on message-passing and their implementation. Further they also showed that using this kind of concurrency allows to overcome some problems of low level concurrent programming as well.

Using functional programming for DES was discussed in (Jankovic and Such 2007) 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 (Schneider, Dutchyn, and Osgood 2012, Vendrov, Dutchyn, and Osgood 2014). This language called FRABJOUS is human readable and easily understandable by domain-experts. It is not directly implemented in FRP/Haskell but is compiled to Haskell code which they claim is also readable. This supports that FRP is a suitable approach to implement ABS in Haskell. Unfortunately, the authors do not discuss their mapping of ABS to FRP on a technical level, which would be of most interest to functional programmers.

3 FUNCTIONAL PROGRAMMING

Functional programming (FP) is called *functional* because it makes functions the main concept of programming, promoting them to first-class citizens: functions can be assigned to variables, they can be passed as arguments to other functions and they can be returned as values from functions. The roots of FP lie in the

Lambda Calculus which was first described by Alonzo Church (Church 1936). This is a fundamentally different approach to computing than imperative programming (which includes established object-orientation) which roots lie in the Turing Machine (Turing 1937). Rather than describing *how* something is computed as in the more operational approach of the Turing Machine, due to the more *declarative* nature of the Lambda Calculus, code in functional programming describes *what* is computed.

In our research we are using the *pure* functional programming language Haskell. The paper of (Hudak, Hughes, Peyton Jones, and Wadler 2007) gives a comprehensive overview over the history of the language, how it developed and its features and is very interesting to read and get accustomed to the background of the language. The main points why we decided to go for Haskell are:

- Rich Feature-Set - it has all fundamental concepts of the pure functional programming paradigm of which we explain the most important below. Further, Haskell has influenced a large number of languages, underlining its importance and influence in programming language design.
- Real-World applications - the strength of Haskell has been proven through a vast amount of highly diverse real-world applications (Hudak, Hughes, Peyton Jones, and Wadler 2007), is applicable to a number of real-world problems (O'Sullivan, Goerzen, and Stewart 2008) and has a large number of libraries available [1].
- Modern - Haskell is constantly evolving through its community and adapting to keep up with the fast changing field of computer science. Further, the community is the main source of high-quality libraries.

3.1 Fundamentals

To explain the central concepts of functional programming, we give an implementation of the factorial function in Haskell:

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

When looking at this function we can identify the following:

1. Declarative - we describe *what* the factorial function is rather than how to compute it. This is supported by *pattern matching* which allows to give multiple equations for the same function, matching on its input.
2. Immutable data - in functional programming we don't have mutable variables - after a variable is assigned, it cannot change its contents. This also means that there is no destructive assignment operator which can re-assign values to a variable. To change values, we employ recursion.
3. Recursion - the function calls itself with a smaller argument and will eventually reach the case of 0. Recursion is the very meat of functional programming because they are the only way to implement loops in this paradigm due to immutable data.
4. Static Types - the first line indicates the name and the type of the function. In this case the function takes one Integer as input and returns an Integer as output. Types are static in Haskell which means that there can be no type-errors at run-time e.g. when one tries to cast one type into another because this is not supported by this kind of type-system.

[1] https://wiki.haskell.org/Applications_and_libraries

5. Explicit input and output - all data which are required and produced by the function have to be explicitly passed in and out of it. There exists no global mutable data whatsoever and data-flow is always explicit.
6. Referential transparency - calling this function with the same argument will *always* lead to the same result, meaning one can replace this function by its value. This means that when implementing this function one can not read from a file or open a connection to a server. This is also known as *purity* and is indicated in Haskell in the types which means that it is also guaranteed by the compiler.

It may seem that one runs into efficiency-problems in Haskell when using algorithms which are implemented in imperative languages through mutable data which allows in-place update of memory. The seminal work of (Okasaki 1999) showed that when approaching this problem with a functional mind-set this does not necessarily be the case. The author presents functional data structures which are asymptotically as efficient as the best imperative implementations and discusses the estimation of the complexity of lazy programs.

For an excellent and widely used introduction to programming in Haskell we refer to (Hutton 2016). Other, more exhaustive books on learning Haskell are (Lipovaca 2011, Allen and Moronuki 2016). For an introduction to programming with the Lambda-Calculus we refer to (Michaelson 2011). For more general discussion of functional programming we refer to (Hughes 1989, MacLennan 1990, Hudak, Hughes, Peyton Jones, and Wadler 2007).

3.2 Side-Effects

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

Obviously, to write real-world programs which interact with the outside-world we need side-effects. Haskell allows to indicate in the *type* of a function that it does or does *not* have side-effects. Further there are a broad range of different effect types available, to restrict the possible effects a function can have to only the required type. This is then ensured by the compiler which means that a program in which one tries to e.g. read a file in a function which only allows drawing random-numbers will fail to compile. Haskell also provides mechanisms to combine multiple effects e.g. one can define a function which can draw random-numbers and modify some global data. The most common side-effect types are: *IO* allows all kind of I/O related side-effects: reading/writing a file, creating threads, write to the standard output, read from the keyboard, opening network-connections, mutable references; *Rand* allows drawing random-numbers; Reader / Writer / State allows to read / write / both from / to an environment.

A function without any side-effect type is called *pure*, and the *factorial* function is indeed pure. Below we give an example of a function which is not pure. The *queryUser* function *constructs* a computation which, when executed asks the user for its user-name and compares it with a given user-configuration. In case the user-name matches it returns True, and False otherwise after printing a corresponding message.

```
queryUser :: String -> IO Bool
queryUser username = do
  -- print text to console
  putStr "Type in user-name: "
```

```
-- wait for user-input
str <- getLine
-- check if input matches user-name
if str == username
  then do
    putStrLn "Welcome!"
    return True
  else do
    putStrLn "Wrong user-name!"
    return False
```

The *IO* in the first line indicates that the function runs in the IO effect and can thus (amongst others) print to the console and read input from it. What seems striking is that this looks very much like imperative code - this is no accident and intended. When we are dealing with side-effects, ordering becomes important, thus Haskell introduced the so-called do-notation which emulates an imperative style of programming. Whereas in imperative programming languages like C, commands are chained or composed together using the ; operator, in functional programming this is done using function composition: feeding the output of a function directly into the next function. The machinery behind the do-notation does exactly this and desugars this imperative-style code basically into function compositions which run custom code between each line, depending on the type of effect the computation runs in (IO in this case). This approach of function composition with custom code in between each function allows to emulate a broad range of imperative-style effects, including the above mentioned ones. For a technical, in-depth discussion of the concept of side-effects and how they are implemented in Haskell using Monads, we refer to the following papers: (Moggi 1989, Wadler 1992, Wadler 1995, Wadler 1997, Jones 2002).

Although it might seem very restrictive at first, we get a number of benefits from making the type of effects we can use in the function explicit. First we can restrict the side-effects a function can have to a very specific type which is guaranteed at compile time. This means we can have much stronger guarantees about our program and the absence of potential errors already at compile-time which implies that we don't need test them with e.g. unit-tests. Second, because effect-runners are themselves *pure*, we can execute effectful functions in a very controlled way by making the effect-context explicit in the parameters to the effect-runner. This allows a much easier approach to isolated testing because the history of the system is made explicit.

Further, this type system allows Haskell to make a very clear distinction between parallelism and concurrency. Parallelism is always deterministic and thus pure without side-effects because although parallel code runs concurrently, it does by definition not interact with data of other threads. This can be indicated through types: we can run pure functions in parallel because for them it doesn't matter in which order they are executed, the result will always be the same due to the concept of referential transparency. Concurrency is potentially non-deterministic because of non-deterministic interactions of concurrently running threads through shared data. For a technical, in-depth discussion on Parallelism and Concurrency in Haskell we refer to the following books and papers: (Marlow 2013, O'Sullivan, Goerzen, and Stewart 2008, Harris, Marlow, Peyton-Jones, and Herlihy 2005, Marlow, Peyton Jones, and Singh 2009).

4 CASE-STUDY: PURE FUNCTIONAL SUGARSCAPE

To explore how to approach ABS based on pure functional programming concepts as introduced before, we did a *full and verified* implementation of the seminal Sugarscape model (Epstein and Axtell 1996). We chose the model because it is quite well known in the ABS community, it was highly influential in sparking the interest in ABS, it is quite complex with non-trivial agent-interactions and it used object-oriented techniques and explicitly advocates them as a good fit to ABS.

Our goal was first to develop techniques and concepts to show *how* to engineer a clean, maintainable and robust ABS in Haskell. The second step was then to identify benefits and drawbacks to identify *why* one would follow such an approach. In a third step we pushed the benefits of pure functional programming further and tried to find a remedy for the drawbacks. Absolutely paramount in our research was, that we are being *pure*, which avoids the IO effect type under all circumstances because we would practically lose all strong compile time guarantees [2].

TODO: (Macal 2016) sugarscape is level 4, chapter 2 is level 3 or 2?

4.1 A Functional View

Due to the fundamentally different approaches of functional programming (FP) an ABS needs to be implemented fundamentally different as well compared to established object-oriented (OO) approaches. We face the following challenges:

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

The fundamental building blocks to solve these problems are *recursion* and *continuations*. In recursion a function is defined in terms of itself: in the process of computing the output it *might* call itself with changed input data. Continuations in turn allow to encapsulate the execution state of a program including local variables and pick up computation from that point later on.

This allows us to define an agent as a function, the question is what its input and output are. Event-driven approach (Meyer 2014) Agent-agent interactions are trivial in object-orientation: one either makes a direct method call or send an event, mutating the internal state of the receiving agent. In functional programming we need to come up with alternatives because neither method-calls nor globally mutable state is available. TODO: derive the agent-interface, which is driven by agent-interactions

As output the agent returns a data-structure which holds all *observable* information which the agent wants to share with the outside world. Together with the continuation this guarantees that the agent is in full control over its local state, which no one can mutate or access from outside. This also implies that one can only get information out of the agent by running its function. It also means that the output type of the function has to cover all possible input cases - it cannot change or depend on the input.

The alternative are *synchronous* interactions which are necessary when an arbitrary number of interactions between two agents need to happen instantaneously without any time-steps in between. 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 (Epstein and Axtell 1996). In object-oriented programming, the concept of synchronous communication between agents is trivially implemented directly with method calls but it can get tricky to get right in an functional programming setting. The only option one has, is to dynamically find the target agents signal function and run it within the source agent. This would imply some effectful context which allows read/write to all signal functions in the system: we need to read it to find the target and write it to put the continuation back in because it has locally encapsulated state. This is active research we conduct at the moment and we leave this for further research as it is out of the scope of this paper.

[2] The code is freely accessible from https://github.com/thalerjonathan/phd/tree/master/public/towards/SugarScape

```
TODO: give a short example of continuation agent
```

From this example it becomes apparent that we can encapsulate local state which which is not accessible and mutable from outside but only through explicit inputs and outputs to the continuation.

Obviously the agents in the Sugarscape are located in a discrete 2d environment where they move around and harvest resources, which means the need to read and write data of environment. This is conveniently implemented by adding a State side-effect type to the agent continuation function. Further we also add a Random effect type because dynamics in most ABS in general and Sugarscapes in particular are driven by random number streams, so our agent needs to have access to one as well.

4.2 Code metrics

We used the command line tool *cloc* to count the lines of Haskell code we have written (ignoring comments, reporting only the 'code' values)

TODO: cite the book / paper (?) which report the metrics of the sugarscape implementation.

Count LoC of NetLogo (4.0.4, as 5.1 seemed to have bugs in some of their functionality): 2128 LoC in a single (!) file (Sugarscape.nlogo) Count LoC of Java implementation (http://sugarscape.sourceforge.net/): 6525 in 5 files Count LoC of Python (https://github.com/citizen-erased/sugarscape): 1109 in 9 files

Count LoC of my implementation - complete project: 4300 in 38 files - complete project without test-code 3660 in 27 files - test code: 635 in 11 files - simulation-core and infrastructure (no rendering): 1550 in 9 files - data-export: 70 in 1 file - visualisation: 200 in 2 files - agent-behaviour only: 1700 in 14 files

Big difference in our implementation - lots of lines are type-, import- and export (module) declarations. We conjecture that roughly 40% of the whole code consists of such declarations.

- several hundred lines are the scenario-definitions - what we provide in addition (netlogo does not need): simulation kernel, infrastructure, utilities, exporting of data, low-level rendering

4.3 Memory and Performance

Haskell is notorious for its space-leaks due to laziness. Even for simple programs one can be hit by a serious space-leak where unevaluated code pieces (thunks) builds up in memory until they are needed, leading to dramatically increased memory usage for a problem which should be solved using a fraction.

It is no surprise that our highly complex sugarscape implementation (TODO: what about our SIR implementation) suffered severely from space-leaks. In Simulation this is a big issue, threatening the value of the whole implementation despite its other benefits: because simulations might run for a (very) long time or conceptually forever, one must make absolutely sure that the memory usage stays constant.

Exactly this was violated in our sugarscape implementation where the memory usage increased linearly with about 40MByte per second! Haskell allows to add so-called Strict pargmas to code-modules which forces strict evaluation of all data even if it is not used. Carefully adding this conservatively file-by file and checking for changes in memory-leaks reduced the memory consumption considerably and also led to a substantial performance increase. Now only the environment data-structure left leaking.

4.4 Concurrency and parallelism

To see how difficult it was to build a concurrent implementation we took the existing sequential implementation and added concurrency to it using Software Transactional Memory (STM). The main idea behind STM is that instead of locking and synchronising access to shared data, STM executes code-blocks as atomic transactions which either commit successfully in case no dirty-read happened or retries in case the value was changed since its last read. Although STM exists in other languages as well, Haskells type-system guarantees that retries have no persisting side-effects, which is crucial for the retry-semantics of STM implementations. We have written a separate paper about using STM to implement concurrent ABS TODO cite my paper in TOMACS, thus we will not go into more detail here but refer to that paper instead.

4.5 Testing

We implemented a number of tests for agent functions which don't cover a whole sub-part of an agents behaviour: checks whether an agent has died of age, check whether an agent has starved to death, the metabolism, immunisation step, check if an agent is a potential borrower, check for fertility, lookout, trading transaction. What all these functions have in common is that they are not pure computations like utility functions but are already running within an agent-context which means they have access to the agent state, environment, simulation context and random-number stream. This makes testing harder because one needs to construct more complex simulation state and needs to run the agent-context with the provided states.

TODO: shortly describe property-based testing Property-Based works surprisingly well in this context because properties seem to be quite abound here. We simply implement data-generators for our agent state and environment and its cells and then let QuickCheck generate the random data and us running the agent with the provided data, checking for the properties. An example for such a property is that an agent has starved to death in case its sugar (or spice) level has dropped to 0. The corresponding property-test generates a random agent state and also a random sugar level which we set in the agent state. We then run the function which returns True in case the agent has starved to death. We can then check that this flag is true only iff the initial random sugar level was less then or equal 0. TODO: maybe explain fertility check or borrower check

This might not sound too exciting but this concept has tremendous potential with reaching consequences: it reliefs one from covering a myriad number of edge cases but shifts it towards writing data-generators and the reliance on QuickCheck to find them (which it does, unless the data is too complex). Also the nature of a property-test has more a specification character, shifting the testing nature more towards a declarative nature, where we test what something is or is not instead of a more operational approach in unit-testing were we test a known fixed input against an a priori known fixed output.

Due to the way Haskell deals with side-effects and separation of data and code in functional programming (which is both strength and weakness in oop / fp respectively), testing is quite straightforward because there are no implicit dependencies, everything is explicit. What is particularly powerful is that one has complete control and insight over the changed state before and after e.g. a function was called on an agent: thus it is very easy to check if the function just tested has changed the agent-state itself or the environment or other data provided to the agent through a Monad: the new environment is returned after running the agent and can be checked for equality of the initial one - if the environments are not the same, one simply lets the test fail. This behaviour is very hard to emulate in OOP because one can not exclude side-effect at compile time, which means that some implicit data-change might slip away unnoticed. In FP we get this for free.

One drawback though is that because the agents monad stack contains the random-number generator we also need to execute the Random Monad runner even if the respective function never makes use of the Random Number functionality - this is simply not possible to detect at compile time. In such a case it is no problem

to simply pass a default random number generator always initialised by a fixed seed. This might look more serious than it is, some functions only make use of the agent state, which they declare in their type: the monad they run in is only a state monad with the AgentState as state-type - this makes it easy to run using the state runner and also guarantees at compile time that no other effects can and will happen.

5 DISCUSSION

Probably the biggest strength is that we can guarantee reproducibility at compile time: given identical initial conditions, repeated runs of the simulation will lead to same outputs. This is of fundamental importance in simulation and addressed in the Sugarscape model: *"... when the sequence of random numbers is specified ex ante the model is deterministic. Stated yet another way, model output is invariant from run to run when all aspects of the model are kept constant including the stream of random numbers."* (page 28, footnote 16) - we can guarantee that in our pure functional approach already at compile time.

Refactoring is very convenient and quickly becomes the norm: guided by types (change / refine them) and relying on compiler to point out problems, results in very effective and quick changes without danger of bugs showing up at run-time. This is not possible in Python because of its lack of compiler and types, and much less effective in Java due to its dynamic type-system although through IDE support one arrives also at a highly strong refactoring performance there.

Adding data-parallelism is easy and often requires simply swapping out a data-structure or library function against its parallel version. Concurrency, although still hard, is less painful to address and add in a pure functional setting due to immutable data and explicit side-effects. Further, the benefits of implementing concurrent ABS based on STM has been shown (TODO: cite my own TOMACS paper) which underlines the strength of Haskell for concurrent ABS due to its strong guarantees about retry-semantics.

Testing in general allows much more control and checking of invariants due to the explicit handling of effects - together with the strong static type system, nothing slips the testing-code as everything is explicit. Property-based testing in particular is a perfect match to testing ABS due to the random nature in both and because it supports convenient expressing of specifications. Thus we can conclude that in a pure functional setting, testing is very expressive and powerful and supports working towards an implementation which is very likely to be correct.

The main drawback of our approach is performance, which at the moment does not come close to object-oriented implementations. TODO: why is this so? 1. FP is known for being slower due to higher level of abstractions which are bought by slower code in general 2. updates are the main bottleneck due to immutable data which requires to copy the whole (or subparts) of a datastructure in cases of a change. One way to address that problem is to add data-parallelism and concurrency. The first one is easily possible as already explained above, the use of concurrency we have addressed in our STM paper (TODO cite). Another potential improvement of this problem is the use of linear types (TODO: cite linear haskell paper), which allow to annotate a variable with how often it is used within a function. From this a compiler can derive aggressive optimisations, resulting in imperative-style performance but retaining the declarative nature of the code.

6 CONCLUSIONS

Our results support the claim that pure functional programming has indeed its place in ABS but only in cases of high-impact and large-scale simulations which results might have far-reaching consequences e.g. influence policy decisions. The reason is that engineering a proper implementation of a non-trivial ABS model takes substantial effort in pure functional programming due to different approaches. This is almost

never necessary for quick prototyping and small case-studies of ABS models for which NetLogo and the established object-oriented approaches using Python, Java, C++ are perfectly suitable.

TODO: a bit more conclusion This approach can support TOMACS Reproducibility Board

6.1 Further Research

We plan on distilling the developed techniques of the case-study into a general purpose ABS library. This should make implementing models much easier and quicker and using Haskell attractive for prototyping models as well.

Often, ABS models build on an underlying DES core, especially when they follow an event-driven approach (Meyer 2014) where they need to schedule actions into the future. Due to the time-driven approach of the Sugarscape model, we didn't implement this in our case-study but our pure functional simulation core is conceptually already very close to a DES approach. We plan on extending it to allow true event-based ABS, which we want to feed into our general purpose ABS library as well.

Due to the recursive nature of functional programming we believe that it is also a natural fit to implement recursive simulations as the one discussed in (Gilmer and Sullivan 2000). In recursive ABS agents are able to halt time and 'play through' an arbitrary number of actions, compare their outcome and then to resume time and continue with a specifically chosen action e.g. the best performing or the one in which they haven't died. More precisely, an agent has the ability to run the simulation recursively a number of times where the this number is not determined initially but can depend on the outcome of the recursive simulation. So recursive ABS gives each Agent the ability to run the simulation locally from its point of view to anticipate its actions in the future and change them in the present. Due to the lack of implicit side-effects and referential transparency, combined with the recursive nature of pure functional programming, we think that implementing a recursive simulation in such a setting should be straight-forward.

This research is only a first step towards an even more extensive use of type systems in ABS. The next step is to investigate the use of dependent types in ABS, using the pure functional language Idris. Dependent types allow to express even stronger guarantees at compile time, going as far as programs being valid proofs for the correctness of the implemented feature.

ACKNOWLEDGMENTS

The authors would like to thank

REFERENCES

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

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

Armstrong, J. 2010, September. "Erlang". *Commun. ACM* vol. 53 (9), pp. 68–75.

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

Church, A. 1936, April. "An Unsolvable Problem of Elementary Number Theory". *American Journal of Mathematics* vol. 58 (2), pp. 345–363.

De Jong, T. 2014. "Suitability of Haskell for Multi-Agent Systems". Technical report, University of Twente.

Di Stefano, A., and C. Santoro. 2005. "Using the Erlang Language for Multi-Agent Systems Implementation". In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, IAT '05, pp. 679–685. Washington, DC, USA, IEEE Computer Society.

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

Gilmer, Jr., J. B., and F. J. Sullivan. 2000. "Recursive Simulation to Aid Models of Decision Making". In *Proceedings of the 32Nd Conference on Winter Simulation*, WSC '00, pp. 958–963. San Diego, CA, USA, Society for Computer Simulation International.

Harris, T., S. Marlow, S. Peyton-Jones, and M. Herlihy. 2005. "Composable Memory Transactions". In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pp. 48–60. New York, NY, USA, ACM.

Hudak, P., J. Hughes, S. Peyton Jones, and P. 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, pp. 12–1–12–55. New York, NY, USA, ACM.

Hughes, J. 1989, April. "Why Functional Programming Matters". *Comput. J.* vol. 32 (2), pp. 98–107.

Hutton, G. 2016, August. *Programming in Haskell*. Cambridge University Press. Google-Books-ID: 1xHPDAAAQBAJ.

Jankovic, P., and O. Such. 2007. "Functional Programming and Discrete Simulation". Technical report.

Jones, S. P. 2002. "Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell". In *Engineering theories of software construction*, pp. 47–96, Press.

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

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

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

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

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

Meyer, R. 2014, May. "Event-Driven Multi-agent Simulation". In *Multi-Agent-Based Simulation XV*, Lecture Notes in Computer Science, pp. 3–16, Springer, Cham.

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

Moggi, E. 1989. "Computational Lambda-calculus and Monads". In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pp. 14–23. Piscataway, NJ, USA, IEEE Press.

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

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

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

Schneider, O., C. Dutchyn, and N. Osgood. 2012. "Towards Frabjous: A Two-level System for Functional Reactive Agent-based Epidemic Simulation". In *Proceedings of the 2Nd ACM SIGHIT International Health Informatics Symposium*, IHI '12, pp. 785–790. New York, NY, USA, ACM.

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

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

Sulzmann, M., and E. Lam. 2007. "Specifying and Controlling Agents in Haskell". Technical report.

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

Varela, C., C. Abalde, L. Castro, and J. Gulías. 2004. "On Modelling Agent Systems with Erlang". In *Proceedings of the 2004 ACM SIGPLAN Workshop on Erlang*, ERLANG '04, pp. 65–70. New York, NY, USA, ACM.

Vendrov, I., C. Dutchyn, and N. D. Osgood. 2014, April. "Frabjous A Declarative Domain-Specific Language for Agent-Based Modeling". In *Social Computing, Behavioral-Cultural Modeling and Prediction*, edited by W. G. Kennedy, N. Agarwal, and S. J. Yang, Number 8393 in Lecture Notes in Computer Science, pp. 385–392. Springer International Publishing.

Wadler, P. 1992. "The Essence of Functional Programming". In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pp. 1–14. New York, NY, USA, ACM.

Wadler, P. 1995. "Monads for Functional Programming". In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pp. 24–52. London, UK, UK, Springer-Verlag.

Wadler, P. 1997, September. "How to Declare an Imperative". *ACM Comput. Surv.* vol. 29 (3), pp. 240–263.

Weaver, I. "Replicating Sugarscape in NetLogo". Technical report.

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

A VALIDATING SUGARSCAPE IN HASKELL

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

Running multiple replications of the same simulation in parallel is extremely easy in functional programming: because each simulation is independent from each other, it is a case of data-parallelism which means that each can run independently in parallel, without the need to change the types. To run e.g. 100 replications just requires to replace a single function call by a different function which runs them all in parallel. Also the testing library we used (Tasty) supports running tests in parallel out of the box without danger of any side-effects interfering with each other. Of course both parallelisms are possible in traditional OOP approaches and if the programmer has done his or her job right there should be no problem but the important message here is that: 1. haskell can guarantee that no interference will occur already at compile time and 2. it does support the parallelisation on a language level without the pain of low level thread management or locks.

A.1 Terracing

Our implementation reproduce the terracing phenomenon as described on page TODO in Animation and as can be seen in the NetLogo implementation as well. We implemented a property-test in which we measure the closeness of agents to the ridge: counting the number of same-level sugarscells around them and if there is at least one lower then they are at the edge. If a certain percentage is at the edge then we accept terracing. The question is just how much, this we estimated from tests and resulted in 45%. Also, in the terracing animation the agents actually never move which is because sugar immediately grows back thus there is no need for an agent to actually move after it has moved to the nearest largest cite in can see. Therefore we test that the coordinates of the agents after 50 steps are the same for the remaining steps.

A.2 Carrying Capacity

Our simulation reached a steady state (variance < 4 after 100 steps) with a mean around 182. Epstein reported a carrying capacity of 224 (page 30) and the NetLogo implementations carrying capacity fluctuates around 205 which both were thus significantly higher than ours. Something was definitely wrong - the carrying capacity has to be around 200 (we trust in this case the NetLogo implementation and deem 224 an outlier).

After inspection of the netlogo model we realised that we implicitly assumed that the metabolism range is *continuously* uniformly randomized between 1 and 4 but this seemed not what the original authors intended: in the netlogo model there were a few agents surviving on sugarlevel 1 which was never the case in ours as the probability of drawing a metabolism of exactly 1 is 0 when drawing from a continuous range. We thus changed our implementation to draw discrete. Note that this actually makes sense as massive floating-point number calculations were quite expensive in the mid 90s (e.g. computer games ran still on CPU only and exploited various clever tricks to avoid the need of floating point calculations whenever possible) when SugarScape was implemented which might have been a reason for the authors to assume it implicitly.

This partly solved the problem, the carrying capacity was now around 204 which is much better than 182 but still a far cry from 210 or even 224. After adjusting the order in which agents apply the sugarscape rules, (by looking at the code of the netlogo implementation), we arrived at a comparable carrying capacity of the netlogo implementation: agents first make their move and harvest sugar and only after this the agents metabolism is applied (and ageing in subsequent experiments).

For regression-tests we implemented a property-test which tests that the carrying capacity of 100 simulation runs lies within a 95% confidence interval of a 210 mean. TODO: variance test. These values are quite reasonable to assume, when looking at NetLogo - again we deem the reported Carrying Capacity of 224 in the Book to be an outlier / part of other details we don't know.

TODO: do a replication experiment with NetLogo (is it possible?)

A.3 Wealth Distribution

By visual comparison we validated that the wealth distribution (page 32-37) becomes strongly skewed with a Histogram showing a fat tail, power-law distribution where very few agents are very rich and most of the agents are quite poor. We compute the skewness and kurtosis of the distribution which is around a skewness of 1.5, clearly indicating a right skewed distribution and a kurtosis which is around 2.0 which clearly indicates the lst histogram of Animation II-3 on page 34. Also we compute the gini-coefficient and it varies between 0.47 and 0.5 - this is accordance with Animation II-4 on page 38 which shows a gini-coefficient which stabilises around 0.5 after. We implemented a regression-test testing skewness, kurtosis and gini-coefficients of 100 runs to be within a 95% confidence interval of a two-sided t-test using an expected skewness of 1.5, kurtosis of 2.0 and gini-coefficient of 0.48.

A.4 Migration

With the information provided by (Weaver) we could replicate the waves as visible in the NetLogo implementation as well. Also we propose that a vision of 10 is not enough yet and shall be increased to 15 which makes the waves very prominent and keeps them up for much longer - agent waves are travelling back and forth between both sugarscape peaks. We haven't implemented a regression-test for this property as we couldn't come up with a reasonable straight forward approach to implement it.

A.5 Polution and Diffusion

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

Note that we spent quite a lot of time of getting this and the terracing properties right because they form the very basics of the other ones which follow so we had to be sure that those were correct otherwise validating would have been much more difficult.

A.6 Order of Rules

order in which rules are applied is not specified and might have an impact on dynamics e.g. when does the agent mate with others: is it after it has harvested but before metabolism kicks in?

A.7 Mating

Could not replicate figureIII-1, our dynamics first raise and then plunge to about 100 agents and go then on to recover and fluctuate around 300. This findings are in accordance with (Weaver), where they report similar findings - also when running their NetLogo code we find the dynamics to be qualitatively the same.

Cycles of population sizes not able to reproduce at first. Then we realised that our agent-behaviour was not correct: agents which died from age or metabolism could still engage in mating before actually dying - fixing this to the behaviour that agents which died from age or metabolism wont engage in mating solved that and produces the same swings as in (Weaver). Although our bug was probably quite obvious, the lack of specification of the order of the application of the rules is an issue in the SugarScape book. TODO: does it really have that much of an influence?

A.8 Inheritance

We couldnt replicate the findings of the Sugarscape Book regarding the gini Coefficient with inheritance. The authors report that they reach a gini coefficient of 0.7 and above in Animation III-4. Our gini coefficient fluctuated around 0.35. Compared to the same configuration but without inheritance - Animation III-1 - which reached a gini coefficient of about 0.21, this is indeed a substantial increase - also with inheritance we reach a larger number of agents of around 1000 as compared to around 300 without inheritance. The sugarscape book compares this to chapter II Animation II-4 for which they report a gini coefficient of around 0.5 which we could reproduce as well. TODO: why is it then lower (lower inequality) with inheritance?

The baseline is that this shows that inheritance indeed has an influence on the inequality in a population. Thus we deemed that our results are qualitatively the same as the make the same point. Still there must be some mechanisms going on behind the scenes which are unspecified in the original sugarscape.

A.9 Cultural Dynamics

We could replicated the cultural dynamics of AnimationIII-6 / Figure III-8: after 2700 steps either one culture (red / blue) dominates both hills or each hill is dominated by different a culture. We wrote a test for it in which we run the simulation for 2.700 steps and then check if either culture dominates with a ratio of 95% or if they are equal dominant with 45%. Because always a few agents stay stationary on sugarlevel 1 (they have a metabolism of 1 and cant see far enough to move towards the hills, thus stay always on same spot because no improvement and grow back to 1 after 1 step), there are a few agents which never participate in the cultural process and thus no complete convergence can happen. This is accordance with (Weaver).

A.10 Combat

Unfortunately (Weaver) didn't implement combat, so we couldn't compare it to their dynamics. Unfortunately we weren't able to replicate the dynamics found in the sugarscape book: the two tribes always formed a clear battlefront where some agents engage in combat e.g. when one single agent strays too far from its tribe and comes into vision of the other tribe it will be killed almost always immediately. This is because crossing the sugar valley is costly: this agent wont harvest as much as the agents staying on their hill thus will be less wealthy and thus easily killed off. Also retaliation is not possible without any of its own tribe anywhere near. We didn't see a single run where an agent of an opposite tribe "invaded" the other tribes hill and ran havoc killing off the entire tribe. We dont see how this can happen: the two tribes start in opposite corners and quickly occupy the respective sugar hills. So both tribes are acting on average the same and also

because of the number of agents no single agent can gather extreme amounts of wealth - the wealth should rise in both tribes equally on average. Thus it is very unlikely that a super-wealthy agent emerges, which makes the transition to the other side and starts killing off agents at large. First: a super-wealthy agent is unlikely to emerge, second making the transition to the other side is costly and also low probability, third the other tribe is quite wealthy as well having harvested for the same time the sugar hill, thus it might be that the agent might kill a few but the closer it gets to the center of the tribe the less like is a kill due to retaliation avoidance - the agent will be simply killed itself.

Also it is unclear in case of AnimationIII-11 if the R rule also applies to agents which get killed in combat. Nothing in the book makes this clear and we left it untouched so that agents who only die from age (original R rule) are replaced. This will lead to a near-extinction of the whole population quite quickly as agents kill each other off until 1 single agent is left which will never get killed in combat because there are no other agents who could kill it - instead it will die and get reborn infinitely thanks to the R rule.

A.11 Spice

The book specifies for AnimationIV-1 vision between 1-10 and a metabolism between 1-5. The last one seems to be quite strange because the maximum sugar / spice an agent can find is 4 which means that agents with metabolism of either 5 will die no matter what they do because the can never harvest enough to satisfy their metabolism. When running our implementation with this configuration the number of agents quickly drops from 400 to 105 and continues to slowly degrade below 90 after around 1000 steps. The implementation of (Weaver) used a slightly different configuration for AnimationIV-1, where they set vision to 1-6 and metabolism to 1-4. Their dynamics stabelise to 97 agents after around 500+ steps. When we use the same configuration as theirs, we produce the same dynamics. Also it is worth nothing that our visual output is strikingly similar to both the book AnimationIV-1 and (Weaver).

A.12 Trading

For trading we had a look at the NetLogo implementation of (Weaver): there an agent engages in trading with its neighbours *over multiple rounds* until MRSs cross over and no trade has happened anymore TODO: be more specific. Because (Weaver) were able to exactly replicate the dynamics of the trading time-series we assume that their implementation is correct. Unfortunately we think that the fact that an agent interact with its neighbours over multiple rounds is made not very clear in the book. The only hint is found on page 102 "This process is repeated until no further gains from trades are possible." which is not very clear and does not specify exactly what is going on: does the agent engage with all neighbours again? is the ordering random? Another hint is found on page 105 where trading is to be stoped after MRS cross-over to prevent infinite loop. Unfortunately this is missing in the Agent trade rule T on page 105. Additional information on this is found in footnote 23 on page 107. Further on page 107: "If exchange of the commodities will not cause the agents' MRSs to cross over then the transaction occurs, the agents recompute their MRSs, and bargaining begins anew.". This is probably the clearest hint that trading could occur over multiple rounds.

We still managed to exactly replicate the trading-dynamics as shown in the book in Figure IV-3, Figure IV-4 and Figure IV-5. The book is also pretty specific on the dynamics of the trading-prices standard-deviation: on page 109 the authors specify that at t=1000 the std will have always fallen below 0.05 (Figure IV-5), thus we implemented a property test which tests for exactly that property and the test passed. Unfortunately we didn't reach the same magnitude of the trading volume where ours is much lower around 50 but it is equally erratic, so we attribute these differences to other missing specifications or different measurements because the price-dynamics match that well already so we can safely assume that our trading implementation is correct.

According to the book, Carrying Capacity (Animation II-2) is increased by Trade (page 111/112). To check this it is important to compare it not against AnimationII-2 but a variation of the configuration for it where spice is enabled, otherwise the results are not comparable because carrying capacity changes substantially when spice is on the environment and trade turned off. We could replicate the findings of the book: the carrying capacity increases slightly when trading is turned on. Also does the average vision decrease and the average metabolism increase. This makes perfect sense: trading allows genetically weaker agents to survive which results in a slightly higher carrying capacity but shows a weaker genetic performance of the population.

According to the book, increasing the agent vision leads to a faster convergence towards the (near) equilibrium price (page 117/118/119, Figure IV-8 and Figure IV-9). We could replicate this behaviour as well.

According to the book, when enabling R rule and giving agents a finite life span between 60 and 100 will lead to price dispersion: the trading prices won't converge around the equilibrium and the standard deviation will fluctuate wildly (page 120, Figure IV-10 and Figure IV-11). We could replicate this behaviour as well.

The gini coefficient should be higher when trading is enabled (page 122, Figure IV-13) - We could replicate this behaviour.

Finite Lives with sexual reproduction lead to prices which dont converge (page 123, Figure IV-14). We could reproduce this as well but it was important to re-set the parameters to reasonable values: increasing number of agents from 200 to 400, metabolism to 1-4 and vision to 1-6, most important the initial endowments back to 5-25 (both sugar and spice) otherwise hardly any mating would happen because need too much wealth to engage. What was kind of interesting is that in this scenario the trading volume of sugar is substantially higher than the spice volume - about 3 times as high.

We didn't implement Effect of Culturally Varying Preferences, page 124 - 126 Externalities and Price Disequilibrium: The effect of Pollution, page 126 - 118 On The Evolution of Foresight page 129 / 130

A.13 Lending (Credit)

Not really much information to validate was available and the (Weaver) implementation ran into an exception so there was not much to validate against. What was unexpected was that this was the most complex behaviour to implement, with lots of subtle details to take care of (spice on/off, inheritance,...). Note that we implemented lending of sugar and spice, although it looks from the book (Animation IV-5) that they only implemented it for sugar.

A.14 Diseases

We were able to exactly replicate the behaviour of Animation V-1 and Animation V-2: in the first case the population rids itself of all diseases (maximum 10) which happens pretty quickly, in less than 100 ticks. In the second case the population fails to do so because of the much larger number of diseases (25) in circulation. We used the same parameters as in the book. The authors of (Weaver) could only replicate the first animation exactly and the second was only deemed "good". Their implementation differs slightly from ours: In their case a disease can be passed to an agent who is immune to it - this is not possible in ours. In their case if an agent has already the disease, the transmitting agent selects a new disease, the other agent has not yet - this is not the case in our implementation and we think this is unreasonable to follow: it would require too much information and is also unrealistic. We wrote regression tests which check for animation V-1 that after 100 ticks there are no more infected agents and for animation V-2 that after 1000 ticks there are still infected agents left and they dominate: more infected than recovered.