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

TODO

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:

©2019 Society for Modeling & Simulation International (SCS)

- 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 TODO.

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 TODO RELATED WORK we present related work on functional programming in ABS. In Section TODO FUNCTIONAL PROGRAMMING 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 TODO CASE-STUDY we present our Sugarscape case-study and with its challenges encountered and benefits and drawbacks. In Section TODO DISCUSSION we discuss our initial claims in the light of the case-study. In Section TODO CONCLUSIONS we conclude and point out further research. We added an Appendix TODO APPENDIX 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 ESTABLISHED APPROACHES

2.1 RePast Java

TODO

2.2 NetLogo

TODO: NetLogo: language looks and feels a bit like a functional language (declarative) but works fundamentally through side-effects. Large and complex models become very difficult to maintain because only one file.

2.3 AnyLogic

TODO

2.4 Java, C++, Python

TODO

3 CASE-STUDY: PURE FUNCTIONAL SUGARSCAPE

TODO

TODO: also SIR is a case-study

why sugarscape - original sugarscape sparked ABS and use of OOP, therefore - quite complex model, will challenge implementation techniques

[1]

(Weaver)

page 28, footnote 16: we can guarantee that in haskell at compile time

TODO: investigate where data-parallelisation is possible. concurrency has been dealt with in the STM paper already.

4 CHAPTER II

each agent is a Signal Function with no input and outputs an AgentOut which contains a list of agents it wants to spawn, a flag if the agent is to be removed from the simulation (e.g. starved to death) and observable properties the agent exhibits to the outside world. All the agents properties are encapsulated in the SF continuation and there is no way to access and manipulate the data from outside without running the SF itself which will produce an AgentOut.

An agent has access to the shared environment state, a random-number generator and a shared ABS-system state which contains the next agent-id when birthing a new agent. All this is implemented by sharing the data-structures amongst all agents which can read/write it - this is possible in functional programming using Monadic Programming which can simulate a global state, accessible from within a function which can then read/write this state. The fundamental difference to imperative oop-programming is that all reads / writes are explicit using functions (no assignment).

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

Updating the agents is straight-forward because in this chapter, the agents interact with each other indirectly through the environment. In each step the agents are shuffled and updated one after another, where agents can see actions of agents updated before.

Our approach of sharing the environment globally and the agent-state locally works but immediately creates potential problems like: ordering of updates matter - in the end we are implementing a kind of an imperative approach but embedded in a functional language. The benefits are that we have much stronger type-safety and that the access and modification of the states is much more explicit than in imperative approaches - also we dont have mutable references.

We implemented a different approach to iterating: instead of running the agents one after another and interacting through a globally shared environment all agents are now run *conceptually* at the same time and receive the current environment as additional input and have to provide it in the output. This has the following implications: we end up with n copies of the environment where n is the number of agents, agents are not able to see the actions of others until the next step, there can be conflicts where multiple agents end up on the same position. Obviously, positional conflicts need to be solved as the sugarscape specification clearly states that only one agent stays on a site at a time. Functional programming makes solving such conflicts easy: we pick a winning agent and rollback the other agents by re-running them with their SF at the beginning of the step - this will undo all changes within the encapsulation. Obviously it would be possible to have conflicts again thus one needs to recursively run the conflict-resolving process until no more conflicts are present. Although this solution is much slower and more complex to implement and thus not feasible to use in practice but we wanted to explore it for the following reasons: - it is "closer" to functional programming in spirit because programming with globally mutable state (even if its restricted, explicit and only simulated) should be avoided as far as possible. - we can exploit data-parallelism (but in this case its not possible anyway because of monadic computations: need mapM which can by definition not be parallel because ordering matters) - it serves more as a study to what different approaches are possible and how difficult / easy it is to implement them in FP, in this case, "rolling back" the actions of an agent is trivial in FP as long as the underlying monadic context is immune to rollbacks, in our case we argue that it is: incrementing agentids in ABSState does not matter, as it doesnt matter that we have a changed random-number stream. It would be a different matter if there is a global shared state which was modified by the agent. - in the extreme case this degenerates to a (much more expensive) sequential update

5 CHAPTER III

This chapter reveals the fundamental difference and difficulty in pure functional programming over established OOP approaches in the field: direct agent-interaction e.g. in mating where 2 agents interact synchronously with each other and might updated their internal state. These interactions *must* happen synchronously because there are resource constraints in place which could be violated if an agent interacts with multiple agents virtually at the same time.

In established OOP approaches this is nearly trivial and straight forward: the agent which initiates the direct interaction holds or looks up (e.g. through a central simulation management object) a reference to the other agent (e.g. by neighbourhood) and then makes direct method calls to the other agent where internal agent-states of both agents may be mutated. This approach is not possible in pure functional programming because: 1. there are no objects which encapsulate state and behaviour and 2. there are not side-effects possible which would allow such a mutation of local state [2].

[2] Relaxing our constraint by also allowing *impure* functional features so we can workaround the limitation of not being able to locally mutate state but this is not what we are interested in in this paper because we lose all relevant guarantees which make FP relevant and of benefit.

This makes implementation of direct agent-interactions utterly difficult. If we build on the approach we used for Chapter II (and which worked very well there!) we quickly run into painful problems:

- To mutate local agent state or to generate an output / seeing local properties requires to run the SF.
- Running the SF is intrinsically linked in stepping the simulation forward from t to t+1. Currently the agent has no means to distinguish between different reasons why the SF is being run.
- The agents are run after another (after being shuffled) and cannot make invokations of other agents SF during being executed due to pure functional programming.

A solution is to change to an event-driven approach: SF now have an input, which indicates an EventType and Agents need some way of initiating a multi-step interaction where a reply can lead to a new event and so on. In case of a simple time-advancement the SF is run with a "TimeStep" event, if an agent requests mating, then it sends "MatingRequest" to the other SF. This requires a completely different approach to iterating the agents.

Stateful programming (or programming that *feels* stateful) comes inherently with difficulties where one can forget to update a state or mutate state where not appropriate. A pure functional approach to that is no exception and shows the same problems. In our case we ran into a bug where the trading agent saw an outdated MRS value of the trading-partner resulting into two different trading-prices which obviously must be prevented under all circumstances because it would destroy / create wealth. The origin of the bug was that MRS depends on the wealth (sugar and spice) of the agent and we simply forgot to update the MRS in the environment from which the offering agent can read it when the trading agents wealth changed (e.g. through harvesting, inheritance,...).

explain continuation, explain monads = replacement of ; operator, runs custom (depending on monad) Code between evaluations

5.1 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 severeyl by space-leaks. In Simulation this is a severe 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. This

We found that the crucial files / modules were: initialisation, environment data-structure handling, simulation model data-structure, simulation core. What was particularly interesting was that when we added it to our initialisation module where the whole sugarscape model is constructed (agents and environment) it led to a huge improvement of memory-leaks and performance, so it seems to be necessary and quite beneficial to force strictness / evaluation for initialisation for a smooth running simulation.

Init.hs -> Major Common.hs -> Major Discrete.hs -> Minor Model.hs -> Minor Simulation.hs -> Minor

After fixing the memory-leaks we get a very low level memory consumption - depending on number of agents is around 3 MB in case of 250 Agents in Animation III-1. What is interesting is that the concurrent implementation consistently uses less memory than the sequential one with the Animation III-1 using up around only 2 MB.

TODO: performance comparison with netlogo implementation TODO: laziness can save Performance: laziness vs strictness

5.2 Concurrency

Although concurrent programming in general is hard, Haskell takes much of the difficulties out through its functional nature and its strong static type system. Because of its referential transparency it is easy to guarantee that no concurrent modification of state will happen (unless running in IO). Also through the type system it is possible to indicate that concurrent computations might or might not happen: also being clear about difference between parallelism and concurrency in types is possible: parallel computations run in parallel and do NOT interfere with each other e.g. through synchronisation or data-dependencies / data-mutation. Concurrent computations run in parallel but might interfere with each other through synchronisation primitives and shared data. Haskell allows to distinguish between these two types of computations in its type-system: a parallel computation is always deterministic and thus pure / referential transparent. Concurrency is indicated using IO or STM.

5.2.1 Getting it right

There were a few subtle bugs in my implementation as getting a concurrent implementation right is still hard even when using Haskell. Still Haskells type system and lack of effects helps a lot when reasoning about concurrent behaviour and also the run-time provides amazing help. For example will the program terminate with an exception when a thread blocks on a synchronisation primitive (e.g. MVar) which no other thread references - this is an example for a classic deadlock which cannot be recovered. It is highly beneficial that Haskell actually detects such deadlocks which would be quite difficult to detected without such facilities and in many other languages one would simply end up with infinitely hanging threads.

https://www.fpcomplete.com/blog/2018/05/pinpointing-deadlocks-in-haskell

6 CODE

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

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

7 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.

7.1 Further Research

- distil the developed techniques into a general purpose ABS library so implementing models becomes much easier and quicker and makes using Haskell attractive for prototyping models as well.

- distributed - DES and PDES - gintis-case study - recursive simulation

ACKNOWLEDGMENTS

The authors would like to thank

REFERENCES

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

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.

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.

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

A VALIDATING SUGARSCAPE IN HASKELL

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.