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 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 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 (**?**, **?**, **?**).

A multi-method simulation library in Haskell called *Aivika 3* is described in the technical report (**?**). 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 (**?**) 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 (**?**, **?**, **?**) of using the functional programming language Erlang (**?**) to implement ABS. The language is inspired by the actor model (**?**) 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 (**?**). 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 (**?**) 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 (**?**, **?**). 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

The reason why functional programming is called *functional* is because because it makes functions the main concept of programming, promoting them to first-class citizens as we will describe later on. Its roots lie in the Lambda Calculus which was first described by Alonzo Church church$_{u}nsolvable_{1}936.Thisisafundamentallydifferentap$
$orientedprogrammingwhichrootslieintheTuringMachineturing_{c}omputable_{1}937.Ratherthandescribinghowsomethingiscom$

In our research we are using the functional programming language Haskell. The paper of hudak$_{h}istory_{2}007givesacomprehens$

- Rich Feature-Set - it has all fundamental concepts of the pure functional programming paradigm of which we explain the most important below.
- 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 (**?**) 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.

As a short example 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 already see the central concepts of functional programming:

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 types 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.
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 (**?**) showed that when approaching this problem from a functional mind-set this does not necessarily be the case. The author presents functional data structures which are asymptotically as efficient as the best imperative implementations and discusses the estimation of the complexity of lazy programs.

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

For an excellent and widely used introduction to programming in Haskell we refer to (**?**). Other, more exhaustive books on learning Haskell are (**?**, **?**). For an introduction to programming with the Lambda-Calculus we refer to (**?**). For more general discussion of functional programming we refer to (**?**, **?**, **?**).

3.1 Lazy evaluation, Higher Order Functions and Currying

TODO:

3.2 Side-Effects

One of the fundamental strengths of functional programming and Haskell is their 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 the behaviour it 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 to draw random-numbers.
- Reader - Allows to read from an environment.
- Writer - Allows to write to an environment.
- State - Allows to read and write an environment.

A function with side-effects has to indicate this in their type e.g. if we want to give our factorial function for debugging purposes the ability to write to the standard output, we add IO to its type: factorial :: Integer -> IO Integer. A function without any side-effect type is called *pure*. A function with a given effect-type needs to be executed with a given effect-runner which takes all necessary parameters depending on the effect and runs a given effectful function returning its return value and depending on the effect also an effect-related result. For example when running a function with a State-effect one needs to specify the initial environment which can be read and written. After running such a function with a State-effect the effect-runner returns the changed environment in addition with the return value of the function itself. Note that we cannot call functions of different effect-types from a function with another effect-type, which would violate the guarantees. Calling a *pure* function though is always allowed because it has by definition no side-effects. An effect-runner itself is a *pure* function. The exception to this is the IO effect type which does not have a runner but originates from the *main* function which is always of type IO.

Although it might seem very restrictive at first, we get a number of benefits from making the type of effects we can use 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.

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: (**?**, **?**, **?**, **?**, **?**).

3.3 Parallelism and Concurrency

TODO: write this section

Also Haskell makes 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. Although data in functional programming is immutable, Haskell provides primitives which allow to share immutable data between threads. Accessing these primitives is but only possible from within an IO or STM context which means that when we are using concurrency in our program, the types of our functions change from pure to either IO or STM effect context.

Spawning thousands of threads in Haskell is no problem and has very low memory footprint because they are lightweight user-space threads, managed by the Haskell Runtime System which maps them to physical operating-system threads.

TODO: in haskell we can distinguish between parallelism and concurrency in the types: parallelism is pure, concurrency is impure TODO: parallelism for free because all isolated e.g. running multiple replications or parameter-variations

TODO: For a technical, in-depth discussion on parallelism and concurrency in Haskell we refer to the excellent book (**?**).

TODO: explain STM, Problem: live locks, For a technical, in-depth discussion on Software Transactional Memory in Haskell we refer to the following papers: harris$_c omposable_2 005, discolo_l ock_2 006, osullivan_r eal_2 008, perfumo_l in$
$needmuchmorepapersonSTM, parallelismandconcurrency$

3.4 Functional Reactive Programming

Functional Reactive Programming (FRP) is a way to implement systems with continuous and discrete time-semantics in functional programming. The central concept in FRP is the Signal Function which can be understood as a process over time which maps an input- to an output-signal. A signal in turn, can be understood as a value which varies over time. Thus, signal functions have an awareness of the passing of time by having access to a Δt which are positive time-steps with which the system is sampled. In general, signal functions can be understood to be computations that represent processes, which have an input of a specific type, process it and output a new type. Note that this is an important building block to represent agents in functional programming: by implementing agents as signal functions allows us to implement them as processes which act continuously over time, which implies a time-driven approach to ABS. We have also applied the concept of FRP to event-driven ABS meyer$_e vent - driven_2 014$.

FRP provides a number of functions for expressing time-semantics, generating events and making state-changes of the system. They allow to change system behaviour in case of events, run signal functions, generate deterministic (after fixed time) and stochastic (exponential arrival rate) events and provide random-number streams.

TODO: libraries Yampa and Dunai

For a technical, in-depth discussion on FRP in Haskell we refer to the following papers: wan$_f$*unctional*$_2$000, *hughes*$_g$*eneralisi*

3.5 Property-Based Testing

TODO: write this section

Although property-based testing has been brought to non-functional languages like Java and Python as well, it has its origins in Haskell and it is here where it truly shines.

We found property-based testing particularly well suited for ABS. Although it is now available in a wide range of programming languages and paradigms, propert-based testing has its origins in Haskell claessen$_q$*uickcheck* :$_2$ 000, *claessen*$_t$*esting*$_2$002*andwearguethatforthatreasonitreallyshinesinpurefunctionalprogramming.Property* − *basedtestingallowstoformulatefunctional specificationsincodewhichthentheproperty* − *testinglibrary*(*e.g.QuickCheckclae*
datacoveringasmuchcasesaspossible.Whenaninputisfoundforwhichthepropertyfails, *thelibrarythenreducesittothemostsi*
basedtestingsuitsverywelltheconstructivenatureofABS.

TODO: use the reverse or the reverse is the list again

For a technical, in-depth discussion on property-based testing in Haskell we refer to the following papers: claessen$_q$*uickcheck* :$_2$ 000, *claessen*$_t$*esting*$_2$002.

4 CASE-STUDY: PURE FUNCTIONAL SUGARSCAPE

TODO

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

2

(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.

4.1 A Functional View

The restrictions functional programming imposes, directly removes serious sources of bugs which leads to simulation which is more likely to be correct. These restrictions force us to solve the fundamental concepts in ABS implementation differently. Note that we could fall back to using IO throughout all the simulation in which case we have access to mutable references but then we lose important compile-time guarantees and

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

introduce those serious sources of bugs we want to get rid of - also testing becomes more complicated and not as strong any more because we cannot guarantee at compile time that no random IO stuff is happening within the agents. Also note that obviously no one would do random IO stuff in an agent (e.g. read from a file, open connection to server...) but one must not underestimate the value of guaranteeing its absence at compile-time. Thus, 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? How can be make it pro-active?
   In OO the obvious approach is to map an agent directly onto an object which encapsulates data and provides methods which implement the agents actions. Obviously we don't have objects in FP thus we need to find a different approach to represent the agents actions and to encapsulate its state. In the established OO approach one represents the state of an Agent explicitly in mutable member variables of the object which implements the Agent. As already mentioned we don't have objects in FP and state is immutable which leaves us with the very tricky question how to represent state of an Agent which can be actually updated. In the established OO approach, agents have a well-defined interface through their public methods through which one can interact with the agent and query information about it. Again we don't have this in FP as we don't have objects and globally mutable state. In the established OO approach one would either expose the current time-delta in a mutable variable and implement time-dependent functions or ignore it at all and assume agents act on every step. At first this seems to be not a big deal in FP but when considering that it is yet unclear how to represent Agents and their state, which is directly related to time-dependent and reactive behaviour it raises the question how we can implement time-varying and reactive behaviour in a purely functional way.
2. How can we implement agent-agent interactions?
   In the established OO approach Agents can directly invoke other Agents methods which makes direct Agent interaction straight forward. Again this is obviously not possible in FP as we don't have objects with methods and mutable state inside. In the established OO approach agents simply have access to the environment either through global mechanisms (e.g. Singleton or simply global variable) or passed as parameter to a method and call methods which change the environment. Again we don't have this in FP as we don't have objects and globally mutable state.
3. How can we represent an environment and its various types? How can we implement agent-environment interactions
   In the established OO approach an environment is almost always a mutable object which can be easily made dynamic by implementing a method which changes its state and then calling it every step as well. In FP we struggle with this for the same reasons we face when deciding how to represent an Agent, its state and proactivity.
4. How can we step the simulation?
   In the established OO approach agents are run one after another (with being optionally shuffled before to uniformly distribute the ordering) which ensures mutual exclusive access in the agent-agent and agent-environment interactions. Obviously in FP we cannot iteratively mutate a global state.

4.2 Agent representation, local state, interface and pro-activity

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. We present an example for continuations and recursions. TODO: explain

```haskell
newtype Cont a = Cont (a -> (a, Cont a))

adder :: Int -> Cont Int
adder x = Cont (\x' -> (x + x', adder (x + x')))

runCont :: Int -> Cont Int -> IO ()
runCont 0 _ = return ()
runCont n (Cont cont) = do
  let (x, cont') = cont 1
  print x
  runCont (n-1) cont'

test :: IO ()
test = runCont 100 (adder 0)
```

From the continuation 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. The source of pro-activity in ABS comes always from observing the time - when an agent can observe the flow of time, it can become pro-active and initiate actions on its own without external stimuli like events (**?**). Thus we need to make the flow of time available to our agents as well.

FRP (see Section 3.4) provides us with an interesting abstraction for a flow of time, the signal function, which are built on *recursion* and *continuations*. A signal function can be understood as a *process over time* which maps an input- to an output-signal. A signal can be understood as a value which varies over time. Thus, signal functions have an awareness of the passing of time by having access to Δt which are positive time-steps with which the system is sampled. Also some FRP implementations allow to execute signal functions within a context which can have side-effects (**?**) of which we can make use as well (see below).

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

Using signal functions and FRP allows us to solve the presented problems, thus we make an Agent a signal function. Our agent-interface is defined in terms of the input $Signal\,\alpha$ and output $Signal\,\beta$ and the type of side-effects the context allows. Further it allows us to encapsulate local state on a very strong level: there is now way to access or mutate locale state outside of the control of an agent, it all has to go through the inputs and outputs and running the signal function. Pro-activity is not a problem as well as signal functions have an awareness of time which can be used to e.g. emit events *after* a given time-out.

We present a short code-example of an infected agent of the agent-based SIR model (**?**) which recovers after a given time. The first line with the double semi-colons (::) defines the type of a function. We see that *infectedAgent* is a signal function (SF) which has as input the empty tuple (can be seen as void / no input) and outputs the SIR state it is currently in. Also this signal-function is pure and does not run within a side-effect context. By looking at the types we see no explicit Δt as input (it is hidden in the signal function and FRP implementation) thus there is no way to access it explicitly, meaning we removed a potential source of bugs.

The infected agent behaves as infected until the recovery-event happens - from that moment on it will behave as a recovered agent - which is implemented using the *switch* function provided by FRP. The *infected* function returns a tuple with the Event in addition to the SIR state which indicates if the recovery-event has happened or not. If it has, then the *switch* function will detect this and switch into *recoveredAgent*. While infected, the agent 'waits' for the recovery-event which is generated using the *occasionally* function, provided by FRP. It generates on average an event after *illnessDuration*, meaning it generates stochastic events from an exponential distribution. Depending whether the event has occurred, the infected agents outputs Infected or Recovered.

Note that *occasionally* is a stochastic function which means it makes use of a random-number stream, which is passed to *infectedAgent* in its first argument *RandomGen g => g*.

```haskell
-- an agent in the SIR model is either Susceptible, Infected or Recovered
data SIRState = Susceptible | Infected | Recovered

infectedAgent :: RandomGen g => g -> SF () SIRState
infectedAgent g
    -- behave as infected until recovery-event, then behave as
    -- recoveredAgent from that moment on
    = switch infected (const recoveredAgent)
  where
    infected :: SF () (SIRState, Event ())
    infected = proc _ -> do
      -- awaiting the recovery-event
      recEvt <- occasionally g illnessDuration () -< ()
      -- if event occurred a is Recovered, otherwise Infected
      let a = event Infected (const Recovered) recEvt
      -- return the state and event
      returnA -< (a, recEvt)

-- a recovered agent stays Recovered forever
recoveredAgent :: SF () SIRState
recoveredAgent = arr (const Recovered)
```

Note that this approach to ABS is inherently time-driven. This means that depending on the time-semantics of the model, we need to select the right time-deltas by which to sample the simulation. If it is too large we might not arrive at the correct solution, if it is too small, we run into performance problems. An alternative is to follow an event-driven approach (**?**), where agents schedule events in a Discrete Event Simulation fashion. To implement such an approach is possible using signal functions and FRP as well by running within a State context in which one manages an event queue. The simulation stepping (see below) then advances the simulation through processing the events instead of time-deltas. Although the event-driven approach can emulate the time-driven approach and is thus more general, it might be more natural to implement the simulation in a time-driven way due to the model semantics fit more natural to it.

4.3 Agent-Agent interactions

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.

A simple solution is to implement *asynchronous* interactions, which can be seen as a message sent to the target agent which will arrive in the next time step, passed in through the signal function input. If the receiving agent doesn't handle the event, it will be lost if we are in a pure context because there is no concept of a persistent message-box which would require mutable data. In a effectful context e.g. STM or

State, we could implement stateful message-boxes. Whether an asynchronous approach is suitable, depends entirely on model semantics - it might work for some models or parts of a model, but not for others.

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.

TODO: implement synchronous agent interaction

TODO: discuss macals 4 classifications of his paper (**?**)

4.4 Environment representation and Agent-Environment interactions

Depending on which kind of environment we are using we have different approaches on how to solve these problems. We distinguish between four different environment types:

1. Passive read-only e.g. a static neighbour network - The environment is passed as additional input to each agent.
2. Passive read/write e.g. a shared 2D grid like in the schelling model (**?**) - The environment is shared through an effectful State context which can be accessed and manipulated by the agents.
3. Active read-only e.g. ? - The environment is made a signal function too which broadcasts asynchronous messages about changes in the environment to all agents.
4. Active read/write e.g. Sugarscapes environment in which agents move around and harvest resources but where the environment regrows them - The environment is made a signal function which acts on a shared state which is made available to the environment *and* the agents using an effectful State context.

4.5 Stepping the simulation

An FRP library generally provides functions to either run signal functions with or without any effectful context. Further they might also provide a looping function which runs within the IO context to e.g. continuously render outputs to a window. All of it is built on the concept of recursion and continuations as we have introduced earlier which allows to feed the output of the current step into the next one, generating a closed feedback-loop.

4.6 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).

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

4.7 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 [3].

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

4.8 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.

[3]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.

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

4.9 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.

4.9.1 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

4.10 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

4.11 Testing

To see how well pure functional programming is suited for code-testing we implemented tests on 4 different levels. Note that we only implemented a few tests on each level to develop an insight in their usefulness and how well FP is suited for each level. We didn't cover the whole functionality because of lack of time. In a proper, high-quality implementation the whole functionality needs to be covered.

4.11.1 Testing utility functions

We implemented a number of tests for utility functions which we implemented as simple unit-tests and a few also as property-based tests (see next section). These utility functions are pure computations like calculating the MRS, exchange rates of a trade, genetic crossover, neighbourhood distance, wrapping coordinates. Due to their nature they are very easy to test because the have no side-effects and don't need any construction of complex simulation state. Often it is not really necessary to test these functions because they are sufficiently short and one can reason about its correctness directly in code - a key feature of functional programming due to its different notion of computation: we can reason equationally about pure computations as they were simple math equations.

TODO: discrete and bestcell

4.11.2 Testing individual agent functions

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.

TODO: implement - fertility - potential lender

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.

4.11.3Testing agent behaviour

These tests have the purpose of testing whole parts of agent behaviour. Due to the very different approach to ABS in FP this is also easier to test because agents have less dependencies between each other as they interact with each other through messages. This decouples them to a very high level. Also the sending of messages happens through passing the messages as output which will then be handled by the simulation kernel. Examples for such tests are all the handlers for incoming mating, trading or lending messages. More difficult to test is the behaviour of the Tick event which is scheduled to each agent in each time-step: depending on the scenario this can have multiple different paths and is quite involved.

TODO: implement lending ´ TODO: can we find the lending with inheritance bug with that?

Note that we are lacking testing of interaction between agents which we had to leave due to lack of time. This should follow a similar approach to testing agent behaviour but we leave this for further research.

4.11.4 Testing of the whole simulation

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.

5 DISCUSSION

TODO

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.

6.1 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.