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 explore ways of approaching ABS using the functional programming paradigm with the language Haskell. We present fundamental concepts and advanced features of functional programming and we show how to leverage the benefits of it (Hudak and Jones 1994, **?**) to become available when implementing ABS functionally.

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

To substantiate our claims we implemented the *full* SugarScape (Epstein and Axtell 1996) model in Haskell as a case-study and discuss the implications, drawbacks and benefits of a pure functional implementation of it.

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. To the best of our knowledge, it is the first one to do so.

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

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

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

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

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.

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].

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,

[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.

then it sends "MatingRequest" to the other SF. This requires a completely different approach to iterating the agents.

6 REPLICATION ISSUES

6.1 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.

6.2 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?)

6.3 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.

6.4 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.

6.5 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.

6.6 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?

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., and M. Jones. 1994, October. "Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity". Research Report YALEU/DCS/RR-1049, Department of Computer Science, Yale University, New Haven, CT.

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.