Functional programming in Agent-Based Simulation and Modelling

Jonathan Thaler

December 13, 2016

Abstract

Agent-Based Modelling and Simulation (ABM/S) is still a young discipline and the dominant approach to it is object-oriented computation. This thesis goes into the opposite direction and asks how ABM/S can be mapped to and implemented using pure functional computation and what one gains from doing so. To the best knowledge of the author, so far no proper treatment of ABM/S in this field exists but a few papers which only scratch the surface. The author argues that approaching ABM/S from a pure functional direction offers a wealth of new powerful tools and methods. The most obvious one is that when using pure functional computation reasoning about the correctness and about total and partial correctness of the simulation becomes possible. Also pure functional approaches allow the design of an embedded domain specific language (EDSL) in which then the models can be formulated by domain-experts. The strongest point in using EDSL is that ideally the distinction between specification and implementation disappears: the model specification is then already the code of the simulation-program. This allows to rule out a serious class of errors where specification and implementation does not match, which is especially a big problem in scientific computing.
In this paper we look at the very simple social-simulation of *Heroes & Cowards* invented by TODO: cite to study the new methods mentioned above and to highlight its potential use and its limitations as opposed to object-oriented methods.

Introduction still untouched: how to implement bidirectional multistep conversations? this will be of interest in ACE
Important: The strength of simulations is to put hypotheses to tests: The hypothesis-simulation-refinement cycle:

1 Agent-definition

An Agent is a metaphor for a pro-active unit, able to spawn new Agents, and interacting with other Agents in a network of neighbours by exchange of messages. The implementation of Agents can vary and strongly depends on the programming language and the kind of domain the simulation and model is

situated in. This paper looks at how Agents can be implemented in various languages using the simple Heroes & Cowards model (see below). The following implementations are discussed:

1. Java, Object-Oriented
2. Haskell, Pure Functional
3. Akka, Mixed Functional with Actors

2 Heroes and Cowards

One starts with a crowd of Agents where each Agents is positioned *randomly* in a continuous 2D-space. Each of the Agents then selects *randomly* one friend and one enemy (except itself in both cases) and decided with a given probability whether the Agent acts in the role of a "Hero" or a "Coward" - friend, enemy and role don't change after the initial set-up. Now the simulation can start: in each step the Agent will move a given distance towards a given point. If the Agent is in the role of a "Hero" this point will be the half-way distance between the Agents friend and enemy - the Agent tries to protect the friend from the enemy. If the Agent is acting like a "Coward" it will try to hide behind the friend also the half-way distance between the Agents friend and enemy, just in the opposite direction.
Note that this simulation is determined by the random starting positions, random friend & enemy selection, random role selection and number of agents. Note also that during the simulation-stepping no randomness is mentioned in the model and given the initial random set-up, the simulation-*model* is completely deterministic - whether this is the case for the implementations is another question, not relevant to the model.
TODO: add random-noise with a configurable strength to direction: is closer to reality and also agents will never stop completely - may result in completely different pattern

3 Related Research

TODO: read papers for haskell abm, see also folders. postpone ACE for later

4 Run-Time Complexity

what if the number of agents grows? how does the run-time complexity of the simulation increases? Does it differ from implementation to implementation? The model is O(n) but is this true for the implementation?

5 Reasoning

Allowing to reason about a program is one of the most interesting and powerful features of a Haskell-program. Just by looking at the types one can show that there is no randomness in the simulation *after* the random initialization, which is not slightest possible in the case of a Java, Scala, ReLogo or NetLogo solution. Things we can reason about just by looking at types:

- Concurrency involved?
- Randomness involved?
- IO with the system (e.g. user-input, read/write to file(s),...) involved?
- Termination?

This all boils down to the question of whether there are *side-effects* included in the simulation or not.
What about reasoning about the termination? Is this possible in Haskell? Is it possible by types alone? My hypothesis is that the types are an important hint but are not able to give a clear hint about termination and thus we we need a closer look at the implementation. In dependently-typed programming languages like Agda this should be then possible and the program is then also a proof that the program itself terminates.
reasoning about Heros & Cowards: what can we deduce from the types? what can we deduce from the implementation?

Compare the pure-version (both Yampa and classic) with the IO-version of haskell: we loose basically all power to reason by just looking at the types as all kind of side-effects are possible when running in the IO-Monad.
in haskell pure version i can guarantee by reasoning and looking at the types that the update strategy will be simultaneous deterministic. i cant do that in java

5.1 Debugging & Testing

Because functions compose easier than classes & objects (TODO: we need hard claims here, look for literature supporting this thesis or proof it by myself) it is also much easier to debug *parts* of the implementation e.g. the rendering of the agents without any changes to the system as a whole - just the main-loop has do be adopted. Then it is very easy to calculate e.g. only one iteration and to freeze the result or to manually create agents instead of randomly create initial ones.
TODO: quickcheck

6 World

The coordinates calculated by the agents are *virtual* ones ranging between 0.0 and 1.0. This prevents us from knowing the rendering-resolution and polluting code which has nothing to do with rendering with these implementation-details. Also this simulation could run without rendering-output or any rendering-frontend thus sticking to virtual coordinates is also very useful regarding this (but then again: what is the use of this simulation without any visual output=

- Infinite: movement is unrestricted.
- Clipping: coordinates are clipped at 0.0 and 1.0
- Wraparound: coordinates are wrapped around to 0.0 when reaching 1.0. Will lead pursuing friends to change direction apruptly when wrapping around.

7 Lazy Evaluation

can specify to run the simulation for an unlimited number of steps but only the ones which are required so far are calculated.

8 Performance

Java outperforms Haskell implementation easily with 100.000 Agents - at first not surprising because of in-place updates of friend and enemies and no massive copy-overhead as in haskell. But look WHERE exactly we loose / where the hotspots are in both solutions. 1000.000 seems to be too much even for the Java-implementation.

9 Numerical Stability

The agents in the Java-implementation collapsed after a given number of iterations into a single point as during normalization of the direction-vector the length was calculated to be 0. This could be possible if agents come close enough to each other e.g. in the border-worldtype it was highly probable after some iterations when enough agents have assembled at the borders whereas in the Wrapping-WorldType it didn't occur in any run done so far.
In the case of a 0-length vector a division by 0 resulting in NaN which *spread* through the network of neighbourhood as every agent calculated its new position it got *infected* by the NaN of a neighbour at some point. The solution was to simply return a 0-vector instead of the normalized which resulted in no movement at all for the current iteration step of the agent.

10 Visualization

Render all Render cowards only Render heroes only

11 Update-Strategies

1. All states are copied/frozen which has the effect that all agents update their positions *simultaneously*
2. Updating one agent after another utilizing aliasing (sharing of references) to allow agents updated *after* agents before to see the agents updated before them. Here we have also two strategies: deterministic- and random-traversal.
3. Local observations: Akka

11.1 Different results with different Update-Strategies?

Problem: the following properties have to be the same to reproduce the same results in different implementations:

Same initial data: Random-Number-Generators Same numerical-computation: floating-point arithmetic Same ordering of events: update-strategy, traversal, parallelism, concurrency

- Same Random-Number Generator (RNG) algorithm which must produce the same sequence given the same initial seed.
- Same Floating-Point arithmetic
- Same ordering of events: in Scala & Actors this is impossible to achieve because actors run in parallel thus relying on os-specific non-deterministic scheduling. Note that although the scheduling algorithm is of course deterministic in all os (i guess) the time when a thread is scheduled depends on the current state of the system which can change all the time due to *very* high number of variables outside of influence (some of the non-deterministic): user-input, network-input, ... which in effect make the system appear as non-deterministic due to highly complex dependencies and feedback.
- Same dt sequence =¿ dt MUST NOT come from GUI/rendering-loop because gui/rendering is, as all parallelism/concurency subject to performance variations depending on scheduling and load of OS.

It is possible to compare the influences of update-strategies in the Java implementation by running two exact simulations (agentcount, speed, dt, herodistribution, random-seed, world-type) in lock-step and comparing the positions of the agent-pairs with same ids after each iteration. If either the x or y coordinate

is no equal then the positions are defined to be *not* equal and thus we assume the simulations have then diverged from each other.
It is clear that we cannot compare two floating-point numbers by trivial == operator as floating-point numbers always suffer rounding errors thus introducing imprecision. What may seem to be a straight-forward solution would be to introduce some epsilon, measuring the absolute error: abs(x1 - x2) ¿ epsilon, but this still has its pitfalls. The problem with this is that, when number being compared are very small as well then epsilon could be far too big thus returning to be true despite the small numbers are compared to each other quite different. Also if the numbers are very large the epsilon could end up being smaller than the smallest rounding error, so that this comparison will always return false. The solution would be to look at the *relative error*: abs((a-b)/b) ¡ epsilon.
The problem of introducing a relative error is that in our case although the relative error can be very small the comparison could be determined to be different but looking in fact exactly the same without being able to be distinguished with the eye. Thus we make use of the fact that our coordinates are virtual ones, always being in the range of [0..1] and are falling back to the measure of absolute error with an epsilon of 0.1. Why this big epsilon? Because this will then definitely show us that the simulation is *different*.

The question is then which update-strategies lead to diverging results. The hypothesis is that when doing simultaneous updates it should make no difference when doing random-traversal or deterministic traversal =¿ when comparing two simulations with simultaneous updates and all the same except first random- and the other deterministic traversal then they should never diverge. Why? Because in the simultaneous updates there is no ordering introduce, all states are frozen and thus the ordering of the updates should have no influence, *both simulations should never diverge,* ***independent how dt and epsilon are selected***.
Do the simulation-results support the hypothesis? Yes they support the hypothesis - even in the worst cast with very large dt compared to epsilon (e.g. dt = 1.0, epsilon = 1.0-12)
The 2nd hypothesis is then of course that when doing consecutive updates the simulations will *always* diverge independent when having different traversal-strategies.
Simulations show that the selection of *dt* is crucial in how fast the simulations diverge when using different traversal-strategies. The observation is that *The larger dt the faster they diverge and the more substantial and earlier the divergence.*. Of course it is not possible to proof using simulations alone that they will always diverge when having different traversal-strategies. Maybe looking at the dynamics of the error (the maximum of the difference of the x and y pairs) would reveal some insight?

The 3rd hypothesis is that the number of agents should also lead to increased speed of divergence when having different traversal-strategies. This could be shown when going from 60 agents with a dt of 0.01 which never exceeded a

global error of 0.02 to 6000 agents which after 3239 steps exceeded the absolute error of 0.1.

12 Reproducing Results in different Implementations

actors: time is always local and thus information as well. if we fall back to a global time like system time we would also fall back to real-time. anyway in distributed systems clock sync is a very non-trivial problem and inherently not possible (really?). thus using some global clock on a metalevel above/outside the simulation will only buy us more problems than it would solve us.
qualitative comparison: print pucture with patterns. all implementations are able to reproduce these patterns independent from the update strategy
no need to compare individual runs and waste time in implementing RNGs, what is more interesting is whether the qualitative results are the same: does the system show the same emergent behaviour? Of course if we can show that the system will behave exactly the same then it will also exhibit the same emergent behaviour but that is not possible under some circumstances e.g. the simulation-runs of Akka are always unique and never comparable due to random event-ordering produced by concurrency & scheduling. Also we don't have to proof the obvious: given the same algorithm, the same random-data, the same treatment of numbers and the same ordering of events, the outcome *must* be the same, otherwise there are bugs in the program. Thus when comparing results given all the above mentioned properties are the same one in effect tests only if the programs contain no bugs - or the same bugs, if they *are the same*.

Thus we can say: the systems behave qualitatively the same under different event-orderings.
Thus the essence of this boils down to the question: "Is the emergent behaviour of the system is stable under random/different/varying event-ordering?". In this case it seems to be so as proofed by the Akka implementation. In fact this is a very desirable property of a system showing emergent behaviour but we need to get much more precise here: what is an event? what is an emergent behaviour of a system? what is random-ordering of events? (Note: obviously we are speaking about repeated runs of a system where the initial conditions may be the same but due to implementation details like concurrency we get a different event-ordering in each simulation-run, thus the event-orderings vary between runs, they can be in fact be regarded as random).

12.1 Problem of RNG

Have to behave EXACTLY The same: VERY difficult because of differing interfaces e.g. compare java to haskell RNGs. Solution: create a deterministic RNG generating a number-stream starting from 1 and just counting up. The program should work also in this case, if not, something should be flawed!

Peer told me to implement a RNG-Trace: generate a list of 1000.0000 pre-calculated random-numbers in range of [0..1], store them in a file and read the trace in all implementations. Needs lots of implementation.

13 Simulation-Loops

There are at least 2 parts to implementing a simulation: 1. implementing the logic of an agent and 2. implementing the iteration/recursion which drives the whole simulation
Classic
Yampa
TODO: use par to parallelize Gloss
gloss provides means for simple simulation using simulate method. But: are all ABM systems like that?

14 Agent-Representation

Java: (immutable) Object Haskell Classic: a struct Haskell Yampa: a Signal-Function Gloss: same as haskell classic Akka: Actors

15 EDSL

simplify simulation into concise EDSL: distinguish between different kind if sims: continuous/discrete iteration on: fixed set, growing set, shrinking set, dynamic set.

References

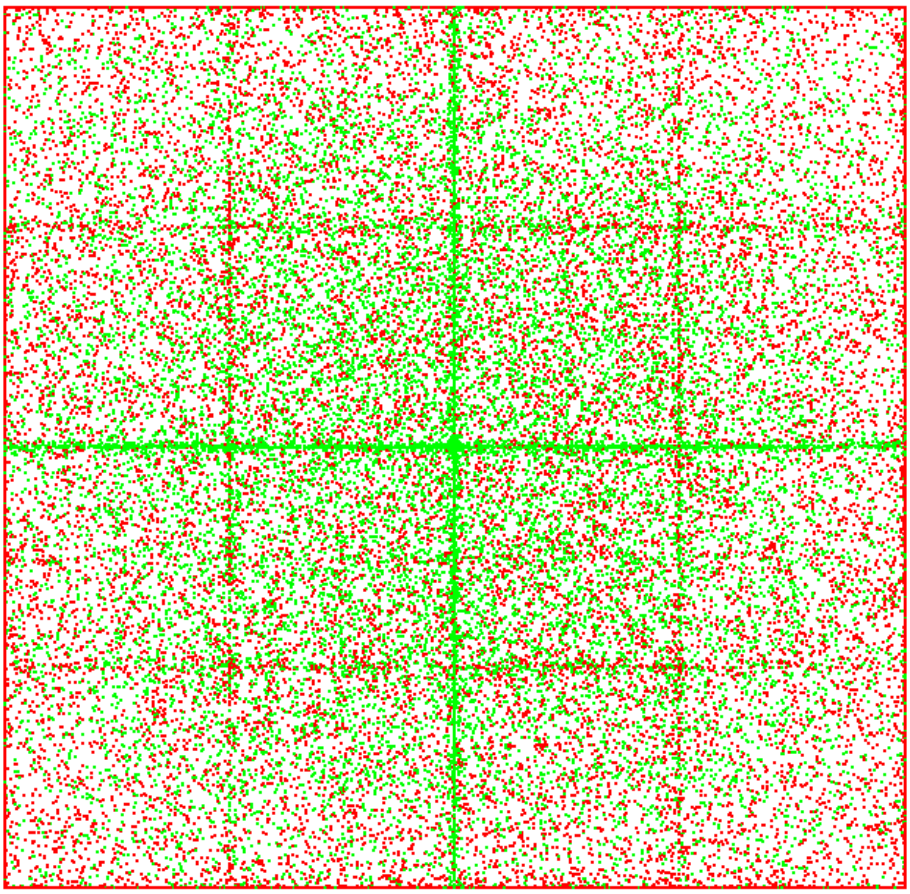

Figure 1: The emergent pattern used as criteria for qualitative comparison of implementations. Note the big green cross in the center and the smaller red crosses in each sub-sector. World-type is *border* with 100.000 Agents where 25% are Heroes.