Functional programming in Agent-Based Simulation and Modelling

Jonathan Thaler

December 7, 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

1 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?

This all boils down to the question of whether there are *side-effects* included in the simulation or not.

1.1 Reasoning about termination

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.

1.2 Debugging

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.

2 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=

- Clipping: *calculated* coordinates are clipped at 0.0 and 1.0
- Wraparound: *calculated* coordinates are wrapped around to 0.0 when reaching 1.0. Will lead pursuing friends to change direction apruptly when wrapping around.
- Remainder: *calculated* coordinates are never clipped but in rendering only the remainder after the comma is taken. Of course this then always requires a visual output to observe the effects. This will result in an effect like the classic asteroid game where agents their friends but wont change direction when they seem to clamp around like in the wraparound version.

3 Lazy Evaluation

can run a simulation for a given number of steps but

4 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.

5 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.

6 Visualization

Render all Render cowards only Render heroes only

7 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 in Scala & Actors

7.1 Different results with different Update-Strategies?

Problem: need to be *exactly* the same calculations =¿ same Random-Number-Generators, same floating-point arithmetic, same ordering of events: IMPOSSIBLE using Scala & Actors

- 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.

References