Functional programming in Agent-Based Simulation and Modelling

Jonathan Thaler

December 14, 2016

Abstract

In this paper we look at the very simple social-simulation of *Heroes & Cowards* invented by [26] to study new methods in Agent-Based Modelling and Simulation (ABM/S) and to highlight their potentials and limitations as opposed to object-oriented ones which are dominant in this field. We go into the opposite direction and ask how ABM/S can be done using functional programming, a method which has so far not been considered very deeply in this field. Obviously this method requires to look at ABM/S from a different perspective which this paper tries to provide. Although it might be strange in the beginning the author argues that approaching ABM/S from a functional direction offers a wealth of new powerful tools and methods, depending on the functional language and framework one is using. In this paper we look into two approaches where both offer different benefits: *pure* functional using Haskell and *multi-paradigm* functional using Scala. The most obvious benefits of using a pure functional approach are that it allows to reason about various properties of the program and implementation of an embedded domain-specific language (EDSL) where ideally the distinction between specification and implementation disappears. Thus with these two benefits this approach is especially suited to scientific computing where correctness and reasoning about properties of a program is of very importance. Scala, as already outlined, is a multi-paradigm approach to functional programming and allows to incorporate objects and side-effects. Thus one must take care and be disciplined to stick to a functional style in Scala. Also this language is known for its use in the framework *Akka* which is a distributed programming environment built upon the *Actor-Model* which has been gaining popularity in recent years (TODO: cite some claim?). Thus for Scala we chose an actor-model approach and functional paradigm to implement the simulation. For comparison with state-of-the-art methods we also implement the model in AnyLogic and ReLogo and look into an object-oriented implementation in Java without using any ABM/S framework.

1 Introduction

Important: The strength of simulations is to put hypotheses to tests: The hypothesis-simulation-refinement cycle:
All implementations of ABM/S models must solve two problems: 1. the model/data/ part: how agents can be represented and 2. the dynamic part: how the simulation is stepped

2 Agent-Based Modelling and Simulation (ABM/S)

ABM/S is a method of modelling and simulating a system where the global behaviour may be unknown but the behaviour and interactions of the parts making up the system is of knowledge (Wooldrige, M. (2009). An Introduction to MultiAgent Systems. John Wiley & Sons). Those parts, called agents, are modelled and simulated out of which then the aggregate global behaviour of the whole system emerges. Thus the central aspect of ABM/S is the concept of an Agent which can be understood as 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.

3 Functional programming

TODO: read [4]
The state-of-the-art approach to implementing Agents are object-oriented methods and programming as the metaphor of an Agent as presented above lends itself very naturally to object-orientation (OO). The author of this thesis claims that OO in the hands of inexperienced or ignorant programmers is dangerous, leading to bugs and hardly maintainable and extensible code. The reason for this is that OO provides very powerful techniques of organising and structuring programs through Classes, Type Hierarchies and Objects, which, when misused, lead to the above mentioned problems. Also major problems, which experts face as well as beginners are 1. state is highly scattered across the program which disguises the flow of data in complex simulations and 2. objects don't compose as well as functions. The reason for this is that objects always carry around some internal state which makes it obviously much more complicated as complex dependencies can be introduced according to the internal state. All this is tackled by (pure) functional programming which abandons the concept of global state, Objects and Classes and makes data-flow explicit. This then allows to reason about correctness, termination and other properties of the program e.g. if a given function exhibits side-effects or not. Other benefits are fewer lines of code, easier maintainability and ultimately fewer bugs thus making functional programming the ideal choice for scientific computing and simulation and thus

also for ACE. A very powerful feature of functional programming is Lazy evaluation. It allows to describe infinite data-structures and functions producing an infinite stream of output but which are only computed as currently needed. Thus the decision of how many is decoupled from how to (Hughes, J. (1989). Why functional programming matters. Comput. J., 32(2):98–107.). The most powerful aspect using pure functional programming however is that it allows the design of embedded domain specific languages (EDSL). In this case one develops and programs primitives e.g. types and functions in a host language (embed) in a way that they can be combined. The combination of these primitives then looks like a language specific to a given domain, in the case of this thesis ACE. The ease of development of EDSLs in pure functional programming is also a proof of the superior extensibility and composability of pure functional languages over OO (Henderson P. (1982). Functional Geometry. Proceedings of the 1982 ACM Symposium on LISP and Functional Programming.). One of the most compelling example to utilize pure functional programming is the reporting of Hudak (Hudak P., Jones M. (1994). Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity. Department of Computer Science, Yale University.) where in a prototyping contest of DARPA the Haskell prototype was by far the shortest with 85 lines of code. Also the Jury mistook the code as specification because the prototype did actually implement a small EDSL which is a perfect proof how close EDSL can get to and look like a specification.
Functional languages can best be characterized by their way computation works: instead of *how* something is computed, *what* is computed is described. Thus functional programming follows a declarative instead of an imperative style of programming. The key points are:

- No assignment statements - variables values can never change once given a value.
- Function calls have no side-effect and will only compute the results - this makes order of execution irrelevant, as due to the lack of side-effects the logical point in *time* when the function is calculated within the program-execution does not matter.
- higher-order functions
- lazy evaluation
- Looping is achieved using recursion, mostly through the use of the general fold or the more specific map.
- Pattern-matching

This alone does not really explain the *real* advantages of functional programming and one must look for better motivations using functional programming languages. One motivation is given in [16] which is a great paper explaining to non-functional programmers what the significance of functional programming is

and helping functional programmers putting functional languages to maximum use by showing the real power and advantages of functional languages. The main conclusion is that *modularity*, which is the key to successful programming, can be achieved best using higher-order functions and lazy evaluation provided in functional languages like Haskell. [16] argues that the ability to divide problems into sub-problems depends on the ability to glue the sub-problems together which depends strongly on the programming-language and [16] argues that in this ability functional languages are superior to structured programming.
TODO: comparison of functional and object-oriented programming. My points are:

- The way state can be changed and treated - distributed over multiple objects - is often very difficult to understand.
- Inheritance is a dangerous thing if not used with care because inheritance introduces very strong dependencies which cannot be changed during runtime anymore.
- Objects don't compose very well: `http://zeroturnaround.com/rebellabs/why-the-debate-on-object-oriented-vs-functional-programming-is-all-about-composition/`
- (Nearly) impossible to reason about programs

In conclusion the upsides of functional programming as opposed to OO are:

- Much more explicit flow of data & control
- Much better compose-able
- Much better parallelism

3.1 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

3.2 The Model: 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
The most obvious shortcomings of this model are its simplicity but that was chosen intentionally to prevent the research of the methods to be cluttered with too many subtle details of the model - also young children can easily understand how this model works and thus one does not need to constantly explain subtle implementation details because everything is pretty obvious.

3.2.1 Extension 1: 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.

3.2.2 Extension 2: Random-Noise

4 Visualization

Render all Render cowards only Render heroes only

5 Implementations

5.1 ABM/S Frameworks

NetLogo AnyLogic ReLogo

5.2 Object-Oriented in Java

Mutable / Immutable Side-Effects all over the place?
TODO: show how one can program more in a functional style in Java.

5.3 Pure Functional in Haskell

No framework Yampa Gloss Aivika

5.4 Multi-Paradigm Functional in Scala

6 Related Research

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

6.1 Scientific Computation

TODO: discuss [19] TODO: discuss [17] TODO: discuss [6]

6.2 Haskell

[5] constructs two frameworks: an agent-modelling framework and a DES framework, both written in Haskell. They put special emphasis on parallel and concurrency in their work. The author develops two programs: HLogo which is a clone of the NetLogo agent-modelling framework and HDES, a framework for discrete event simulation - where in both implementations is the very strong emphasis on parallelism. Here only the HLogo implementation is of interest as it is directly related to agent-based simulation. In this implementation the author claims to have implemented an EDSL which tries to be close to the language used for modelling in NetLogo (Logo) "which lifts certain restrictions of the original NetLogo implementation". Also the aim was to be "faster in most circumstances than NetLogo" and "utilizes many processor cores to speedup the execution of Agent Based Models". The author implements a primitive model of concurrent agents which implements a non-blocking concurrent execution of

agents which report their results back to the calling agent in a non-blocking manner. The author mentions that a big issue of the implementation is that repeated runs with same inputs could lead to different results due to random event-orderings happening because of synchronization. The problem is that the author does not give a remedy for that and just accepts it as a fact. Of course it would be very difficult, if not impossible, to introduce determinism in an inherently concurrent execution model of agents which may be the reason the author does not even try. Unfortunately the example implementation the author uses for benchmarking is a very simplistic model: the basic pattern is that agent A sends to agent B and thats it - no complex interactions. Of course this lends itself very good to parallel/concurrent execution and does not need a sophisticated communication protocol. The work lacks a proper treatment of the agent-model presented with its advantages and disadvantages and is too sketchy although the author admits that is is just a proof of concept.

Tim Sweeney, CTO of Epic Games gave an invited talk about how "future programming languages could help us write better code" by "supplying stronger typing, reduce run-time failures; and the need for pervasive concurrency support, both implicit and explicit, to effectively exploit the several forms of parallelism present in games and graphics." [23]. Although the fields of games and agent-based simulations seem to be very different in the end, they have also very important similarities: both are simulations which perform numerical computations and update objects - in games they are called "game-objects" and in abm they are called agents but they are in fact the same thing - in a loop either concurrently or sequential. His key-points were:

- Dependent types as the remedy of most of the run-time failures.
- Parallelism for numerical computation: these are pure functional algorithms, operate locally on mutable state. Haskell ST, STRef solution enables encapsulating local heaps and mutability within referentially transparent code.
- Updating game-objects (agents) concurrently using STM: update all objects concurrently in arbitrary order, with each update wrapped in atomic block - depends on collisions if performance goes up.

TODO: discuss [20] TODO: discuss [25] TODO: discuss [22] TODO: discuss [18] TODO: discuss [9] TODO: discuss [21]

6.3 Erlang

TODO: discuss [11] TODO: discuss [10] TODO: discuss [24]

6.4 Actors

The Actor-Model, a model of concurrency, has been around since the paper [15] in 1973. It was a major influence in designing the concept of Agents and

although there are important differences between Actors and Agents there are huge similarities thus the idea to use actors to build agent-based simulations comes quite natural. Although there are papers around using the actor model as basis for their ABMS unfortunately no proper theoretical treatment of using the actor-model in implementing agent-based simulations has been done so far. This paper looks into how the more theoretical foundations of the suitability of actor-model to ABMS and what the upsides and downsides of using it are.
`http://www.grids.ac.uk/Complex/ABMS/`
[5] describes in chapter 3.3 a naive clone of NetLogo in the Erlang programming language where each agent was represented as an Erlang process. The author claims the 1:1 mapping between agent and process to "be inherently wrong" because when recursively sending messages (e.g. A to B to A) it will deadlock as A is already awaiting Bs answer. Of course this is one of the problems when adopting Erlang/Scala with Akka/the Actor Model for implementing agents *but it is inherently short-sighted to discharge the actor-model approach just because recursive messaging leads to a deadlock*. It is not a problem of the actor-model but merely a very problem with the communication protocol which needs to be more sophisticated than [5] described. The hypothesis is that the communication protocol will be in fact *very highly application-specific* thus leading to non-reusable agents (across domains, they should but be re-usable within domains e.g. market-simulations) as they only understand the domain-specific protocol. This is definitely NOT a drawback but can't be solved otherwise as in the end (the content of the) communication can be understand to be the very domain of the simulation and is thus not generalizable. Of course specific patterns will show up like "multi-step handshakes" but they are again then specifically applied to the concrete domain.
TODO: discuss [15] TODO: discuss [12] TODO: discuss [8] TODO: discuss [1] TODO: discuss [3] TODO: discuss [2] TODO: discuss [13] TODO: discuss [14]

7 Results

7.1 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?
- 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

7.1.1 The type of a Simulation

the type of a simulation: try to define the most general types of a simulation and then do reasoning about it

7.2 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 [7]

7.3 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.

7.4 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.

7.5 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.

7.6 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

7.7 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 diver-*

gence.. 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.

7.8 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. real-time does not help either as it is never hard real time and thus also unpredictable: if one tells the actor to send itself a message after 100ms then one relies on the capability of the OS-timer and scheduler to schedule exactly after 100ms: something which is always possible thus 100ms are never hard 100ms but soft with variations.
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).

7.9 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.

7.10 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?

7.11 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?

7.12 Agent-Representation

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

7.13 EDSL

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

8 Conclusions

consecutive updates are expensive in functional (immutable) approaches when avoiding side-effects

9 Further Research

9.1 Multi-Step Conversations

The communication in this simulation is single-step unidirectional: in each step of the simulation an agent looks at the position of the enemy and friend and updates its position, there is no conversation going on between the agent and its friend and enemy thus making it single-step and unidirectional because the whole information flow is initiated from one agent and no response is given. This is becomes kind of relaxed in the Scala implementation but is still basically unidirectional and single-step - the agents don't engage in a conversation. In many ABM/S models this is perfectly reasonable because many of the models work this way but when having e.g. bartering processes like in agent-based computational economics (ACE) where agents have a conversation with multiple asks and bids to find a price they are happy with, this method becomes obviously too restricted.
The author investigates exactly this problem in an additional paper, where he looks at how to implement bartering-processes in ACE using Akka and Haskell (bidirectional multi-step conversations)

9.2 LISP

LISP is the oldest functional programming language and the second-oldest high-level programming language, only one year younger than Fortran. It would have been very interesting to research how we can do ABM/S in LISP utilizing its *homoiconicity* but that would have opened up too much complexity also because LISP, despite being a functional programming language, is too far away from both Haskell and Scala. Thus the topic of applying LISP to ABM/S is left for further research in another paper.

9.3 Process-Calculi

There is a strong connection of the ideas between the Actor-Model and Process-Calculi like the Pi-Calculus (TODO: cite Milner) and research has been done on connecting both worlds (TODO: cite Agha Gul). Also because the Actor-Model is so close to Agents because it was a major inspiration for the development of Agents and thus be regarded as one way of implementing Agents, one can argue due to transitivity that Agents can be connected to Pi-Calculus as well. This would allow to formalize Agents using the algebraic power and tools developed in Pi-Calculus.

References

[1] Agha, G. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.

[2] Agha, G. An algebraic theory of actors and its application to a simple object-based language. In *In Ole-Johan Dahl's Festschrift, volume 2635 of LNCS* (2004), Springer, pp. 26–57.

[3] Agha, G. A., Mason, I. A., Smith, S. F., and Talcott, C. L. A foundation for actor computation. *J. Funct. Program. 7*, 1 (Jan. 1997), 1–72.

[4] Backus, J. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. *Commun. ACM 21*, 8 (Aug. 1978), 613–641.

[5] Bezirgiannis, N. Improving performance of simulation software using haskell's concurrency & parallelism. Master's thesis, Utrecht University - Dept. of Information and Computing Sciences, 2013.

[6] Botta, N., Mandel, A., Ionescu, C., Hofmann, M., Lincke, D., Schupp, S., and Jaeger, C. A functional framework for agent-based models of exchange. *Applied Mathematics and Computation 218*, 8 (2011), 4025 – 4040.

[7] Claessen, K., and Hughes, J. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not. 35*, 9 (Sept. 2000), 268–279.

[8] Clinger, W. D. Foundations of actor semantics. Tech. rep., Cambridge, MA, USA, 1981.

[9] De Jong, T. Suitability of haskell for multi-agent systems. Tech. rep., 2014.

[10] Di Stefano, A., and Santoro, C. Using the erlang language for multi-agent systems implementation. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology* (Washington, DC, USA, 2005), IAT '05, IEEE Computer Society, pp. 679–685.

[11] Di Stefano, A., and Santoro, C. exat: an experimental tool for programming multi-agent systems in erlang. Tech. rep., 2007.

[12] Grief, I., and Greif, I. Semantics of communicating parallel processes. Tech. rep., Cambridge, MA, USA, 1975.

[13] Hewitt, C. *What Is Commitment? Physical, Organizational, and Social (Revised).* Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 293–307.

[14] Hewitt, C. Actor model for discretionary, adaptive concurrency. *CoRR abs/1008.1459* (2010).

[15] Hewitt, C., Bishop, P., and Steiger, R. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (San Francisco, CA, USA, 1973), IJCAI'73, Morgan Kaufmann Publishers Inc., pp. 235–245.

[16] Hughes, J. Why functional programming matters. *Comput. J. 32*, 2 (Apr. 1989), 98–107.

[17] Ionescu, C., and Jansson, P. *Dependently-Typed Programming in Scientific Computing.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 140–156.

[18] Jankovic, P., and Such, O. Functional programming and discrete simulation. Tech. rep., 2007.

[19] Kamiński, Bogumił, a. S. P. *Verification of Models in Agent Based Computational Economics — Lessons from Software Engineering.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 185–199.

[20] Schneider, O., Dutchyn, C., and Osgood, N. Towards frabjous: A two-level system for functional reactive agent-based epidemic simulation. In *Proceedings of the 2Nd ACM SIGHIT International Health Informatics Symposium* (New York, NY, USA, 2012), IHI '12, ACM, pp. 785–790.

[21] Sorokin, D. Aivika 3: Creating a Simulation Library based on Functional Programming, 2015.

[22] Sulzmann, M., and Lam, E. Specifying and controlling agents in haskell. Tech. rep., 2007.

[23] Sweeney, T. The next mainstream programming language: A game developer's perspective. *SIGPLAN Not. 41*, 1 (Jan. 2006), 269–269.

[24] Varela, C., Abalde, C., Castro, L., and Gulías, J. On modelling agent systems with erlang. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Erlang* (New York, NY, USA, 2004), ERLANG '04, ACM, pp. 65–70.

[25] Vendrov, I., Dutchyn, C., and Osgood, N. D. *Frabjous: A Declarative Domain-Specific Language for Agent-Based Modeling.* Springer International Publishing, Cham, 2014, pp. 385–392.

[26] Wilensky, U., and Rand, W. *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NETLogo.* MIT Press, 2015.

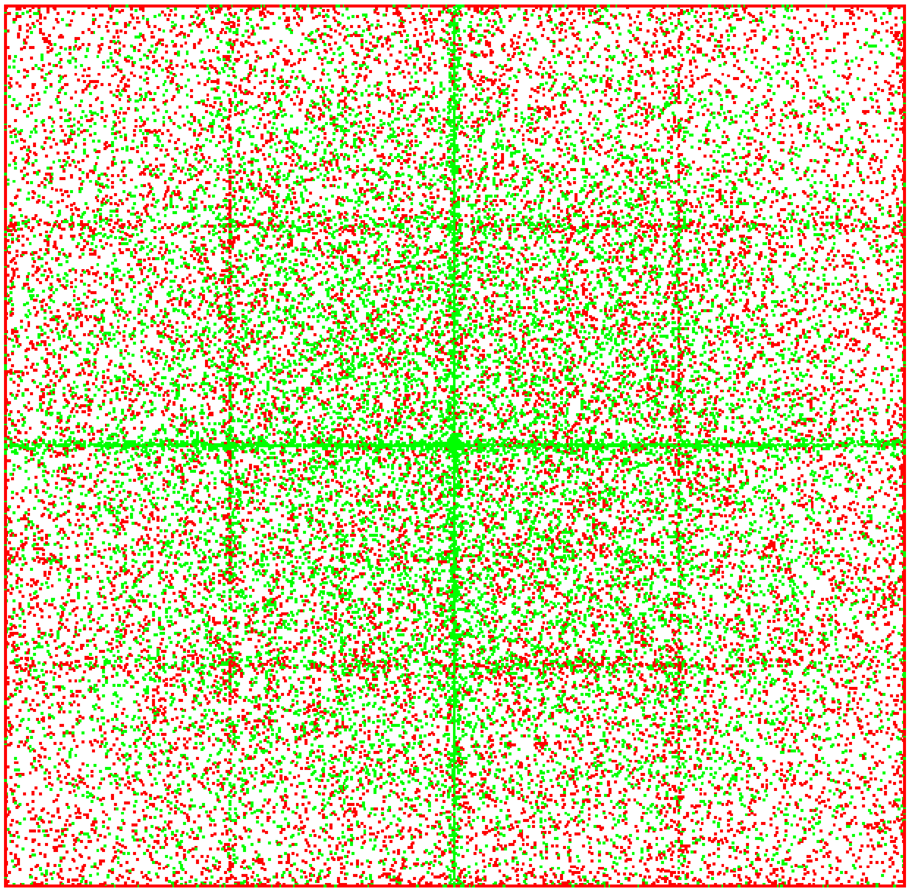

Figure 1: The emergent pattern used as criteria for qualitative comparison of implementations. Note the big green cross in the center and the smaller red crosses in each sub-sector. World-type is *border* with 100.000 Agents where 25% are Heroes.