Influence of Simulation-Semantics on Dynamics of an Agent-Based Simulation

Jonathan Thaler

December 22, 2016

Abstract

In this paper we look at the very simple social-simulation of *Heroes & Cowards* invented by [28] to study the impact of different simulation-semantics on the dynamics of the simulation. By simulation-semantics we understand the different approaches of how to step a simulation and we ask whether the dynamics are somewhat stable, change or completely break down under different semantics. We develop a classification of all potential simulation-semantics and discuss general implementation-considerations independent of the programming language. We then give implementations of each simulation-semantic in varying programming languages and compare the resulting dynamics. Depending on the given semantics we choose out of three languages: Java, Haskell and Scala with Actors where each of them has their strengths and limitations implementing simulation-semantics. We implement only the semantics for which the given language is suited for without abusing it. Thus in Java we focus on object-oriented programming, side-effects and global data, where in Haskell we focus on pure functional programming without side-effects or global data and in Scala with Actors we put emphasis on a mixed-paradigm approach and the usage of Actors as defined by [1].

It is important to note that the implementations in Haskell present a novel, pure functional approach to ABM/S which strengths are the declarative style of programming and the strong static type-system. This allows to reason about a program and implement embedded domain-specific languages (EDSL) where ideally the distinction between a formal specification and the implementation disappears. We present the EDSL implemented for this Model and investigate what and how we can reason about properties and dynamics of our simulation and of ABM/S in general having the EDSL, the types and our program at hand.

Thus the main contributions of this paper are fourfold. First it establishes a terminology for speaking about and classifying simulation-semantics, second it gives a general framework to discuss dynamics of an ABM/S under different simulation-semantics, third it discusses the suitability of three very different programming-languages to implement the various simulation-semantics and fourth it looks into the possibility and power of reasoning about properties and dynamics of an ABM/S with specific simulation-semantics implemented in the pure functional language Haskell.

1 Introduction

Today simulations are at the very heart of many sciences. They allow to put hypotheses to test by building a model which abstracts from reality, keeping only the important and relevant details, and then bringing this model to life in simulation. Based on the results shown by the dynamics, previously formulated hypotheses can be verified or falsified resulting in a formulate-simulate-refine cycle.
The meaning of simulating a model can be understood as calculating the dynamics of a (model of a) system over time thus the state of the system at time t depends on the state of the system at time t - epsilon. Here we only consider simulations in a computer-system (TODO: are there simulations NOT in a computer?), which is an inherently discrete system which poses us with the question of how to represent time which seems linear and continuously flowing to us in reality (NOTE: this may not be physically the case but for our considerations this should be a good approximation). Being in a discrete system, of course implies that time has to be discretised as well and there are two ways of doing it: discrete and continuous where in discrete case time advances in steps of the natural numbers and where in the continuous case time advances in steps of real-numbers. Note that in both cases the system is iterated in steps where only the *numerical type* of the input to the time-dependent functions differs. Thus a simulation in a computer can be understood as an iteration over a model for a given number of steps where each step advances time by dt (either discrete or continuous) and, based on the previous model-state, producing an updated model-state which again becomes the input for the next step. Thus in each simulation we have three inputs: 1. the model, 2. number of steps, 3. time dt. There are of course different models and types of simulations and in this paper we will focus on one particular one: agent-based, which will be described next.

1.1 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 [29]. 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. Whereas the majority of ABM/S are implemented in object-oriented (OO) languages e.g. Java, C++, this paper focuses on functional ones.

1.2 Why Functional programming?

Object-oriented (OO) programming is the current state-of-the-art method used in implementing ABM/S due to the natural way of mapping concepts and models of ABM/S to an OO-language. Although this dominance in the field we claim that OO has also its serious drawbacks:

- Mutable State is distributed over multiple objects which is often very difficult to understand, track and control.
- Inheritance is a dangerous thing if not used properly and with care because it introduces very strong dependencies which cannot be changed during runtime any-more.
- Objects don't compose very well due to their internal (mutable) state (note that we are aware that there is the concept of immutable objects which are becoming more and more popular but that does not solve the fundamental problem.
- It is (nearly) impossible to reason about programs.

We claim that these drawbacks are non-existent in pure functional programming like Haskell due to the nature of the functional approach. To give an introduction into functional programming is out of scope of this paper but we refer to the classical paper of [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 of this classical paper 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.

1.3 The Model: Heroes & Cowards

To study various properties of implementations of ABM/S we select the very simple model *Heroes & Cowards* from social-simulation invented by [28]. Although it is very simple, it will prevent the research of the methods to be cluttered with too many subtle details of the model thus focusing on the methods and implementation than rather on the model itself.
One starts with a crowd of Agents where each Agent 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 decides 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.
The world this model is situated in is restricted by borders in the form of a rectangle: the agents cannot move out of it and will be clipped against the border if the calculation would end them up outside.
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.

1.3.1 Extension 1: World-Types

We extend the model by introducing 2 additional world-types: Infinite and Wrapping thus ending up with 3 World-Types:

1. Border - Agents cannot move out of the restricted rectangle and are clipped at the border. This is the world-type of the original model.
2. Infinite - Agents can move unrestricted.
3. Wrapping - Same as Border but when crossing border the Agents "teleport" to the opposite side of the world thus the world folds back unto itself in 2D.

1.3.2 Extension 2: Random-Noise

The original model is completely deterministic after the initial set-up. We add the possibility for optional random noise at two points: the distance / position when hiding/protecting can be made subject to random noise in a given range as well as the width of the step the agent makes when moving towards the hiding/protecting position.

1.4 Other Models: SIRS, Wildfire and Schelling Segregation

We will also look into other classic Models of ABM/S but not with the same focus as *Heroes & Cowards* and just to show that it is very easy to implement them using our ABM/S library in Haskell.

1.4.1 SIRS

Simulates the spreading of a disease throughout a population.

1.4.2 Wildfire

Simulates a Wildfire on a discrete 2D-Grid where each cell corresponds to a part of a forest with varying amount of burnable wood. The wildfire is then ignited at some random cell and spreads over the whole 2D-Grid by igniting neighbouring cells which burn for a duration proportional to the amount of burnable woods in this cell.

1.4.3 Schelling Segregation

TODO: describe

1.5 Contributions

When using pure functional instead of the state-of-the-art OO programming we need to find other ways of representing Agents and implement an ABS. This is the main contributions of this paper in form of an implementation of a library for ABM/S implemented in pure Haskell as so far no proper library which meets the requirements for ABM/S exists in Haskell (see Related Research). Also the paper compares the library to implementations with state-of-the-art ABM/S Frameworks: NetLogo, AnyLogic and ReLogo. Also comparisons are drawn with an OO solution in Java. Of special interest is the comparison to the multi-paradigm functional solution in Scala & Actors as this is the method closest to the Haskell-Library implemented.
We claim that our library is well suited for ABM/S in the field of scientific computing as it provides an array of benefits over the methods compared with: explicit data-flow, EDSL, higher order functions, avoidance of true randomness, reasoning, pattern matching,... TODO: mention ABM/S Frameworks, TODO: mention OO solutions. Although the Scala & Actors version might be a competitor in this field, it is not as its strength lies in the development of highly distributed high-availability applications e.g. Twitter and not in simulation where we need fine-control over (global) time and interactions. The other main contribution is to systematically look at the Actor-Model in implementing ABM/S.

2 Related Research

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

2.1 Scientific Computation

TODO: discuss [20] TODO: discuss [18] TODO: discuss [5]

2.2 Haskell

[4] 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." [25]. 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 [22] TODO: discuss [27] TODO: discuss [24] TODO: discuss [19] TODO: discuss [8] TODO: discuss [23]
TODO: check out the internet for Actors/Agents i Haskell, but havn't found anything promising
`http://haskell-distributed.github.io/wiki.html` looks good but too big and not well suited for simulations `https://code.google.com/archive/p/haskellactor/` makes heavy use of IORef and running in IO-Monad, something we deliberately want to avoid to keep the ability to reason about the program. TODO: `https://github.com/fizruk/free-agent` look into

2.3 Erlang

TODO: discuss [9] TODO: discuss [10] TODO: discuss [26]

2.4 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/`
[**?**] 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 [**?**] 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 [7] TODO: discuss [1] TODO: discuss [3] TODO: discuss [2] TODO: discuss [13] TODO: discuss [14]

3 Implementation: General Considerations

All implementations of ABM/S models must solve two problems:

1. Agent-Implementation: how can the Agent in the model-specification be implemented?
2. Simulation-Stepping: which kind of stepping is required or best suited for the given model?

Of course both problems influence each other and cannot be considered separated from each other.
-¿ Java: supports global data =¿ suitable to implement global decisions: implementing global-time, sequential iteration with global decisions -¿ Haskell: has no global data =¿ local decisions (has support for global data through STM/IO but then looses very power?) =¿ implementing global-time, parallel iteration with local-decisions. -¿ Haskell STM solution =¿ implementing concurrent version using STM? but this is very complicated in its own right but utilizing STM it will be much more easier than in java -¿ Scala: mixed, can do both =¿ implementing local time with random iteration and local decisions

3.0.1 Agent-Implementation

This is the process of implementing the behaviour of the Agent as specified in the model. Although there are various kinds of Agent-Models like BDI but the basic principle is always the same: sense the environment, process messages, execute actions: change environment, send messages. According to [29] and also influenced by Actors from [1] one can abstract the abilities in each step of an Agent to be the following:

1. Process received messages
2. Create new Agents
3. Send messages to other Agents
4. Sense (read) the environment
5. Influence (write) the environment

The difference between communicating with the environment and other agents is that the communication with the former one is synchronized, persists and is visible immediately (at least by the agent performing the action) whereas the communication with other agents is asynchronous.

3.0.2 Semantics of a Simulation

When one has implemented the model of behaviour of an Agent one needs to bring the whole simulation to life by enabling the Agents to execute their behaviour in a recurring fashion. This allows an Agent to change the environment by actions and react to changes in the environment, either by other Agents or the environment itself thus resulting in a feedback-loop. There are two ways of looking and implementing such feedback-loops.

Global Stepping In this case the simulation is iterated in global steps where in each step each Agent is updated by running its behaviour.

1. **Sequential** - update one Agent after another. We assume that, given the updates are done in order of the index $i1ton$, then Agents $a_{n>i}$ see the updated agent-state / influence on the environment of agent a_i. Note that if this is not the case we would end up in the parallel-case (see next) *independent* whether it is in fact running in parallel or not. For breaking deterministic ordering which could result in giving an Agent an advantage (e.g. having more information towards the end of the step) one could implement a random-walk in each step but this does not fundamentally change this approach. Also if one thinks the simulation continuously, where each step is just a very small update like in Heroes & Cowards, then the random ordering should not change anything fundamental as no agent has real information-benefit over others as there is continuous iteration thus the agent once ahead is then behind. TODO: maybe need to make more formal

2. **Parallel** - update all Agents in parallel. This case is obviously only possible if the agents cannot interfere with each other or the environment through shared state. In this case it will make no difference how we iterate over the agents, the outcome *has to be* the same - it is event-ordering invariant as all events/updates happen *virtually* at the *same time*. Haskell is a strong proponent of this implementation-technique.

3. **Concurrent** - update all Agents concurrently. In this case the agents run in parallel but share some state which access has to be synchronized thus introducing real random event-orderings which may or may not be desirable in the given simulation model. Can be implemented in both Java and Haskell.

Local Stepping In this case there is no global iteration over steps but all the Agents run in parallel, doing local stepping and communicate with each other either through shared state or messages. Note that this does not impose any specific ordering of the update and can thus regarded to be real random due to its concurrent nature. It is possible to simulate the global-stepping methods from above by introducing some global locking forcing the agents into lock-step. This is the approach chosen for Scala & Actors.

The following table gives an overview of the methods presented above. Real Randomness identifies methods which produce a random ordering of their events due to their implicit workings (e.g. concurrency) as opposed to explicit implementation (e.g random-walk of agents using a random-number generator).

Time	*Order*	*Decisions*	*Non-Deterministic*	*Type*
Global	Sequential	Global	No	Continuous
Global	Parallel	Local	No	Discrete
Global	Concurrent	Global	Yes	Continuous
Local	Random	Local	Yes	Continuous

Table 1: Summary of simulation-stepping methods.

note that different types of update-strategies amount to different types of simulation . all can be used in continuous but discrete is in parallel only
semantics in interface: if the simultion is discrete, use queueMsg, if it is continuous, use sendMsg
central question is: should we hide the semantics from the sgent by providing a single interface e.g. sendMsg and make the semantic explicit only when executing the simulation or should we have different interfaces for explicit semantics e.g. sendMsg and queueMsg? the point is: when we want parallel semantics where all updates happen at once we can also run them technically in parallel. i feel we should make this explicit thus providing a queueMsg which will deliver it at the end of the iteration
global, concurrent = parallel with iterative updates where the following agents see updates. but random ordering introduced due to sync and scheduling
Each of the above presented methods imposes a different kind of event-ordering and thus all will obviously result in different *absolute* simulation results. The point here is that when using ABM/S to study a system one is not interested in individual runs but in replications due to randomness and whether the system shows some emergent behaviour or not. Thus one can ask the question whether the emergent behaviour of a simulation is stable under event-ordering or not. TODO: I have no clue how to show that other than by simulations, this is also a limitation of simulations: just because it does not show up in a run it does not mean that it isn't there, just that it is unlikely - also the reverse is true: just because the emergent behaviour was there in the last n runs, does not mean it is ALWAYS there. For this we need different, more formal methods. But then again, if the level of complexity is too high we cannot solve such systems in closed form and must again fall back to simulation.

4 Implementation in Haskell

It is important to prevent oneself to implement an object-oriented approach in Haskell because then we would loose the power of pure functional programming. Thus the main design-considerations were:

=¿ it seems that par-version is slower than sequential ??? maybe because of evaluating all data?

1. Implement all 4 approaches mentioned in General Implementation for maximum flexibility for users of the library so they can choose which semantics of the simulation they want.
2. Pure simulation to keep up the ability to reason about it. This means to avoid running in IO Monad at all costs (except for the main-loop).
3. Utilizing STM which allows very flexible handling of state and allows running.
4. Run inside Yampa/Dunai to leverage the EDSL and continuous/discrete time-implementation.

take haskell, add yampa and dunai and implement ActorModel on top using STM =¿ have an ABM library in Haskell, put on hackage. NO IO!

4.1 Utilizing STM

with stm one can determine when everything is visible: at the end of the step or after each agent update. but this applies only to parallelism! if running agents in parallel they must execute atomically, otherwise the probability for concurrent read/writes goes to 100% also as the number of agents goes up. if no parallelism then one final atomically at the end of each step enough. but in any case: using STM leads to the effect that agents see later updates. so really 2 cases where STM is of use in haskell: global, sequential and global concurrent how can we implement true parallelism? can we use STM somehow or do we need local mailboxes?
=¿ STM solution is much slower than the par-version without yet using parallel-execution!

4.2 Structuring the Program

Of course the basic pure functional primitives alone do not make a well structured functional program by themselves as the usage of classes, interfaces, objects and inheritance alone does not make a well structured object-oriented program. What is needed are *patterns* how to use the primitives available in pure functional programs to arrive at well structure programs. In object-orientation much work has been done in the 90s by the highly influential book [11] whereas in functional programming the major inventions were also done in the 90s by the invention of Monads through [**?**], [**?**] and [**?**] and beginning of the 2000s by the invention of Arrows through [**?**].

4.2.1 Higher Order Functions & Monads

map & fmap, foldl, applicatives [17] gives a great overview and motivation for using fmap, applicatives and Monads. TODO: explain Monads

4.2.2 Arrows

[**?**] is a great tutorial about *Arrows* which are very well suited for structuring functional programs with effects.

> Just like monads, arrow types are useful for the additional operations they support, over and above those that every arrow provides.

The main difference between Monads and Arrows are that where monadic computations are parameterized only over their output-type, Arrows computations are parameterised both over their input- and output-type thus making Arrows more general.

> In real applications an arrow often represents some kind of a process, with an input channel of type a, and an output channel of type b.

In the work [**?**] an example for the usage for Arrows is given in the field of circuit simulation. They use previously introduced streams to advance the simulation in discrete steps to calculate values of circuits thus the implementation is a form of *discrete event simulation* - which is in the direction we are heading already with ABM/S. Also the paper mentions Yampa which is introduced in the section (TODO: reference) on functional reactive programming.

4.2.3 Functional reactive programming (FRP)

FRP is a paradigm for programming hybrid systems which combine continuous and discrete components. Time is explicitly modelled: there is a continuous and synchronous time flow.

there have been many attempts to implement FRP in frameworks which each has its own pro and contra. all started with fran, a domain specific language for graphics and animation and at yale FAL, Frob, Fvision and Fruit were developed. The ideas of them all have then culminated in Yampa which is the reason why it was chosen as the FRP framework. Also, compared to other frameworks it does not distinguish between discrete and synchronous time but leaves that to the user of the framework how the time flow should be sampled (e.g. if the sampling is discrete or continuous - of course sampling always happens at discrete times but when we speak about discrete sampling we mean that time advances in natural numbers: 1,2,3,4,... and when speaking of continuous sampling then time advances in fractions of the natural numbers where the difference between each step is a real number in the range of [0..1])

time- and space-leak: when a time-dependent computation falls behind the current time. TODO: give reason why and how this is solved through Yampa.
Yampa solves this by not allowing signals as first-class values but only allowing signal functions which are signal transformers which can be viewed as a function that maps signals to signals. A signal function is of type SF which is abstract,

thus it is not possible to build arbitrary signal functions. Yampa provides primitive signal functions to define more complex ones and utilizes arrows [**?**] to structure them where Yampa itself is built upon the arrows: SF is an instance of the Arrow class.

Fran, Frob and FAL made a significant distinction between continuous values and discrete signals. Yampas distinction between them is not as great. Yampas signal-functions can return an Event which makes them then to a signal-stream - the event is then similar to the Maybe type of Haskell: if the event does not signal then it is NoEvent but if it Signals it is Event with the given data. Thus the signal function always outputs something and thus care must be taken that the frequency of events should not exceed the sampling rate of the system (sampling the continuous time-flow). TODO: why? what happens if events occur more often than the sampling interval? will they disappear or will the show up every time?

switches allow to change behaviour of signal functions when an event occurs. there are multiple types of switches: immediate or delayed, once-only and recurring - all of them can be combined thus making 4 types. It is important to note that time starts with 0 and does not continue the global time when a switch occurs. TODO: why was this decided?

[**?**] give a good overview of Yampa and FRP. Quote: "The essential abstraction that our system captures is time flow". Two *semantic* domains for progress of time: continuous and discrete.

The first implementations of FRP (Fran) implemented FRP with synchronized stream processors which was also followed by [**?**]. Yampa is but using continuations inspired by Fudgets. In the stream processors approach "signals are represented as time-stamped streams, and signal functions are just functions from streams to streams", where "the Stream type can be implemented directly as (lazy) list in Haskell...":

```
type Time = Double
type SP a b = Stream a -> Stream b
newtype SF a b = SF (SP (Time, a) b)
```

Continuations on the other hand allow to freeze program-state e.g. through closures and partial applications in functions which can be continued later. This requires an indirection in the Signal-Functions which is introduced in Yampa in the following manner.

```
type DTime = Double

data SF a b =
        SF { sfTF :: DTime -> a -> (SF a b, b)
```

The implementer of Yampa call a signal function in this implementation a *transition function*. It takes the amount of time which has passed since the previous time step and the durrent input signal (a). It returns a *continuation* of type SF a b determining the behaviour of the signal function on the next step (note that exactly this is the place where how one can introduce stateful functions like integral: one just returns a new function which encloses inputs from the previous time-step) and an *output sample* of the current time-step.

When visualizing a simulation one has in fact two flows of time: the one of the user-interface which always follows real-time flow, and the one of the simulation which could be sped up or slowed down. Thus it is important to note that if I/O of the user-interface (rendering, user-input) occurs within the simulations time-frame then the user-interfaces real-time flow becomes the limiting factor. Yampa provides the function embedSync which allows to embed a signal function within another one which is then run at a given ratio of the outer SF. This allows to give the simulation its own time-flow which is independent of the user-interface.

One may be initially want to reject Yampa as being suitable for ABM/S because one is tempted to believe that due to its focus on continuous, time-changing signals, Yampa is only suitable for physical simulations modelled explicitly using mathematical formulas (integrals, differential equations,...) but that is not the case. Yampa has been used in multiple agent-based applications: [**?**] uses Yampa for implementing a robot-simulation, [**?**] implement the classical Space Invaders game using Yampa, the thesis of [**?**] shows how Yampa can be used for implementing a Game-Engine, [**?**] implemented a 3D first-person shooter game with the style of Quake 3 in Yampa. Note that although all these applications don't focus explicitly on agents and agent-based modelling / simulation all of them inherently deal with kinds of agents which share properties of classical agents: game-entities, robots,... Other fields in which Yampa was successfully used were programming of synthesizers (TODO: cite), Network Routers, Computer Music Development and various other computer-games. This leads to the conclusion that Yampa is mature, stable and suitable to be used in functional ABM/S.
Jason Gregory (Game Engine Architecture) defines Computer-Games as "soft real-time interactive agent-based computer simulations".
To conclude: when programming systems in Haskell and Yampa one describes the system in terms of signal functions in a declarative manner (functional programming) using the EDSL of Yampa. During execution the top level signal functions will then be evaluated and return new signal functions (transition functions) which act as continuations: "every signal function in the dataflow graph returns a new continuation at every time step".
"A major design goal for FRP is to free the programmer from 'presentation' details by providing the ability to think in terms of 'modeling'. It is common that an FRP program is concise enough to also serve as a specification for the problem it solves" [**?**]. This quotation describes exactly one of the strengths using FRP in ACE

4.2.4 Dunai

TODO: [21]

4.3 Putting it all together

5 Prototyping

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=

5.1 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
TODO: implement haskell-version with shared-state (STM primitives) without using IO TODO: implement haskell-version which defines abstract types for the simulation

5.1.1 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
simulation :: Model -¿ Double -¿ Int -¿ [Model]
step :: Model -¿ Double -¿ Model
TODO: can we say something about the methods Model can/must/should support?

5.2 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 [6]

5.3 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.

5.4 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.

5.5 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.

5.6 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

5.7 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 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.

5.8 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).

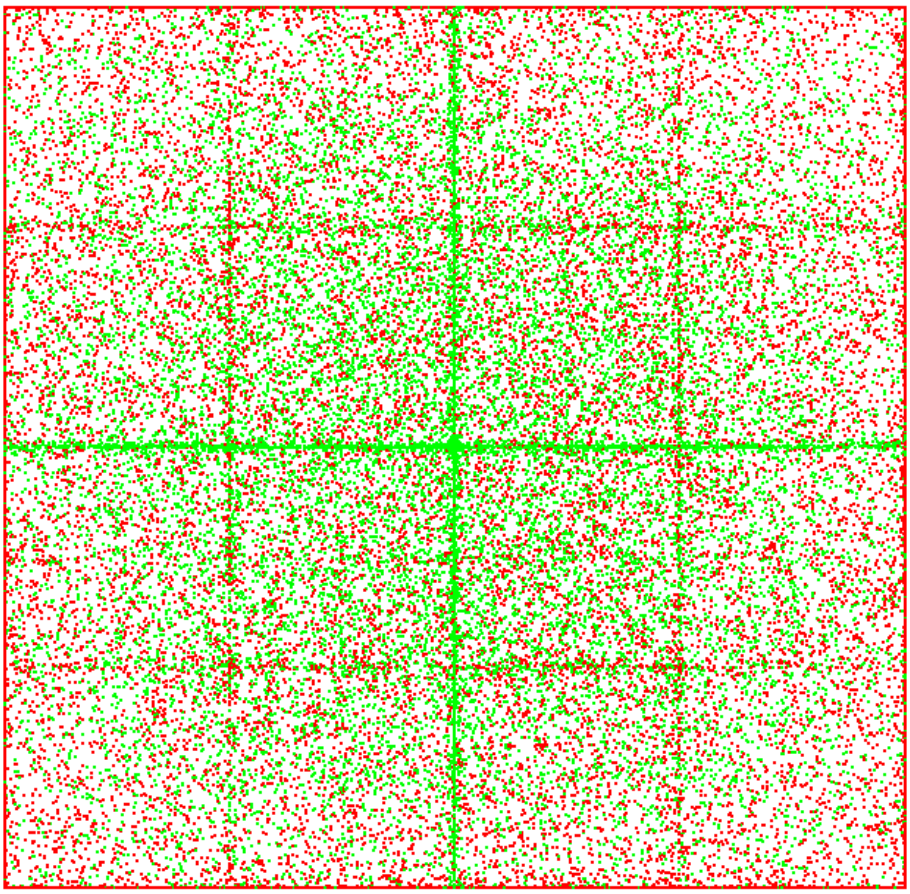

Figure 1: The emergent pattern used as criteria for qualitative comparison of implementations. Note the big green cross in the center and the smaller red crosses in each sub-sector. World-type is *border* with 100.000 Agents where 25% are Heroes.

5.9 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 precalculated random-numbers in range of [0..1], store them in a file and read the trace in all implementations. Needs lots of implementation.

5.10 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?

5.11 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?

5.12 Agent-Representation

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

5.13 EDSL

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

6 Results

7 Conclusions

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

8 Further Research

8.1 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)

8.2 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.

8.3 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.

8.4 Dependent Types

Holy Grail: would solve a specific class of problems with types, but typing and programing becomes more complicated then - also poses problems because everything programmed in this way has to be constructive. For this paper out of focus but will look into this in the main work.

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] Bezirgiannis, N. *Improving Performance of Simulation Software Using Haskell's Concurrency & Parallelism*. PhD thesis, Utrecht University - Dept. of Information and Computing Sciences, 2013.

[5] 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 (Dec. 2011), 4025–4040.

[6] Claessen, K., and Hughes, J. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2000), ICFP '00, ACM, pp. 268–279.

[7] Clinger, W. D. Foundations of Actor Semantics. Tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA, 1981.

[8] De Jong, T. Suitability of Haskell for Multi-Agent Systems. Tech. rep., University of Twente, 2014.

[9] 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.

[10] Di Stefano, A., and Santoro, C. eXAT: an Experimental Tool for Programming Multi-Agent Systems in Erlang. Tech. rep., 2007.

[11] Gamma, E., Helm, R., Johnson, R., Vlissides, J., and Booch, G. *Design Patterns: Elements of Reusable Object-Oriented Software*, 1 edition ed. Addison-Wesley Professional, Oct. 1994.

[12] Grief, I., and Greif, I. SEMANTICS OF COMMUNICATING PARALLEL PROCESSES. Tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA, 1975.

[13] Hewitt, C. What Is Commitment? Physical, Organizational, and Social (Revised). In *Coordination, Organizations, Institutions, and Norms in Agent Systems II*, P. Noriega, J. Vázquez-Salceda, G. Boella, O. Boissier, V. Dignum, N. Fornara, and E. Matson, Eds., no. 4386 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 293–307. DOI: 10.1007/978-3-540-74459-7_19.

[14] Hewitt, C. Actor Model of Computation: Scalable Robust Information Systems. *arXiv:1008.1459 [cs]* (Aug. 2010). arXiv: 1008.1459.

[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] Hutton, G. *Programming in Haskell*. Cambridge University Press, Cambridge, UK ; New York, Jan. 2007.

[18] Ionescu, C., and Jansson, P. Dependently-Typed Programming in Scientific Computing. In *Implementation and Application of Functional Languages* (Aug. 2012), R. Hinze, Ed., Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 140–156. DOI: 10.1007/978-3-642-41582-1_9.

[19] Jankovic, P., and Such, O. Functional Programming and Discrete Simulation. Tech. rep., 2007.

[20] Kamiński, B., and Szufel, P. Verification of Models in Agent Based Computational Economics — Lessons from Software Engineering. In *Perspectives in Business Informatics Research* (Sept. 2013), A. Kobyliński and A. Sobczak, Eds., Lecture Notes in Business Information Processing, Springer Berlin Heidelberg, pp. 185–199. DOI: 10.1007/978-3-642-40823-6_15.

[21] Perez, I., Bärenz, M., and Nilsson, H. Functional Reactive Programming, Refactored. In *Proceedings of the 9th International Symposium on Haskell* (New York, NY, USA, 2016), Haskell 2016, ACM, pp. 33–44.

[22] 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.

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

[24] Sulzmann, M., and Lam, E. Specifying and Controlling Agents in Haskell. Tech. rep., 2007.

[25] Sweeney, T. The Next Mainstream Programming Language: A Game Developer's Perspective. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2006), POPL '06, ACM, pp. 269–269.

[26] 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.

[27] Vendrov, I., Dutchyn, C., and Osgood, N. D. Frabjous: A Declarative Domain-Specific Language for Agent-Based Modeling. In *Social Computing, Behavioral-Cultural Modeling and Prediction*, W. G. Kennedy, N. Agarwal, and S. J. Yang, Eds., no. 8393 in Lecture Notes in Computer Science. Springer International Publishing, Apr. 2014, pp. 385–392. DOI: 10.1007/978-3-319-05579-4_47.

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

[29] Wooldridge, M. *An Introduction to MultiAgent Systems*, 2nd ed. Wiley Publishing, 2009.