Functional methods in agent-based computational economics

Novel approaches for implementing ACE models

The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

Jonathan Andreas Thaler

Supervisor: **Prof. Peer-Olaf Siebers**

Prof. Thorsten Altenkirch

Department of Computer Science

University of Nottingham

This dissertation is submitted for the degree of

Doctor of Philosophy

November 2016

I would like to dedicate this thesis to my loving parents which made this work possible through their all encompassing love and support throughout my life.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Jonathan Andreas Thaler
November 2016

Acknowledgements

And I would like to acknowledge Peer-Olaf Thorsten Altenkirch Hans Vollbrecht Martin Summer

Abstract

This is where you write your abstract ...

Table of contents

List of figures **xiii**

List of tables **xv**

1 Introduction **1**

1.1 Methods . . . 3

1.2 Scenarios . . . 3

1.3 Comparison . . . 3

1.4 Functional programming . . . 4

1.5 Related Research . . . 5

2 Agents **7**

2.1 Agent Models . . . 7

2.1.1 Actors . . . 7

2.1.2 TODO: is FRP an agent-model? . . . 7

2.2 Implementing Agents . . . 8

3 ACE **9**

3.1 TODO: market-models . . . 10

4 Haskell **11**

4.1 Related Research . . . 11

4.2 Structuring pure functional programs . . . 12

4.2.1 Higher Order Functions & Monads . . . 12

4.2.2 Arrows . . . 12

4.3 Frameworks . . . 13

4.3.1 Functional reactive programming (FRP) . . . 13

4.4 Reasoning . . . 16

4.4.1 Time and Semantics . . . 16

4.5 Determinism . . . 16
4.5.1 EDSL . . . 16
4.6 Implementations . . . 17
4.6.1 Testing . . . 18
4.7 Performance . . . 18
4.7.1 Active vs inactive Agents . . . 18
4.7.2 Parallelism and Concurrency . . . 18

5 Scala & Actors 19
5.0.3 The Actor Model . . . 20
5.0.4 Agents vs. Actors . . . 20
5.0.5 Actors are pure functional! . . . 20
5.0.6 Hypothesis . . . 20
5.1 Actor Model implementations . . . 21
5.1.1 Erlang . . . 21
5.1.2 Akka . . . 21
5.1.3 Haskell . . . 21
5.2 Theoretical reflections . . . 22
5.2.1 The problem of time . . . 22
5.2.2 Conversations . . . 22
5.3 Example implementation . . . 22
5.3.1 Wildfire with/without wind . . . 22
5.3.2 Discrete SIR/S . . . 23
5.4 Decentralized bartering . . . 23
5.5 Further Research . . . 23
5.5.1 Pure functional approach . . . 23
5.5.2 Explicit time . . . 23
5.6 Conclusion . . . 23
5.6.1 Real concurrency not needed in simulation . . . 23

6 Lisp 25

7 Java & Repast 27

References 29

List of figures

List of tables

Chapter 1

Introduction

I noticed that it is pretty hard to convince an agent-based economics specialist who is not a computer scientist about a pure functional approach. My conjecture is that the implementation technique and method does not matter much to them because they have very little knowledge about programming and are almost always self-taught - they don't know about software-engineering, nothing about proper software-design and architecture, nothing about software-maintenance, nothing about unit-testing,... In the end they just "hack" the simulation in whatever language they are able to: C++, Visual Basic, Java or toolboxes like Netlogo. For them it is all about to *get things done somehow* and not to get things done the right way or in a beautiful way - the way and the method doesn't matter, its just a necessary evil which needs to be done. Thus if functional programming could make their lives easier, then they will definitely welcome it. But functional programming is, i think, harder to learn and harder to understand - so one needs to provide an abstraction through EDSL. So I REALLY need to come up with convincing arguments why to use pure functional approaches in ACE THEY can understand, otherwise I will be lost and not heard (not published,...).

What ACE economists care for:

- Very: Qualitative modelling with quantitative results
- Yes: Easy reproducibility
- Likely: Reasoning about convergence?
- Likely: EDSL

My contributions are: pure functional framework, functional agent-model for market-simulations, EDSL for market-simulations, qualitative / implicit modelling with quanitative results, reasoning in my framework about convergence

IDEA: could I develop non-causal modelling (models are expressed in terms of non-directed equations, modelled in signal-relations) to allow for qualitative modelling for the agent-based economists? See hybrid modelling paper of Yampa. **THIS WOULD BE A HUGE NOVEL CONTRIBUTION TO ACE ESPECIALLY WHEN COMBINED WITH AN EDSL AND PROVIDING FULL REFERENTIAL TRANSPARENCY TO KEEP THE ABILITY TO REASON ABOUT CONVERGENCE**. This should be covered in the "EDSL"-paper.

TODO: maybe i should really focus only on market models? otherwise too much?

central novelty of my PhD: model specification = runnable code. possible through EDSL. but only in specific subfield of ACE: market-models. need a functional description of the model, then translate it to model specification in EDSL and then run it to see dynamics. But: model specification moves closer to functional programming languages.

another novelty approach: model specification through qualitative instead of quantiative approaches. is this possible?

WHY FUNCTIONAL? "because its the ultimate approach to scientific computing": fewer bugs due to mutable state (why? is thos shown obkectively by someone?), shorter (again as above, productivity), more expressive and closer to math, EDSL, EDSL=model=simulation, better parallelising due to referental transparency, reasoning

scientific results need to be reproduced, especially when they have high impact. a more formal approach of specifying the model and the simulation (model=simulation) could lead to easier sharing and easier reporduction without ambigouites

pure functional agent-model & theory, EDSL framework in Haskell for ACE

1. Which kind of problem do we have?

2. What aim is there? Solving the problem?

3. How the aim is achieved by enumerating VERY CLEAR objectives.

4. What the impact one expects (hypothesis) and what it is (after results).

Note: It is not in the interest of the researcher to develop new economic theories but to research the use of functional methods (programming and specification) in agent-based computational economics (ACE).

NOTE: Get the reader's attention early in the introduction: motivation, significance, originality and novelty.

1.1 Methods

Methods need to be selected to implement the simulations. Special emphasis will be put on functional ones which will then be compared to established methods in the field of ABM/S and ACE.

Claim: non-programming environments are considered to be not powerful enough to capture the complexity of ACE implementations thus a programming approach to ACE will be always required.

1.2 Scenarios

To apply and test functional methods in ACE, four scenarios of ACE are selected and then the methods applied and compared with each other to see how each of them perform in comparison. The 4 selected scenarios represent a selection of the challenges posed in ACE: from very abstract ones to very operational ones.

1.3 Comparison

Each of the selected scenarios is then implemented using the selected methods where each solution is then compared against the following criteria:

1. suitability for scientific computation
2. robustness
3. error-sources
4. testability
5. stability

6. extendability
7. size of code
8. maintainability
9. time taken for development
10. verification & correctness
11. replications & parallelism
12. EDSL

This will then allow to compare the different methods against each other and to show under which circumstances functional methods shine and when they should not be used.

1.4 Functional programming

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 [17] 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. [17] 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 [17] 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-orien
- (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

1.5 Related Research

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." [26]. 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.

Chapter 2

Agents

TODO: general agents: what they are and what they are not

2.1 Agent Models

Agent Models are NOT specific to any programming language implementation but should in theory be implementable in all languages which support the required primitives of the model or which allows the primitives of the model to be mapped to primitives of the language. Of course this says nothing about how well a language is suited to implement a given agent model and how readable and natural the mapping and implementation is.

Start from wooldridge 2.6 (look at the original papers which inspired the 2.6 chapter) and weiss book and the original papers those chapter is based upon. Look into denotational semantics of actor model. Look also in functional models of czesar ionescu. why: this is the major contribution of my thesis and is new knowledge. Must find intuitive, original and creative approach.
From functional agent models (e.g. Wooldridge) to implementation of ACE in Haskell (this should go into research-proposal introduction)

2.1.1 Actors

2.1.2 TODO: is FRP an agent-model?

It is specific to (pure) functional programming languages and can only be implemented with very cumbersome overhead in object-oriented languages like Java. So maybe we can do that in Scala and LISP as well?

2.2 Implementing Agents

In the end it all boils down to 1. the agent-model and 2. how the agent-model is implemented in the according language. Of course both points influence each other: functional languages will come up with a different agent-model (e.g. hybrid like yampa) than object-oriented ones (e.g. actors).

The only constants are: see actors

Chapter 3

ACE

read ACE introduction papers, summarize in this research-proposal

look into computable economics book: http://www.e-elgar.com/shop/computable-economics

TODO: the reading should pull out the essence of what types of ACE there are and what features each type has (continuous/discrete time, complex agent communication, equilibriua, networks amongst agents,...)

NOTE: I REALLY need to work out what is special in ACE? what is the unique property of ACE AS compared to other ABM/S? Conjecture: equilibrium of dynamics is the central aspect. http://www2.econ.iastate.edu/tesfatsi/ace.htm

[4] Agent-based modeling and economic theory: where do we stand? - Ballot, Mandel, Vignes
[25] Agent-based Computational Economics. A Short Introduction - Richiardi
[28] Agent-based computational economics: a constructive approach to economic theory - tesfatsion
[6] Introduction to computer science and economic theory - blume, easley, kleinberg
[27] agent-based computational economics - tesfatsion

"[...] computational modeling of economic processes (including whole economies) as open-ended dynamic systems of interacting agents." Leigh Tesfatsion

The book [21] is a critique of classic economics with the triple of rational agents, "average" inidividuum, equilibrium theory. Although it does not mention ACE it can be seen as an important introduction to the approach of ACE as it introduces many important concepts and views dominant in ACE. Also ACE can be seen as an approach of tackling the problems introduced in this book:

page 6: "the view of economy is much closer to that of social insects than to the traditional view of how economies function."

page 7: "... main argument that it is the interaction between individuals that is at the heart of the explanation of many macroeconomic phenomena..."
page 15: "problem of equilibrium is information"
page 21: "the theme of this book will be that the very fact individuals interact with each other causes aggregate behaviour to be different from that of individuals"

3.1 TODO: market-models

find all agent-based market models (gintis, gode and sunder,...), read gintis equilibrium stuff and summarize in the paper: EDSL for agent-based market-simulations

Chapter 4

Haskell

NOTE: this chapter should be the very first implementation chapter as the approach of Haskell lays the very foundations for the functional approach by introducing the basic functional concepts

So far the literature on agent-based modelling & simulation (ABM/S) hasn't focused much on models for functional agents and is lacking a proper treatment of implementing agents in pure-functional languages like Haskell. This paper looks into how agents can be specified functionally and then be implemented properly in the pure functional language Haskell. The functional agent-model is inspired by wooldridge 2.6. The programming paradigm used to implement the agents in Haskell is functional reactive programming (FRP) where the Yampa framework will be used. The paper will show that specifying and implementing agents in a pure functional language like Haskell has many advantages over classical object-oriented, concurrent ones but needs also more careful considerations to work properly.

4.1 Related Research

[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.

4.2 Structuring pure functional programs

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 [23], [29] and [30] and beginning of the 2000s by the invention of Arrows through [18].

4.2.1 Higher Order Functions & Monads

map & fmap, foldl, applicatives [20] gives a great overview and motivation for using fmap, applicatives and Monads. TODO: explain Monads

4.2.2 Arrows

[19] 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 [19] 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.

4.3 Frameworks

4.3.1 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 [19] 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 signalfunctions 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?

[24] 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 [31]. 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...": [frame=single] 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. [frame=single] 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: [16] uses Yampa for implementing a robot-simulation, [10] implement the classical Space Invaders game using Yampa, the thesis of [22] shows how Yampa can be used for implementing a Game-Engine, [7] 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" [31]. This quotation describes exactly one of the strengths using FRP in ACE

4.4 Reasoning

Give example by showing reasoning in ACE: convergence, correctness,... Look into Graham Huttons Book on Haskell: there are suggestions for further reading

4.4.1 Time and Semantics

[31] discuss the semantic framework of FRP. Very difficult to understand and full of corollaries and theorems and proofs, have to study in depth at another time.

4.5 Determinism

no concurrent execution: deterministic

deterministic: can use random-numbers but to be reproducible/deterministic one has to specify the same seed or even provide an own RNG-implementation (which is easily possible using the RNG in haskell)

it is of most importance in simulations to be reproducible under given conditions: two runs with the same input (e.g. time, agent-count, parameters, RNG seeds) should result in the exact same results otherwise the simulation-software is of very little benefit.

4.5.1 EDSL

In this paper I present an EDSL which allows to formulate models for agent-based market-simulations which can be directly run in a Haskell framework implemented for this. Thus the distinction between model specification and programming vanishes - the model specification becomes the actual code. The major novelty of this EDSL is that it allows to model the system in a qualitative way: relations among formulas are expressed which can be understood as a kind of non-causal modelling. TODO: better understand what qualitative modelling/simulation in ACE is.

4.6 Implementations

idea: can we implement a message between two agents through events? thus two states: waiting for messages, processing messages. BUT: then sending a message *will take some time*

NOTE: it is important to make a difference about whether the simulation will dynamically *add* or *remove* agents during execution. If this is not the case, a simple par-switch is possible to run ALL agent SF in parallel. If dynamically changes to the agent-population should be part of the simulation, then the dpSwitch or dpSwitchB should be used. Also it should be possible to start/stop agents: if they are inactive then they should have no running SF because would use up resources. Inactive means: doing nothing, also not awaiting something/"doing nothing in the sense that DOING something which is nothing - the best criteria to decide if an agent can be set inactive is when the event which decides if the agents SF should be started comes from outside e.g. if the agent is just statically "living" but not changing and then another agent will "ignite" the "living" agent then this is a clear criterion for being static without a running SF.

NOTE: the route-function will be used to distribute "messages" to the agents when they are communicating with each other

NOTE: [22] argues that in Game-Engines (it is paraphrased in english, as the thesis was written in german): "communication among Game-Objects is always computer-game specific and must be implemented always new but the functionality of Game-objects can be built by combining independent functions and signal-functions which are fully reuse-able". Game-Objects can be understood as agents thus maybe this also holds true for agent-based simulation. [22] thus distinguishes between normal functions e.g. mathematical functions, signal functions which depend on output since its creation in localtime and game-object functions which output depends on inputs AND time (which is but another input).

TODO: need a mechanism to address agents: if agent A wants to send a message to agent B and agent B wants to react by answering with a message to agent A then they must have a mechanism to address each other

TODO: design general input/output data-structures

TODO: design general agent SF

TODO: don't loose STM out of sight!
Wormholes in FRP?

4.6.1 Testing

TODO: look into [8]

4.7 Performance

4.7.1 Active vs inactive Agents

signalfunctions add up, multiple chains of events add up, need to remove inactive agents or exclude them somehow from computation chain: use freezing?

4.7.2 Parallelism and Concurrency

[5] puts special emphasis on concurrency and parallelism in implementing simulation framework in Haskell. TODO: explain deeper

Due to the pure-functional property of a agents SF (which fully describes the agents behaviour over time) all signal-functions of all agents should be able to run in parallel in each iteration as they are pure functional: they don't touch global/shared state. Also the routing and switching functions could be sped up using par. But Question: how is this possible in Yampa?

Wormholes in FRP?

Chapter 5

Scala & Actors

This method was selected because Scala is an object-oriented functional programming language and has a powerful library included which implements the actor-model. Because actors and agents are closely related this is an obvious method to follow.

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.

5.0.3 The Actor Model

Read in the following order

1. [15]
2. [12]
3. [9]
4. [1]
5. [3]
6. [13]
7. [14]
8. [2]

upside: extreme huge number of agnts possible due to distributed and parallel technology downside: depends on system & hardware: scheduler, system time, systime resolution (not very nice for scientific computation), much more complicated, debugging difficult due to concurrency, no global notion of time appart from systemtime, thus always runs in real-time, but there is no global notion of time in the actor model anyway, no EDSL full of technical details, no determinism, no reasoning

5.0.4 Agents vs. Actors

Agents more a high-level concept, Actors low level, technical concurrency primitives

5.0.5 Actors are pure functional!

5.0.6 Hypothesis

This makes simulations very difficult and also due to concurrency implementing a sync conversation among agents is very cumbersome. I have already experience with the Actor

Model when implementing a small version of my Master-Thesis Simulation in Erlang which uses the Actor Model as well. For a continuous simulation it was actually not that bad but the problem there was that between a round-trip between 2 agents other messages could have already interfered - this was a problem when agents trade with each other, so one has to implement synchronized trading where only messages from the current agent one trades with are allowed otherwise budget constraints could be violated. Thus I think Erlang/Akka/Actor Model is better suited for distributed high-tolerance concurrent/parallel systems instead for simulations. Note: this is definitely a major point I have to argue in my thesis: why I am rejecting the actor model.

AKKA: thus my prediction is: akka/actor model is very well suited to simulations which 1. dont rely on global time 2. dont have multi-step conversations: interactions among agents which are only question-answer. TODO: find some classical simulation model which satisfies these criterias.

5.1 Actor Model implementations

It is important to note that the actor-model is not tied to any particular programming language and could be (and is) implemented in different types of languages like Java (Object-Oriented), Haskell (pure functional) and Scala (mix of functional and Object-Oriented).

5.1.1 Erlang

TODO: erlang is an old implementation of the actor model

5.1.2 Akka

TODO: akka is a modern implementation of the actor model

5.1.3 Haskell

TODO: can implement actor-model using forkIO and STM

5.2 Theoretical reflections

5.2.1 The problem of time

how can we simulate global time? how can we implement multistep conversations (by futures)?

The real problem seems to be concurrency but i feel we can simulate concurrency by synchronizing to continuous time. computations are carried out after another but because time is explicitly modelled they happen logically at the same time. these rules hold: an agent cannot be in two conversations at the same time, the agent can be in only one or none conversation at a given time t.

What if time is of no importance and only the continuous dynamics are of interest?

To put it another way: real concurrency (with threads) makes time implicit which is what one does NOT want in simulation. Maybe FRP is the way to go because it allows to explicitly model continuous and discrete time, but I have to get into FRP first to make a proper judgement about its suitability.

5.2.2 Conversations

new concept: not single, async messages, but syncronous conversations which (can) take time = multiple synchronous messages between agents which (can) change the state of an agent in the end.

5.3 Example implementation

5.3.1 Wildfire with/without wind

"This is a model of a wildfire. Vegetation is modeled as grid cells - agents in discrete space. The burning time of a cell is proportional to the amount of “fuel” in the cell, which is randomly generated at the model startup. While burning, the cell may cause ignition in adjacent cells. You can cause the initial ignition by clicking a cell. The ignition may also be caused by a bomb dropped by the aircraft - an agent moving in continuous 2D space that overlaps the discrete space. This model, among other things, shows how the two types of space can be linked. The model is computationally very efficient because there are no time steps in this model; the behavior of vegetation cells and the aircraft is defined in the form of a statechart. Unlike in the full version, in this model wind is not taken into consideration."

"This is a model of a wildfire. Vegetation is modeled as grid cells - agents in discrete space. The burning time of a cell is proportional to the amount of “fuel” in the cell, which is randomly generated at the model startup. While burning, the cell may cause ignition in adjacent cells. The probability of ignition depends on the wind direction (which you can change on the fly). You can cause the initial ignition by clicking a cell. The ignition may also be caused by a bomb dropped by the aircraft – an agent moving in continuous 2D space that overlaps the discrete space. This model, among other things, shows how the two types of space can be linked. The model is computationally very efficient because there are no time steps in this model; the behavior of vegetation cells and the aircraft is defined in the form of a statechart."

1. No time steps.
2. One-Step interaction.

5.3.2 Discrete SIR/S

1. Discrete time steps.
2. One-Step interaction.

5.4 Decentralized bartering

1. Continuous time.
2. Multi-Step interactions.

5.5 Further Research

5.5.1 Pure functional approach

5.5.2 Explicit time

5.6 Conclusion

5.6.1 Real concurrency not needed in simulation

AKKA is nice but I think the actor model is not very well suited for simulations due to inherent concurrency where time is implicit.

real concurrency is not needed: simultaneous events can be modeled through explicit time but calculated sequential - when we reduce agents to process only one message after another and not multiple concurrently. thus true parallelism is only a technical detail for performance enhancement.

Chapter 6

Lisp

Why LISP: the oldest language and functional. Has extremely powerful properties: homoiconic: data = code = data. We ask how those can be utilized to implement agent-based simulations in economics.

NEPI$^2 framework$

Chapter 7

Java & Repast

Is a state-of-the-art OO method. This is the golden standard to be compared to the other functional approaches to show where the differences are and what the up- and downs are.

These tools are state-of-the-art in ABM/S and ACE and are included to show how one can perform scenarios (see below) with these tools. Java is the state-of-the-art programming language in ABM/S and ACE and is thus included as well as a benchmark against such a state-of-the-art.

References

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

[2] Agha, G. (2004). 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*, pages 26–57. Springer.

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

[4] Ballot, G., Mandel, A., and Vignes, A. (2015). Agent-based modeling and economic theory: where do we stand? *Journal of Economic Interaction and Coordination*, 10(2):199–220.

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

[6] Blume, L., Easley, D., Kleinberg, J., Kleinberg, R., and Tardos, E. (2015). Introduction to computer science and economic theory. *Journal of Economic Theory*, 156(C):1–13.

[7] Cheong, M. H. (2005). Functional programming and 3d games. Master's thesis, University of New South Wales, Sydney, Australia.

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

[9] Clinger, W. D. (1981). Foundations of actor semantics. Technical report, Cambridge, MA, USA.

[10] Courtney, A., Nilsson, H., and Peterson, J. (2003). The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, pages 7–18, New York, NY, USA. ACM.

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

[12] Grief, I. and Greif, I. (1975). Semantics of communicating parallel processes. Technical report, Cambridge, MA, USA.

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

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

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

[16] Hudak, P., Courtney, A., Nilsson, H., and Peterson, J. (2003). *Arrows, Robots, and Functional Reactive Programming*, pages 159–187. Springer Berlin Heidelberg, Berlin, Heidelberg.

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

[18] Hughes, J. (2000). Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111.

[19] Hughes, J. (2005). Programming with arrows. In *Proceedings of the 5th International Conference on Advanced Functional Programming*, AFP'04, pages 73–129, Berlin, Heidelberg. Springer-Verlag.

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

[21] Kirman, A. (2010). *Complex Economics: Individual and Collective Rationality*. Routledge, London ; New York, NY.

[22] Meisinger, G. (2010). Game-engine-architektur mit funktional-reaktiver programmierung in haskell/yampa. Master's thesis, Fachhochschule Oberösterreich - Fakultät für Informatik, Kommunikation und Medien (Campus Hagenberg).

[23] Moggi, E. (1989). Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Piscataway, NJ, USA. IEEE Press.

[24] Nilsson, H., Courtney, A., and Peterson, J. (2002). Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 51–64, New York, NY, USA. ACM.

[25] Richiardi, M. (2007). Agent-based computational economics. a short introduction. LABORatorio R. Revelli Working Papers Series 69, LABORatorio R. Revelli, Centre for Employment Studies.

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

[27] Tesfatsion, L. (2002). Agent-based computational economics. Computational economics, EconWPA.

[28] Tesfatsion, L. (2006). Agent-based computational economics: A constructive approach to economic theory. In Tesfatsion, L. and Judd, K. L., editors, *Handbook of Computational Economics*, volume 2, chapter 16, pages 831–880. Elsevier, 1 edition.

[29] Wadler, P. (1990). Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78, New York, NY, USA. ACM.

[30] Wadler, P. (1995). Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, UK. Springer-Verlag.

[31] Wan, Z. and Hudak, P. (2000). Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 242–252, New York, NY, USA. ACM.