Functional Reactive ABMS: Towards pure functional Agent-Based Modelling & Simulation.

Jonathan Thaler
jonathan.thaler@nottingham.ac.uk
School of Computer Science, University of Nottingham

Peer-Olaf Siebers
peer-olaf.siebers@nottingham.ac.uk
School of Computer Science, University of Nottingham

Thorsten Altenkirch
thorsten.altenkirch@nottingham.ac.uk
School of Computer Science, University of Nottingham

Abstract

The pure functional paradigm has so far got not very much attention in ABS where the dominant programming paradigm is object-orientation, realized by Java. In preliminary research of ours, we came to the insight that pure functional programming using Haskell without any additional libraries is very well suited to implement ABS but that in order to build more complex, reald-world models it would be necessary to leverage the basic concepts with functional reactive programming (FRP). In this paper we show how ABS can be fused with FRP to design an embedded domain specific language (EDSL) which allows to write specifications which (nearly) match pure functional code - the specification becomes Haskell code. For a proof of concept, we chose to implement the famous and well known Sugarscape model from the social sciences and the ACE Trading World from agent-based computational economics (ACE).

Keywords: Agent-Based Simulation, Functional Programming, Reactive Programming, Haskell

1. Introduction

TODO: talk about preliminary research and its drawbacks: need something like FRP / Yampa to leverage TODO: need a formal specification

TODO: this is a paper for working out the depth of one of the directions of the PhD: ACE, formulate research-questions

2. Background

2.1. Agent Based Simulation

We understand ABS as 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. Those parts, called agents, are modelled and simulated out of which then the aggregate global behaviour of the whole system emerges. So the central aspect of ABS is the concept of an agent which can be understood as a metaphor for a pro-active unit, situated in an environment, able to spawn new agents and interacting with other agents in a network of neighbours by exchange of messages [**?**]. It is important to note that we focus our understanding of ABS on a very specific kind of agents where the focus is on communicating entities with individual, localized behaviour from out of which the global behaviour of the system emerges. We informally assume the following about our agents:

- They are uniquely addressable entities with some internal state.
- They can initiate actions on their own e.g. change their internal state, send messages, create new agents, kill themselves.
- They can react to messages they receive with actions as above.

- They can interact with an environment they are situated in.

We will give a formal model of agents in a subsequent section (TODO: reference)

2.2. Pure functional paradigm

TODO: short description what pure functional paradigm is 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 [?] 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. [?] 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 [?] argues that in this ability functional languages are superior to structured programming.

2.2.1. Naive ABS implementation. TODO: shortcomings 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 [?] 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 [?].

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

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.

2.3. 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

TODO: short description what FRP is, cite the necessary papers TODO: hypothesis that FRP is a remedy for the shortcomings

2.4. Models

2.4.1. Sugarscape. [?] introduced the famous *Sugarscape* model which is widely researched in the social sciences. TODO: short explanation We included this model because it is well known and we wanted to test how much effort it takes to implement it with our approach. Because its complexity is moderate and there are only homogeneous agents it allows us to really show the power of our EDSL.

2.5. ACE Trading World

[?] gives a broad overview of agent-based computational economics (ACE), gives the four primary objectives of it and discusses advantages and disadvantages. She introduces a model called *ACE Trading World* in which she shows how an artificial economy can be implemented without the *Walrasian Auctioneer* but just by agents and their interactions. She gives a detailed mathematical specification in the appendix of the paper which should allow others to implement the simulation. We included this model because of its complexity and different, heterogeneous types of agents.

2.6. Verification and Validation in ABS

TODO: `http://jasss.soc.surrey.ac.uk/12/1/1.html` TODO: `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4419595` TODO: `http://dspace.stir.ac.uk/handle/1893/3365#.WNjO1DsrKM8` TODO: `http://www2.econ.iastate.edu/tesfatsi/DockingSimModels.pdf` TODO: `http://www2.econ.iastate.edu/tesfatsi/empvalid.htm` TODO: `http://jasss.soc.surrey.ac.uk/10/2/8.html` TODO: `https://www.openabm.org/faq/how-validate-and-calibrate-agent-based-models` TODO: `https://link.springer.com/chapter/10.1007%2F978-3-642-01109-2_10` TODO: `http://www3.nd.edu/~nom/Papers/ADS019_Xiang.pdf`

2.7. Related Research

The amount of research on using the pure functional paradigm using Haskell in the field of ABS has been moderate so far. Most of the papers look into how agents can be specified using the belief-desire-intention paradigm [?], [?], [?]. A library for Discrete Event Simulation (DES) and System Dynamics (SD) in Haskell called *Aivika 3* is described in [?]. It comes with very basic features for ABS but only allows to specify simple state-based agents with timed transitions. [?] discuss using functional programming for discrete event simulation (DES) and mention the paradigm of FRP to be very suitable to DES.

[?] present an EDSL for Haskell allowing to specify Agents using the BDI model. TODO: We don't go there, thats not our intention.

[?] and [?] present a domain-specific language for developing functional reactive agent-based simulations. This language called FRABJOUS is very human readable and easily understandable by domain-experts. It is not directly implemented in FRP/Haskell/Yampa but is compiled to Haskell/Yampa code which they claim is also readable. This is the direction we want to head but we don't want this intermediate step but look for how a most simple domain-specific language embedded in Haskell would look like. We also don't touch upon FRP and Yampa yet but leave this to further research for another paper of ours.

[?] TODO

TODO: cite julie greensmith paper on haskell

3. The formal Agent-Model

TODO: define an agent: state message-protocol behaviour environment representation

api: create new agent kill self send message to

events: message received time-advanced start stop

4. Functional Reactive ABS

the fundamental problem is that unlike in oo e.g. java there are no objects and no implicit aliases through which to acess and change data: method calls are not there in FP. we must silve the problem of how to represent an agent and how agents can interact with each other

using example SIRS or Wildfire TODO: show how simple state-transition agents work using switch TODO: show how the time-semantics can be used

4.1. Parallel vs. Sequential

TODO: cite my own work

4.2. Environment

TODO: again cite my own work where I discussed the problem of environments

Each agent has a copy of the environment passed in through the AgentIn and can change it by passing a changed version of the environment out through AgentOut. In the sequential update-strategy the environment of the agent i will then be passed to all agents i + 1 AgentIn in the current iteration and override their old environment. Thus all steps of changes made to the environment are visible in the AgentOuts. The last environment is then the final environment int he current iteration and will be returned by the callback function together with the current AgentOuts. In the parallel update-strategy the environment is duplicated to each agent and then each agent can work on it and return the changed environment. Thus after one iteration there are n versions of environments where n is equals to the number of agents. These environments must then be collapsed into a final one which is always domain-specific thus needs to be done through a function provided in the environment itself. In both the sequential and parallel update-strategy after one iteration there is one single environment left. An environment can have an optional behaviour which allows the environment to update its cells. This is a regular SF thus having also the time of the simulation available. Note that the environment cannot send messages to agents because it is not an agent itself. An example of an environment behaviour would be to regrow some good on each cell according to some rate per time-unit (inspired by SugarScape regrowing of Sugar).

4.3. Messaging

TODO: again cite my own work where I discussed the problem of queued messaging

Each Agent can send a message to an other agent where the messages are queued and can be processed when the agent is updated the next time. The agent is free to ignore the messages and if it does not process them they will not be here in the next update.

because we dont have method call in FP it became clear that communicating through method calls is a interaction which takes no time in simulation, which was not so cleae before. but we need this mechanism due to the need of 2 agents acting sync in no-time.

4.3.1. Conversation. When sending messages between agents there is the limitation that an agent cannot reply to a message within the same iteration. In some models this is not a problem as it is not required by the model to reply to messages e.g. in the case of the SIR-Model where an agent just sends a 'Contact Infected' messages which infects the receiver with a given probability but the receiver does not reply to this message. This limitation is for some models a huge problem which prevents a faithful implementation of it in

FrABS. For this we introduce the concept of a conversation between agents. This allows an agent A to initiate a conversation with another agent B in which the simulation is and halted both can exchange an arbitrary number of messages through calling and responsing without time passing (something not possible without this concept because in each iteration the time advances). After either one agent has finished with the conversation it will terminate it and the simulation will continue with the updated agents.

4.3.2. Sampling Time. my wrongthinking: messaging ALWAYS takes time e.g. send/response roundtrip. conversations dont take time but are restricted for the receiver e.g. the receiver cannot send messages to others or change the environment in a conversation

because an agent cannot reply within the same timestep sampling interval becomes an issue: if we need a reply within a given time then the sampling interval needs to be at least twice as much

difference between discrete and continuous: the successor of discrete is defined whereas in the case of continuous it is not? how is the successor defined in the case of continuous time?

5. Conclusion and future research

advantages: - no side-effects within agents leads to much safer code - edsl for time-semantics - declarative style

further research - verification & validation