Spec == Code:
Pure functional Agent-Based Modelling

Jonathan Thaler

February 13, 2017

Abstract

Building upon our previous work on update-strategies in Agent-Based Simulation ABS where we showed that Haskell is a very attractive alternative to existing object-oriented approaches we ask in this paper if the declarative power of the pure functional language can be utilized to write specifications for simple models of ABS which can be directly translated to our Haskell implementation. We take two well known examples: the SIR and Prisoners Dilemma and express them in our model-specification language and show that they are in fact equals to haskell-code. Thus the novelty of this paper is a formalization of the specification of the latter one, which has been so far neglected.

Keywords

Agent-Based Simulation, Agent-Based Modelling, Haskell, SIRS, Prisoner Dilemma

1 Introduction

the specification language should not be too technical, its focus should be on non-technical expressiveness. The question is: can we abstract away the technicalities and still translate it directly to haskell (more or less)? If not, can be adjust our Haskell implementation to come closer to our specification language? Thus it is a two-fold approach: both languages need to come closer to each other if we want to close the gap
TODO: [2], [1] it is still not exactly clear =¿ we present a formal specification using ABS
it is not trivial to reproduce the results as there is only very informal descriptions in [2]. [1] give a few more details but also stay quite informal. Thus here we represent a pure functional formulation of the original program which makes it formally exactly clear how the simulation should work. we then look how it can be translated to an ABM specification
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.
The code is available at `https://github.com/thalerjonathan/phd/tree/master/coding/papers/declarativeABM/haskell/MinABS`

2 Related Research

[4] present an EDSL for Haskell allowing to specify Agents using the BDI model. We don't go there, thats not our intention.
[3] and [5] 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: cite julie greensmith paper on haskell

We don't focus on BDI or similar but want to rely much more on low-level basic messaging. We can also draw strong relations to Hoare's Communicating Sequential Processes (CSP), Milner's Calculus of Communicating Systems (CCS) and Pi-Calculus. By mapping the EDSL to CSP/CCS/Pi-Calculus we achieve to be able to algebraic reasoning in our EDSL. TODO: hasn't Agha done something similar in connecting Actors to the Pi-Calculus?

3 Background

for exact specification, we also need the update-strategy. we show that there is very small differences in code but, as reported in [**?**] make huge differences in the results. note that the difference between SEQ and PAR in Haskell is in the end a 'fold' over the agents in the case of SEQ and a 'map' in the case of PAR

4 The specification Language

TODO: include environment (continuous, discrete, graph) and position within the environment
introduce environments: discrete, continuous, graph
An Agent A is a 5-tuple ¡aid, s, m, tf¿ aid is the id of the agent s is the generic state of the agent m is the message-protocoll the agent understands
tf is the transformer-function with the following type Agent -¿ Event m -¿ Agent further there is a function for sending a message to other agents send :: Agent -¿ (Aid, m) -¿ Agent
For calculating one iteration of the system we first pair up all agents with the messages they receive in this iteration and then we step every agent iteration :: [Agent] -¿ [Agent] iteration as = stepAll . collectAll
To pair up every agent with all its receiving messages we collect all messages for each single agent collectAll :: [Agent] -¿ [(Agent, [(Aid, m)])] collectAll as = map (¿ (a, collectFor a as)) as
To pair up a single Agent with its receiving messages, we go over all agents and collect all messages for this current agent collectFor :: Agent -¿ [Agent] -¿ [(Aid, m)] collectFor a as = foldl (a' -¿ acc : collectFrom a' aid) [] as where aid = Aid a
To collect all messages from an agent for a given receiver we simply filter all messages in the outgoing messagebox for the receiving id and then we map the resulting list by replacing the receiving id with the sending id collectFrom :: Agent -¿ Aid -¿ [(Aid, m)] collectFrom a rid = map ($(\_, m) -> (sid, m))msgswhere sid = Aid a ms = filter((rid', m) -> rid == rid')(mbox a)$
To step all agents we simply apply step to all agents-message pairs by using map stepAll :: [(Agent, [(Aid, m)])] -¿ [Agent] stepAll = map step
To step an agent we first clear the message-box of the agent, then let consume it its messages and then advance the time step :: (Agent, [(Aid, m)]) -¿ Agent step = advanceTime . consumeMessages

TODO: need to clear the message-box but then the elegant function.composition breaks down but this is probably the most elegant play where to do it
Consuming th messages means starting with the initial agent applying the transformation-function for each message resulting in a new agent in every message consumeMessages :: (Agent, [(Aid, m)]) -¿ Agent consumeMessages (a, ms) = foldl tf a ms where tf = (tf a)
To advance time we simply send the Dt message with the time-step delta to the tranformer-function advanceTime :: Agent -¿ Agent advanceTime a = tf a (-1, Dt 1.0) where tf = (tf a)

5 SIRS Specification

This section provides the specification of the SIR model.

5.1 SIRS Model

TODO: explanation of it in natural language

5.2 Formal Specification

TODO: first the formal 'mathematical' specification, then the translation to haskell-code
First we declare the 5 tuples of our SirAgent. It has a domain-specific state, message-protocol and transformer-function as can be seen in the types SirAgent = ¡Aid, sirState, sirMsg, mbox, sirTf¿
We define the domain-specific state to be the following. Each Agent is in one of the 3 SIR states and knows its duration it has been in this state. SIR = Susceptible , Infected, Recovered sirState = (state element of SIR, durInState :: Double)
We define the domain-specific message-protocol to cover the possibilities of making contact. sirMsg = ContactSusceptible, ContactInfected, ContactRecovered
Next we declare our transformer-function and handle all types of messages here. Note that in the case of domain-specific messages we only handle ContactInfected as in the other cases the model-specification in natural language does not cover these cases: the agent is not affected or changed in these contacts. sirTf :: SirAgent -¿ (Aid, Msg sirMsg) -¿ SirAgent sirTf a (, Dtd) = $sirTimeStepasirTfa(,ContactInfected) = infectedContactasirTfa_{=}a$
In each discrete time-step which happens in the sir-model as specified in the natural-language description an agent first recovers and then makes contact with neighbours sirTimeStep :: SirAgent -¿ SirAgent sirTimeStep = makeContact . recover
Making contact with other neighbours is straight-forward, using send makeContact :: SirAgent -¿ SirAgent makeContact a — is a Susceptible = send a

(receiver, ContactSusceptible) — is a Infected = send a (receiver, ContactInfected) — is a Recovered = send a (receiver, ContactRecovered) where receiver = randomNeighbour
We introduced the is function which returns true if the SirAgent is in the given state is :: SirAgent -¿ SIR -¿ Boolean is a s = (sirState a) == s
Now we define how an Agent recovers. An Agent can only recover if it is actually infected, otherwise there is no need to recover recover :: SirAgent -¿ SirAgent recover a — is a Infected = recoverInfected a — otherwise = a
If an agent is indeed infected, it has recovered when it has spent the given time-steps in the infection-state recoverInfected :: SirAgent -¿ SirAgent recoverInfected a = if (durInState a ¿= I) then a state = Recovered, durInState = 0.0 else a durInState = durInState' where durInState' = (durInState a) + 1.0
Finally we need to describe how an agent is infected when having contact with an other agent which is infected infect :: SirAgent -¿ SirAgent infect a — yes = a state = Infected, durInState = 0.0 — otherwise = a where yes = p ¿= uniformRandom [0.0, 1.0]

6 Prisoners Dilemma Specification

1. calculate local payoff for all. INCLUDE ITSELF!!! otherwise it would not work. this is not 100% clear in both papers as it is quite not logical: why play the prisoner-dilemma with one self? if one does not include itself, the patterns don't emerge. 2. calculate local best for all based upon local payoff

7 Heores & Cowards

TODO: include because also very short and CONTINUOUS environment

8 Results

9 Conclusion

10 Further Research

References

[1] Huberman, B. A., and Glance, N. S. Evolutionary games and computer simulations. *Proceedings of the National Academy of Sciences 90*, 16 (Aug. 1993), 7716–7718.

[2] Nowak, M. A., and May, R. M. Evolutionary games and spatial chaos. *Nature 359*, 6398 (Oct. 1992), 826–829.

[3] 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.

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

[5] 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.