

Dependent types in Agent-Based Simulation

JONATHAN THALER, THORSTEN ALTENKIRCH, and PEER-OLAF SIEBERS, University of Nottingham, United Kingdom

TODO: cite my own 1st paper from SSC2017: add it to citations

TODO: refine it: start with simulating epidemics and then go into ABS

Agent-Based Simulation (ABS) is a methodology in which a system is simulated in a bottom-up approach by modelling the micro interactions of its constituting parts, called agents, out of which the global macro system behaviour emerges. So far, the Haskell community hasn't been much in contact with the community of ABS due to the latter's primary focus on the object-oriented programming paradigm. This paper tries to bridge the gap between those two communities by introducing the Haskell community to the concepts of ABS. We do this by deriving an agent-based implementation for the simple SIR model from epidemiology. In our approach we leverage the basic concepts of ABS with functional reactive programming from Yampa and Dunai which results in a surprisingly fresh, powerful and convenient EDSL for programming ABS in Haskell.

Additional Key Words and Phrases: Haskell, Functional Programming, Functional Reactive Programming, Agent-Based Simulation

ACM Reference Format:

Jonathan Thaler, Thorsten Altenkirch, and Peer-Olaf Siebers. 2018. Dependent types in Agent-Based Simulation. *Proc. ACM Program. Lang.* 9, 4, Article 39 (September 2018), 3 pages. <https://doi.org/0000001.0000001>

1 INTRODUCTION

Dependent Types are the holy grail in functional programming as they allow to express even stronger guarantees about the correctness of programs and go as far where programs and types become constructive proofs [7] which must be total by definition [6], [2], [1], [5]. Thus the next obvious step is to apply them to our pure functional approach of agent-based simulation. So far no research in applying dependent types to agent-based simulation exists at all and it is not clear whether dependent types do make sense in this setting. We explore this for the first time and ask more specifically how we can add dependent types to our pure functional approach, which conceptual implications this has for ABS and what we gain from doing so. Note that we can only scratch the surface and lay down basic ideas and leave a proper in-depth treatment of this topic for further research. We use Idris [3], [4] as language of choice as it is very close to Haskell with focus on real-world application and running programs as opposed to other languages with dependent types e.g. Agda and Coq which serve primarily as proof assistants.

Dependent Types promise the following:

- (1) Types as proofs - In dependently types languages, types can depend on any values and are first-class objects themselves. TODO: make more clear

Authors' address: Jonathan Thaler, jonathan.thaler@nottingham.ac.uk; Thorsten Altenkirch, thorsten.altenkirch@nottingham.ac.uk; Peer-Olaf Siebers, University of Nottingham, 7301 Wollaton Rd, Nottingham, NG8 1BB, United Kingdom, peer-olaf.siebers@nottingham.ac.uk.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery. 2475-1421/2018/9-ART39 \$15.00
<https://doi.org/0000001.0000001>

- (2) **Totality and termination** - Constructive proofs must terminate, this means a well-typed program (which is itself a proof) is always terminating which in turn means that it must consist out of total functions. A total function is defined by [4] as: it terminates with a well-typed result or produces a non-empty finite prefix of a well-typed infinite result in finite time. Idris is turing complete but is able to check the totality of a function under some circumstances but not in general as it would imply that it can solve the halting problem. Other dependently typed languages like Agda or Coq restrict recursion to ensure totality of all their functions - this makes them non turing complete.

2 DEPENDENTLY TYPED SIR

A SIR model enters a steady state as soon as there are no more infected agents. Thus we can informally argue that a SIR model must always terminate as:

- (1) Only infected agents can infect susceptible agents.
- (2) Eventually after a finite time every infected agent will recover.
- (3) There is no way to move from the consuming *recovered* state back into the *infected* state¹.

Thus a SIR model must enter a steady state after a finite amount of time. Using the DE presented in the introduction on the SD model we can show that the system enters a stable state in finite time (TODO: are there any references?):

TODO: show that it must terminate through the SD formulas. also calculate the t for when $I(t) = 0$

This result gives us the confidence, that the agent-based approach will terminate, given it is really a correct implementation of the SD model. Still this does not proof that the agent-based approach itself will terminate and so far no proof of the totality of it was given. Dependent Types and Idris ability for totality and termination checking should theoretically allow us to proof that an agent-based SIR implementation terminates after finite time: if an implementation of the agent-based SIR model in Idris is total it is a proof by construction. Note that such an implementation should not run for a limited virtual time but run unrestricted of the time and the simulation should terminate as soon as there are no more infected agents. We hypothesize that it should be possible due to the nature of the state transitions where there are no cycles and that all infected agents will eventually reach the recovered state. Abandoning the FRP approach and starting fresh, the question is how we implement a *total* agent-based SIR model in Idris. Note that in the SIR model an agent is in the end just a state-machine thus the model consists of communicating / interacting state-machines. In the book [4] the author discusses using dependent types for implementing type-safe state-machines, so we investigate if and how we can apply this to our model. We face the following questions: how can we be total? can we even be total when drawing random-numbers? Also a fundamental question we need to solve then is how we represent time: can we get both the time-semantics of the FRP approach of Haskell AND the type-dependent expressivity or will there be a trade-off between the two?

TODO: implement sir with state-machine approach from Idris. an idea would be to let infected agents generate infection- actions: the more infected agents the more infection-actions => zero infected agents mean zero infection actions. this list can then be reduced?

REFERENCES

- [1] Thorsten Altenkirch, Nils Anders Danielsson, Andres Låŕŕ, and Nicolas Oury. 2010. Pi_Sigma: Dependent Types Without the Sugar. In *Proceedings of the 10th International Conference on Functional and Logic Programming (FLOPS'10)*. Springer-Verlag, Berlin, Heidelberg, 40–55. https://doi.org/10.1007/978-3-642-12251-4_5

¹There exists an extended SIR model, called SIRS which adds a cycle to the state-machine by introducing a transition from recovered to susceptible but we don't consider that here.

- [2] Thorsten Altenkirch, Conor McBride, and James Mckinna. 2005. Why dependent types matter. In *In preparation*, <http://www.e-pig.org/downloads/ydtm.pdf>.
- [3] EDWIN BRADY. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 05 (2013), 552–593. <https://doi.org/10.1017/S095679681300018X>
- [4] Edwin Brady. 2017. *Type-Driven Development with Idris*. Manning Publications Company. Google-Books-ID: eWzE-jwEACAAJ.
- [5] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. <https://homotopytypetheory.org/book>, Institute for Advanced Study.
- [6] Simon Thompson. 1991. *Type Theory and Functional Programming*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.
- [7] Philip Wadler. 2015. Propositions As Types. *Commun. ACM* 58, 12 (Nov. 2015), 75–84. <https://doi.org/10.1145/2699407>

Received March 2018; revised March 2018; accepted March 2018