Dependent Types in Agent-Based Simulation

An Agent-Based Approach

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

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 system behaviour emerges.

So far mainly object-oriented techniques and languages have been used in ABS but in previous research we have demonstrated how to do ABS with pure functional programming and shown that it has already unique benefits over the traditional object-oriented approaches. In this research we go one step further and investigate what dependent types can offer for ABS and if we can gain even more from it. We primarily focus on agent interactions, totality of a simulation and the philosophy behind the constructiveness of ABS and Dependent Types in combination. TODO: refine, need to add 2 more sentences on our findings

Additional Key Words and Phrases: Dependent Types, Verification, Validation, Agent-Based Simulation

ACM Reference Format:
Jonathan Thaler, Thorsten Altenkirch, and Peer-Olaf Siebers. 2019. Dependent Types in Agent-Based Simulation: An Agent-Based Approach. 1, 1 (May 2019), 17 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Previous research (TODO: cite my own paper on Pure Functional Epidemics) has shown that the pure functional programming paradigm as in Haskell is very suitable to implement agent-based simulations. Building on FRP and MSFs the work developed an elegant implementation of an agent-based SIR model which was pure. By statically removing all external influences of randomness already at compile time through types, this guarantees that repeated simulation runs with the same starting conditions will always result in the same dynamics - guaranteed at compile time. This previous research focused only on establishing the basic concepts of ABS in functional programming but it did not explore the inherent strength of functional programming for verification and correctness any further than guaranteeing the reproducibility of the simulation at compile time.

This paper picks up where the previous research has left and wants to investigate the usefulness of pure and dependently typed functional programming for verification and correctness of agent-based simulation. We are especially interested if requirements of an ABS can be guaranteed on a stronger level by those paradigms, if a larger class of bugs can be excluded already at compile time and whether we can express model properties and invariants already at compile time on a type level. Further we are interested in how far we can reason about an agent-based model in a dependently typed implementation.

As use cases we introduce two well known models in ABS. First, the simple SIR model of epidemiology [11] with which one can simulate epidemics, that is the spreading of an infectious disease through a population, in a realistic way. It has the benefit that there exists an analytical solution for

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

2019. XXXX-XXXX/2019/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

it against which one can validate the agent-based implementation. Second, the Sugarscape model [9] which simulates artificial societies.

We first look look into the general potential of dependent types in ABS, derive possible applications and patterns and then apply them in both use-case models.

The aim of this paper is to investigate what dependent types can offer to ABS and what the benefits and drawbacks are. By doing this we give the reader a good understanding of what ABS is, what the challenges are when implementing it and how we solve these in our approach.

TODO: refine The contributions of this paper are:

- To the best of our knowledge, we are the first to *systematically* investigate what dependent types can offer for implementing pure functional agent-based simulations.
- Our approach shows how one can encode agent-based model specifications directly into the types which guarantees correctness on a much stronger level, unprecedented by our previous research and not possible with traditional object-oriented approaches.
- Our approach shows how to prove properties of agent-based models and simulation and express them where then the implementation is both prove and specification and will generate dynamics. this implies that we do not need separate proofs for properties because our dependently typed implementation *is* the proof.
- Our approach shows a total implementation of the agent-based SIR model, which is by definition a constructive proof that the agent-based SIR model will terminate after finite number of steps.
- We explore the philosophical implications and connections of the fact that both ABS and Dependent Types are inherently constructivist approaches.

Section 2 discusses related work. In section 3 we introduce the concepts of agent-based simulation and both use-case models. TODO: not add other sections. Finally, we draw conclusions and discuss issues in section 7 and point to further research in section 8.

2 RELATED WORK

The authors of [2] are using functional programming as a specification for an agent-based model of exchange markets but leave the implementation for further research where they claim that it requires dependent types. This paper is the closest usage of dependent types in agent-based simulation we could find in the existing literature and to our best knowledge there exists no work on general concepts of implementing pure functional agent-based simulations with dependent types. As a remedy to having no related work to build on, we looked into works which apply dependent types to solve real world problems from which we then can draw inspiration from.

The authors of [7] use depend types to implement correct-by-construction concurrency in the Idris language [3]. They introduce the concept of a Embedded Domain Specific Language (EDSL) for concurrently locking/unlocking and reading/writing of resources and show that an implementation and formalisation are the same thing when using dependent types. We can draw inspiration from it by taking into consideration that we might develop a EDSL in a similar fashion for specifying general commands which agents can execute. The interpreter of such a EDSL can be pure itself and doesn't have to run in the IO Monad as our previous research (TODO: cite my PFE paper) has shown that ABS can be implemented pure.

In [8] the authors discuss systems programming with focus on network packet parsing with full dependent types in the Idris language [3]. Although they use an older version of it where a few features are now deprecated, they follow the same approach as in the previous paper of constructing an EDSL and and writing an interpreter for the EDSL. In a longer introduction of Idris the authors discus its ability for termination checking in case that recursive calls have an argument which is structurally smaller than the input argument in the same position and that these arguments belong to a strictly positive data type. We are particularly interested in whether we can implement an agent-based simulation which termination can be checked at compile time - it is total.

In [4] the author discusses programming and reasoning with algebraic effects and dependent types in the Idris language [3]. They claim that monads do not compose very well as monad transformer can quickly become unwieldy when there are lots of effects to manage. As a remedy they propose algebraic effects and implement them in Idris and show how dependent types can be used to reason about states in effectful programs. In our previous research (TODO: cite my PFE paper) we relied heavily on Monads and transformer stacks and we indeed also experienced the difficulty when using them. Algebraic effects might be a promising alternative for handling state as the global environment in which the agents live or threading of random-numbers through the simulation which is of fundamental importance in ABS. Unfortunately algebraic effects cannot express continuations (according to the authors of the paper) which is but of fundamental importance for pure functional ABS as agents are on the lowest level built on continuations - synchronous agent interactions and time-stepping builds directly on continuations. Thus we need to find a different representation of agents - GADTs seem to be a natural choice as all examples build heavily on them and they are very flexible.

In [10] the authors apply dependent types to achieve safe and secure web programming. This paper shows how to implement dependent effects, which we might draw inspiration from of how to implement agent-interactions which, depending on their kind, are effectful e.g. agent-transactions or events.

In [5] the author introduces the ST library in Idris, which allows a new way of implementing dependently typed state machines and compose them vertically (implementing a state machine in terms of others) and horizontally (using multiple state machines within a function). In addition this approach allows to manage stateful resources e.g. create new ones, delete existing ones. We can draw further inspiration from that approach on how to implement dependently typed state machines, especially composing them hierarchically, which is a common use case in agent-based models where agents behaviour is modelled through hierarchical state-machines. As with the Algebraic Effects, this approach doesn't support continuations (TODO: is this so?), so it is not really an option to build our architecture for our agents on it, but it may be used internally to implement agents or other parts of the system. What we definitely can draw inspiration from is the implementation of the indexed Monad *STrans* which is the main building block for the ST library.

The book [6] is a great source to learn pure functional dependently typed programming and in the advanced chapters introduces the fundamental concepts of dependent state machine and dependently typed concurrent programming on a simpler level than the papers above. One chapter discusses on how to implement a messaging protocol for concurrent programming, something we can draw inspiration from for implementing our synchronous agent interaction protocols.

The authors of [14] apply dependent types to FRP to avoid some run-time errors and implement a dependently typed version of the Yampa library in Agda. FRP was the underlying concept of implementing agent-based model we took on in our previous approach (TODO: cite). We could have taken the same route and lift FRP into dependent types but we chose explicitly to not go into this direction and look into complementing approaches on how to implement agent-based models.

The fundamental difference to all these real-world examples is that in our approach, the system evolves over time and agents act over time. A fundamental question will be how we encode the monotonous increasing flow of time in types and how we can reflect in the types that agents act over time.

3 BACKGROUND

TODO: what shall we write here? about dependent types in general?

3.1 Sugarscape

TODO

Sugarscape is an exploratory model inspired by real-world phenomenon which means it has lots of hypotheses implicit in the model but there does not exist real-world data / dynamics against which one could validate the simulated dynamics. Still we can conduct black-box verification because we have an informal model specification but we cannot do any statistical testing of simulated dynamics as we don't have data acting as ground-truth. But what we can do and what we will explore extensively in this section is how we can encode hypotheses about the dynamics (prior to running the simulation) in unit- and property-based tests and check them. Obviously white-box verification applies as well because we can reason about the code whether it matches the informal model specification or not.

3.2 Verification & Validation in ABS

TODO

Validation & Verification in ABS http://www2.econ.iastate.edu/tesfatsi/VVAccreditationSimModels.OBalci1998.pdf : verification = are we building the model right? validation = are we building the right model?

good paper http://www2.econ.iastate.edu/tesfatsi/VVAccreditationSimModels.OBalci1998.pdf : very nice 15 guidelines and life cycles, VERY valuable for background and introduction

http://www2.econ.iastate.edu/tesfatsi/VVSimulationModels.JKleijnen1995.pdf : suggests good programming practice which is extremely important for high quality code and reduces bugs but real world practice and experience shows that this alone is not enough, even the best programmers make mistakes which often can be prevented through a strong static or a dependent type system already at compile time. What we can guarantee already at compile time, doesn't need to be checked at run-time which saves substantial amount of time as at run-time there may be a huge number of execution paths through the simulation which is almost always simply not feasible to check (note that we also need to check all combinations). This paper also cites modularity as very important for verification: divide and conquer and test all modules separately. this is especially easy in functional programming as composability is much better than with traditional oop due to the lack of interdependence between data and code as in objects and the lack of global mutable state (e.g. class variables or global variables) - this makes code extremely convenient to test. The paper also discusses statistical tests (the t test) to check if the outcome of a simulation is sufficiently close to real-world dynamics. Also the paper suggests using animations to visualise the processes within the simulation for verification purposes (of course they note that animation may be misleading when one focuses on too short simulation runs).

good paper:https://link.springer.com/chapter/10.1007/978-3-642-01109-2_10 -> verification. "This is essentially the question: does the model do what we think it is supposed to do? Whenever a model has an analytical solution, a condition which embraces almost all conventional economic theory, verification is a matter of checking the mathematics." -> validation: "In an important sense, the current process of building ABMs is a discovery process, of discovering the types of behavioural rules for agents which appear to be consistent with phenomena we observe." => can we encode phenomena we observe in the types? can we use types for the discovery process as well? can dependent types guide our exploratory approach to ABS? -> "Because such models are based on simulation, the lack of an analytical solution (in general) means that verification is harder, since there is no single result the model must match. Moreover, testing the range of model outcomes provides a test only in respect to a prior judgment on the plausibility of the potential range of outcomes. In this sense, verification blends into validation."

either one has an analytical model as the basis of an agent-based model (ABM) or one does not. In the former case, e.g. the SIR model, one can very easily validate the dynamcis generated by the ABM to the one generated by the analytical solution (e.g. through System Dynamics). Of course the dynamics wont be exactly the same as ABS discretisizes the approach and introduces stochastics which means, one must validate averaged dynamics. In the latter case one has basically no idea or description of the emergent behaviour of the system prior to its execution. It is important to have some hypothesis about the emergent property / dynamics. The question is how verification / validation works in this setting as there is no formal description of the expected behaviour: we don't have a ground-truth against which we can compare our simulation dynamics. (eventuell hilft hier hans vollbrecht weiter: Simulation hat hier den Sinn, die Controller anhand der Roboteraufgabe zu validieren, Bei solchen Simulationen ist man interessiert an allen mÃűglichen Sequenzen, und da das meist zu viele sind, an einer mÃűglichst gut verteilten Stichprobenmenge. Hier geht es weniger um richtige Zeitmodellierung, sondern um den Test aller mÃűglichen Ereignissequenzen.)

look into DEVS

TODO: the implementation phase is just one stage in a longer process http://jasss.soc.surrey.ac.uk/12/1/1.html

WE FOCUS ON VERIFICATION important: we are not concerned here with validating a model with the real world system it simulates. this is an entirely different problem and focuses on the questions if we have built the right model. we are interested here in extremely strong verification: have we built the model right? we are especially interested in to which extend purely and dependently-typed functional programming can support us in this task.

http://jasss.soc.surrey.ac.uk/8/1/5.html: "For some time now, Agent Based Modelling has been used to simulate and explore complex systems, which have proved intractable to other modelling approaches such as mathematical modelling. More generally, computer modelling offers a greater flexibility and scope to represent phenomena that do not naturally translate into an analytical framework. Agent Based Models however, by their very nature, require more rigorous programming standards than other computer simulations. This is because researchers are cued to expect the unexpected in the output of their simulations: they are looking for the 'surprise' that shows an interesting emergent effect in the complex system. It is important, then, to be absolutely clear that the model running in the computer is behaving exactly as specified in the design. It is very easy, in the several thousand lines of code that are involved in programming an Agent Based Model, for bugs to creep in. Unlike mathematical models, where the derivations are open to scrutiny in the publication of the work, the code used for an Agent Based Model is not checked as part of the peer-review process, and there may even be Intellectual Property Rights issues with providing the source code in an accompanying web page."

http://jasss.soc.surrey.ac.uk/12/1/1.html: "a prerequisite to understanding a simulation is to make sure that there is no significant disparity between what we think the computer code is doing and what is actually doing. One could be tempted to think that, given that the code has been programmed by someone, surely there is always at least one person - the programmer - who knows precisely what the code does. Unfortunately, the truth tends to be quite different, as the leading figures in the field report, including the following: You should assume that, no matter how carefully you have designed and built your simulation, it will contain bugs (code that does something different to what you wanted and expected), "Achieving internal validity is harder than it might seem. The problem is knowing whether an unexpected result is a reflection of a mistake in the programming, or a surprising consequence of the model itself. [âĂę] As is often the case, confirming that the model was correctly programmed was substantially more work than programming the model in the first place. This problem is particularly acute in the case of agent-based simulation. The complex and exploratory nature of most agent-based models implies that, before running a model, there is some uncertainty about what the model will produce. Not knowing a priori what to expect makes it difficult to discern whether an unexpected outcome has been generated as a legitimate result of the assumptions embedded in the model or, on the contrary, it is due to an error or an artefact created in the model design, its implementation, or its execution."

general requirements to ABS - modelling progress of time (steward robinson simulation book, chapter 2) - modelling variability (steward robinson simulation book, chapter 2) - fixing random number streams to allow simulations to be repeated under same conditions (steward robinson simulation book, chapter 1.3.2 and chapter 2) - only rely on past -> solved with Arrowized FRP - bugs due to implicitly mutable state -> can be ensured by pure functional programming - ruling out external sources of non-determinism / randomness -> can be ensured by pure functional programming - correct interaction protocols -> can be ensured by dependent state machines - deterministic time-delta -> TODO: can we ensure it through dependent-types at type-level? - repeated runs lead to same dynamics -> can be ensured by pure functional programming

steward robinson simulation book bulletpoints - chapter 8.2: speed of coding, transparency, flexibility, run-speed - chapter 8.3: three activities - 1 coding, 2 testing verification and white-box validating, 3 documenting - chapter 9.7: nature of simulation: terminating vs. non-terminating - chapter 9.7: nature of simulation output: transient or steady-state (steady-state cycle, shifting steady-state)

steward robinson simulation book on implementation - meaning of implementation -> 1 implementing the findings: conduct a study which defines and gathers all findings about the model and document them -> 2 implementing the model -> 3 implementing the learning

steward robinson simulation book on verification, validation and confidence - Verification is the process of ensuring that the model design has been transformed into a computer model with sufficient accuracy (Davis 1992) - Validation is the process of ensuring that the model is sufficiently accurate for the purpose at hand (Carson 1986). - Verification has a narrow definition and can be seen as a subset of the wider issue of validation - In Verification and validation the aim is to ensure, that the model is sufficiently accurate, which always implies its purpose. - => the purpose / objectives mus be known BEFORE it is validated - white-box validation: detailed, micro check if each part of the model represent the real world with sufficient accuracy -> intrinsic to model coding - black-box validation: overall, macro check whether the model provides a sufficiently accurate representation of the real world system -> can only be performed once model code is complete - other definition of verification: it is a test of the fidelity with which the conceptual model is converted into the computer model - verification (and validation) is a continuous process => if it is already there in the programming language / supported by it e.g. through types,... then this is much easier to do - difficulties of verification and validation -> there is no such thing as general validity: a model should be built for one purpose as simple as possible and not be too general, otherwise it becomes too bloated and too difficult / impossible to analyse -> there may be no real world to compare against: simulations are developed for proposed systems, new production / facilities which dont exist yet. -> which real world?: the real world can be interpreted in different ways => a model valid to one person may not be valid to another -> often the real world data are inaccurate -> there is not enough time to verify and validate everything -> confidence, not validity: it is not possible to prove that a model is valid, instead one should think of confidence in its validity. => verification and validation is thus not the proof that a model is correct but trying to prove that the model is incorrect, the more tests/checks one carries out which show that it is NOT incorrect, the more confidence we can place on the models validity - methods of verification and validation -> conceptual model validation: judment based on the documentation -> data validation: analysing data for inconsistencies -> verification and white-box validation -> both conceptually different but often treated together because both occur continuously through model coding -> what should be checked: timings (cycle times, arrival times,...), control of elements (breakdown frequency, shift patterns), control flows (e.g. routing), control logic (e.g. scheduling, stock replenishment), distribution sampling (samples obtained from an empirial distribution) -> verification and whilte-box validation methods -> checking code: reading through code and ensure right data and logic is there. explain to others/discuss together/others should look at your code. -> Visual checks -> inspecting output reports

-> black-box testing: consider overall behaviour of the model without looking into its parts, basically two ways -> comparison with the real system: statistical tests -> comparison with another model (e.g. mathematical equations): could compare exactly or also through statistical tests ->

peers slides: - Model testing (verification and validation) -> Required to place confidence in a study's results -> Model testing is not a process of trying to demonstrate that the model is correct but a process of trying to prove that the model is incorrect!

- Model verification: The process of ensuring that the model design has been transformed into a computer model with sufficient accuracy - Model validation: The process of ensuring that the model is sufficiently accurate for the purpose at hand -> models are not meant to be completely accurate -> models are supposed to be build for a specific purpose

- Data Validation: Determining that the contextual data and the data required for model realisation and validation are sufficiently accurate for the purpose at hand.

- white-Box Validation: Determining that the constituent parts of the computer model represent the corresponding real world elements with sufficient accuracy for the purpose at hand (micro check) -> how: Checking the code, visual checks, inspecting output reports

- Black-Box Validation: Determining that the overall model represents the real world with sufficient accuracy for the purpose at hand (macro check) -> comparison with the real system -> comparison with other (simpler) models

- Experimentation Validation: Determining that the experimental procedures adopted are providing results that are sufficiently accurate for the purpose at hand. -> How can we do this? - Graphical or statistical methods for determining warm-up period, run length and replications (to obtain accurate results) - Sensitivity analysis (to improve the understanding of the model)

- Solution Validation: Determining that the results obtained from the model of the proposed solution are sufficiently accurate for the purpose at hand -> How does this differ from Black Box Validation? Solution validation compares the model of the proposed solution to the implemented solution while black-box validation compares the base model to the real world -> How can we do this? Once implemented it should be possible to validate the implemented solution against the model results

- Verification: Testing the fidelity with which the conceptual model is converted into the computer model. Verification is done to ensure that the model is programmed correctly, the algorithms have been implemented properly, and the model does not contain errors, oversights, or bugs.

-> How can we do this? Same methods as for white-box validation (checking the code, visual checks, inspecting output reports) but ... Verification compares the content of the model to the conceptual model while white-box validation compares the content of the model to the real world

- Difficulties of verification and validation -> There is no such thing as general validity: a model is only valid with respect to its purpose -> There may be no real world to compare against -> Which real world? Different people have different interpretations of the real world -> Often real world data are inaccurate: If the data are not accurate it is difficult to determine if the model's results are correct. Even if the data is accurate, the real world data are only a sample, which in itself creates inaccuracy -> There is not enough time to verify and validate every aspect of a model

- Some final remarks: -> V&V is a continuous and iterative process that is performed throughout the life cycle of a simulation study. Example: If the conceptual model is revised as the project progresses it needs to be re-validated -> V&V work together by removing barriers and objections to model use and hence establishing credibility.

- Conclusion: Although, in theory, a model is either valid or not, proving this in practice is a very different matter. It is better to think in terms of confidence that can be placed in a model!

TODO: explore ABS testing in pure functional Haskell - we need to distinguish between two types of testing/verification -> 1. testing/verification of models for which we have real-world data or an analytical solution which can act as a ground-truth. examples for such models are the SIR model, stock-market simulations, social simulations of all kind -> 2. testing/verification of models which are just exploratory and which are only be inspired by real-world phenomena. examples for such models are Epsteins Sugarscape and Agent_Zero

4 CONCEPTS OF DEPENDENT TYPES IN AGENT-BASED SIMULATION

Independent of the programming paradigm, there exist fundamentally two approaches implementing agent-based simulation: time- and event-driven. In the time-driven approach, the simulation is stepped in fixed Δt and all agents are executed at each time-step - they act virtually in lock-step at the same time. The approach is inspired by the theory of continuous system dynamics (TODO: cite). In the event-driven approach, the system is advanced through events, generated by the agents, and the global system state changes by jumping from event to event, where the state is held constant in between. The approach is inspired by discrete event simulation (DES) (TODO: citation) which is formalized in the DEVS formalism [18].

In a preceding paper we investigated how to derive a time-driven pure functional ABS approach in Haskell (TODO: cite my paper). We came to quite satisfactory results and implemented also a number of agent-based models of various complexity (TODO: cite schelling, sugarscape, agent zero). Still we identified weaknesses due to the underlying functional reactive programming (FRP) approach. It is possible to define partial implementations which diverge during runtime, which may be difficult to determine for complex models for a programmer at compile time. Also sampling the system with fixed Δt can lead to severe performance problems when small Δt are required, as was shown in our paper. The later problem is well known in the simulation community and thus as a remedy an event-driven approach was suggested [12]. In this paper for the first time, we derive a pure functional event-driven agent-based simulation. Instead of using Haskell, which provides already libraries for DES [15], we focus on the dependently typed pure functional programming language Idris. In our previous paper we hypothesised that dependent types may offer interesting new insights and approaches to ABS but it was unclear how exactly we can make use of them, which was left for further research. In this paper we hypothesise that, as opposed to a time-driven approach, the even-driven approach is especially suited to make proper use of dependent types due to its different nature. Note that both a pure functional event-driven approach to ABS *and* the use of dependent types in ABS has so far never been investigated, which is the unique contribution of this paper. If we can construct a dependently typed program of the SIR ABM which is total, then we have a proof-by-construction that the SIR model reaches a steady-state after finite time

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 [17] which must be total by definition [16], [1], [**?**], [13]. 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], [6] 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
(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 [6] 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.

dependent-types: -> encode model-invariants on a meta-level -> encode dynamics (what? feedbacks? positive/negative) on a meta-level -> totality equals steady-state of a simulation, can enforce totality if required through type-level programming -> probabilistic types can encode probability distributions in types already about which we can then reason -> can we encode objectives in types? -> agents as dependently typed continuations?: need a dependently typed concept of a process over time

As shown in our previous research (TODO: cite), the strong static type system of Haskell allows us to guarantee a lot already at compile time:

- Purity - no side-effects possible at all
- Monad - controlled, explicit side-effects possible
- generic types allow to guarantee for all

Dependent Types in Idris bring the strong static type system of Haskell to a new level, which allows us to both guarantee more things at compile time and express things through type-level computations. This means the following at compile time

- Ruling out ever larger classes of bugs
- Dependent State Machines
- Dependent Agent Interactions
- Flow Of Time
- Totality
- Constructive Proofs

4.1 Ruling out Bugs

(1) Index out of bounds access of Lists and Vectors can be guaranteed not to happen any more when using proofs of existence of the element in the list or vector.
(2) Size of list or vector stays constant / increases / decreases / sum of length of multiple vectors guaranteed to be of some number

The question is how far we can generalise our approaches because we fear that the downside of using dependently typed abs is that every implementation needs to start from Scratch: we cant write a general library for it like chimera because the more we put into types, the more specific it is => individual implementation which reuses existing 'patterns' like state machines, messages,...

4.2 General Agent Interface

using dependent types to specify the general commands available for an agent. here we can follow the approach of an DSEL as described in [7] and write then an interpreter for it. It is of importance that the interpreter shall be pure itself and does not make use of any fancy IO stuff.

4.3 Dependent State Machines

dependent state machines in abs for internal state because that is very Common in ABS. Here we can draw inspiration from the paper [5] and book [6].

4.4 Environment

4.5 Dependent Agent Interactions

4.5.1 Agent Transactions. dependently typed message protocols in ABS because its very common, and easily done thorugh methods in OOP: sugarscape mating and trading protocol using a DSEL [7] to restrict the available primitives in the message protocol?

4.5.2 Data Flow. TODO: can dependent types be used in the Data Flow Mechanism?

4.5.3 Event Scheduling. TODO: can dependent types be used in the event-scheduling mechanism?

4.6 Flow Of Time

TODO: can dependent types be used to express the flow of time and its strongly monotonic increasing?

4.7 Totality

totality of parts or the whole simulation e.g. in case of the SIR model we can informally reason that the simulation MUST reach an equilibrium (a steady state from which there is no escape: the dynamics wont't change anymore, derivations are 0) after a finite number of steps. if we can construct a total program which expresses this, we have a formal proof of that which is 1) a specification of the model 2) generates the dynamics 3) is a proof that it reaches equilibrium

4.8 Constructive Proofs

- An agent-based model and the simulated dynamics of it is itself a constructive proof which explain a real-world phenomenon sufficiently good - proof of the existence of an agent: holds always only for the current time-step or for all time, depending on the model. e.g. in the SIR model no agents are removed from / added to the system thus a proof holds for all time. In sugarscape agents are removed / added dynamically so a proof might become invalid after a time or one can construct a proof only from a given time on e.g. when one wants to prove that agent X exists but agent X is only created at time t then before time t the prove cannot be constructed and is uninhabited and only inhabited from time t on. -

4.9 Environment Access

OK: we have solved this basically in our prototyping

Accessing the environment in section **??** involves indexed array access which is always potentially dangerous as the indices have to be checked at run-time.

Using dependent types it should be possible to encode the environment dimensions into the types. In combination with suitable data types for coordinates one should be able to ensure already at compile time that access happens only within the bounds of the environment.

One of the main advantages of Agent-Based Simulation over other simulation methods e.g. System Dynamics is that agents can live within an environment. Many agent-based models place their agents within a 2D discrete NxM environment where agents either stay always on the same cell or can move freely within the environment where a cell has 0, 1 or many occupants. Ultimately this boils down to accessing a NxM matrix represented by arrays or a similar data structure. In imperative languages accessing memory always implies the danger of out-of-bounds exceptions *at run-time*. With dependent types we can represent such a 2d environment using vectors which carry their length in the type (TODO: discuss them in background) thus fixing the dimensions of such a 2D discrete environment in the types. This means that there is no need to drag those bounds around explicitly as data. Also by using dependent types like Fin which depend on the dimensions we can enforce at compile time that we can only access the data structure within bounds. If we want to we can also enforce in the types that the environment will never be an empty one where N, M > 0.

```
Disc2dEnv : (w : Nat) -> (h : Nat) -> (e : Type) -> Type
Disc2dEnv w h e = Vect (S w) (Vect (S h) e)

data Disc2dCoords : (w : Nat) -> (h : Nat) -> Type where
  MkDisc2dCoords : Fin (S w) -> Fin (S h) -> Disc2dCoords w h

centreCoords : Disc2dEnv w h e -> Disc2dCoords w h
centreCoords {w} {h} _ =
    let x = halfNatToFin w
        y = halfNatToFin h
    in  mkDisc2dCoords x y
  where
    halfNatToFin : (x : Nat) -> Fin (S x)
    halfNatToFin x =
      let xh   = divNatNZ x 2 SIsNotZ
          mfin = natToFin xh (S x)
      in  fromMaybe FZ mfin

setCell :  Disc2dCoords w h
        -> (elem : e)
        -> Disc2dEnv w h e
        -> Disc2dEnv w h e
setCell (MkDisc2dCoords colIdx rowIdx) elem env
    = updateAt colIdx (\col => updateAt rowIdx (const elem) col) env

getCell :  Disc2dCoords w h
        -> Disc2dEnv w h e
        -> e
getCell (MkDisc2dCoords colIdx rowIdx) env
    = index rowIdx (index colIdx env)
```

```
neumann : Vect 4 (Integer, Integer)
neumann = [         (0,  1),
           (-1,  0),         (1,  0),
                    (0, -1)]

moore : Vect 8 (Integer, Integer)
moore = [(-1,  1), (0,  1), (1,  1),
         (-1,  0),          (1,  0),
         (-1, -1), (0, -1), (1, -1)]

-- TODO: can we express that n <= len?
filterNeighbourhood :  Disc2dCoords w h
                    -> Vect len (Integer, Integer)
                    -> Disc2dEnv w h e
                    -> (n ** Vect n (Disc2dCoords w h, e))
filterNeighbourhood {w} {h} (MkDisc2dCoords x y) ns env =
    let xi = finToInteger x
        yi = finToInteger y
    in  filterNeighbourhood' xi yi ns env
  where
    filterNeighbourhood' :  (xi : Integer)
                         -> (yi : Integer)
                         -> Vect len (Integer, Integer)
                         -> Disc2dEnv w h e
                         -> (n ** Vect n (Disc2dCoords w h, e))
    filterNeighbourhood' _ _ [] env = (0 ** [])
    filterNeighbourhood' xi yi ((xDelta, yDelta) :: cs) env
      = let xd = xi - xDelta
            yd = yi - yDelta
            mx = integerToFin xd (S w)
            my = integerToFin yd (S h)
        in case mx of
            Nothing => filterNeighbourhood' xi yi cs env
            Just x  => (case my of
                        Nothing => filterNeighbourhood' xi yi cs env
                        Just y  => let coord      = MkDisc2dCoords x y
                                       c          = getCell coord env
                                       (_ ** ret) = filterNeighbourhood' xi yi cs env
                                   in  (_ ** ((coord, c) :: ret)))
```

4.10 State Transitions

TODO: is currently under research in our prototyping

In the SIR implementation one could make wrong state-transitions e.g. when an infected agent should recover, nothing prevents one from making the transition back to susceptible.

Using dependent types it might be possible to encode invariants and state-machines on the type level which can prevent such invalid transitions already at compile time. This would be a huge benefit for ABS because many agent-based models define their agents in terms of state-machines.

4.11 Time-Dependent Behaviour

TODO: is currently under research in our prototyping

An infected agent recovers after a given time - the transition of infected to recovered is a timed transition. Nothing prevents us from *never* doing the transition at all.

With dependent types we might be able to encode the passing of time in the types and guarantee on a type level that an infected agent has to recover after a finite number of time steps.

4.12 Safe Agent-Interaction

TODO: is currently under research in our prototyping

In more sophisticated models agents interact in more complex ways with each other e.g. through message exchange using agent IDs to identify target agents. The existence of an agent is not guaranteed and depends on the simulation time because agents can be created or terminated at any point during simulation.

Dependent types could be used to implement agent IDs as a proof that an agent with the given id exists *at the current time-step*. This also implies that such a proof cannot be used in the future, which is prevented by the type system as it is not safe to assume that the agent will still exist in the next step.

4.13 Interaction Protocolls

TODO: is currently under research in our prototyping

Using dependent types we should be able to encode a protocol for agent-interactions which e.g. ensures on the type-level that an agent has to reply to a request or that a more specific protocol has to be followed e.g. in auction- or trading-simulations.

4.14 Totality

TODO: is currently under research in our prototyping

In our implementation, we terminate the SIR model always after a fixed number of time-steps. We can informally reason that restricting the simulation to a fixed number of time-steps is not necessary because the SIR model *has to* reach a steady state after a finite number of steps. This means that at that point the dynamics won't change any more, thus one can safely terminate the simulation. Informally speaking, the reason for that is that eventually the system will run out of infected agents, which are the drivers of the dynamic. We know that all infected agents will recover after a finite number of time-steps *and* that there is only a finite source for infected agents which is monotonously decreasing.

Using dependent types it might be possible to encode this in the types, resulting in a total simulation, creating a correspondence between the equilibrium of a simulation and the totality of its implementation. Of course this is only possible for models in which we know about their equilibria a priori or in which we can reason somehow that an equilibrium exists.

5 AN INDEXED AGENT MONAD

Drawing inspiration from the papers on real-world application of dependent types using Idris we came to the conclusion that the best starting point to implement agents is to define an indexed agent monad using GADTs.

```
data Agent : (m : Type -> Type) ->
             (ty : Type) ->
             (sa_pre : Type) ->
             (sa_post : ty -> Type) ->
             (evt : Type) ->
             (w : Nat) ->
             (h : Nat) ->
             (t : Nat) -> Type
```

The agent monads' indices are:

(1) *m* - an underlying computation (monadic) context
(2) *ty* - the result type of the respective agent command
(3) *sa_pre* - some internal agent state previous to running the agent command
(4) *sa_post* - some internal agent state after running the agent command, depending on the result type of the command
(5) *evt* - an event / agent-interaction protocol
(6) *w,h* - 2d environment boundary parameters
(7) *t* - current simulation time-step

The agent monad supports a number of low level commands acting on the monad, including the mandatory Pure and Bind (»=) operations which make them a monad.

TODO: define low level agent-commands

The commands the agent monad supports are on a very low level and for implementations of concrete agent-models it is convenient to provide higher level abstractions. We follow the same approach as in [5] and implement an interface which describes the high level operations an agent of a specific model supports where the types of the operations of such an interface all build on the indexed agent monad. The interface itself is then used to implement the agent behaviour of the specific model. Finally an instance of the interface class must be provided which interprets the commands in a given computational context - this gives us incredible freedom e.g. we can provide implementations running in a mock-up state monad or a debugging IO monad which allows to print to the console.

```
instance SIRAgent (m : Type -> Type) where
        TODO: give interface operations

-- this is the debugging IO implementation
SIRAgent IO where

-- this is the mockup test implementation
SIRAgent (StateT MockupData Identity) where
```

6 DEPENDENTLY TYPED SIR

Intuitively, based upon our model and the equations we can argue that the 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* or *susceptible* state [1].

Thus a SIR model must enter a steady state after finite steps / in finite time.

[1]There 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.

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 [6] 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: express in the types – SUSCEPTIBLE: MAY become infected when making contact with another agent – INFECTED: WILL recover after a finite number of time-steps – RECOVERED: STAYS recovered all the time

– SIMULATION: advanced in steps, time represented as Nat, as real numbers are not constructive and we want to be total – terminates when there are no more INFECTED agents

show formally that abs does resemble the sd approach: need an idea of a proof and then implement it in dependent types: look at 3 agent system: 2 susceptible, 1 infected. or maybe 2 agents only

6.1 Environment Access

OK: we have solved this basically in our prototyping

Accessing the environment in section **??** involves indexed array access which is always potentially dangerous as the indices have to be checked at run-time.

6.2 State Transitions

TODO: is currently under research in our prototyping

In the SIR implementation one could make wrong state-transitions e.g. when an infected agent should recover, nothing prevents one from making the transition back to susceptible.

6.3 Time-Dependent Behaviour

TODO: is currently under research in our prototyping

An infected agent recovers after a given time - the transition of infected to recovered is a timed transition. Nothing prevents us from *never* doing the transition at all.

6.4 Safe Agent-Interaction

TODO: is currently under research in our prototyping

In more sophisticated models agents interact in more complex ways with each other e.g. through message exchange using agent IDs to identify target agents. The existence of an agent is not guaranteed and depends on the simulation time because agents can be created or terminated at any point during simulation.

6.5 Interaction Protocolls

TODO: is currently under research in our prototyping

TODO: does this occur in SIR?

6.6 Totality

TODO: is currently under research in our prototyping

In our implementation, we terminate the SIR model always after a fixed number of time-steps. We can informally reason that restricting the simulation to a fixed number of time-steps is not necessary because the SIR model *has to* reach a steady state after a finite number of steps. This means that at that point the dynamics won't change any more, thus one can safely terminate the simulation. Informally speaking, the reason for that is that eventually the system will run out of infected agents, which are the drivers of the dynamic. We know that all infected agents will recover after a finite number of time-steps *and* that there is only a finite source for infected agents which is monotonously decreasing.

The idea is to implement a total agent-based SIR simulation, where the termination does NOT depend on time (is not terminated after a finite number of time-steps, which would be trivial). The dynamics of the system-dynamics SIR model are in equilibrium (won't change anymore) when the infected stock is 0. This can (probably) be shown formally but intuitionistic it is clear because only infected agents can lead to infections of susceptible agents which then make the transition to recovered after having gone through the infection phase. Thus an agent-based implementation of the SIR simulation has to terminate if it is implemented correctly because all infected agents will recover after a finite number of steps after then the dynamics will be in equilibrium. Thus we need to 'tell' the type-checker the following: 1) no more infected agents is the termination criterion 2) all infected agents will recover after a finite number of time => the simulation will eventually run out of infected agents But when we look at the SIR+S model we have the same termination criterion, but we cannot guarantee that it will run out of infected => we need additional criteria 3) infected agents are 'generated' by susceptible agents 4) susceptible agents are NOT INCREASING (e.g. recovered agents do NOT turn back into susceptibles) Interesting: can we adopt our solution (if we find it), into a SIRS implementation? this should then break totality. also how difficult is it?

The HOTT book states that lists, trees,... are inductive types/inductively defined structures where each of them is characterized by a corresponding "induction principle". For a proof of totality of SIR we need to find the "induction principle" of the SIR model and implement it. What is the inductive, defining structure of the SIR model? is it a tree where a path through the tree is one simulation dynamics? or is it something else? it seems that such a tree would grow and then shrink again e.g. infected agents. Can we then apply this further to (agent-based) simulation in general?

TODO: https://stackoverflow.com/questions/19642921/assisting-agdas-termination-checker/39591118

7 CONCLUSIONS

Issues

8 FURTHER RESEARCH

ACKNOWLEDGMENTS

The authors would like to thank TODO: for constructive comments and valuable discussions.

REFERENCES

[1] Thorsten Altenkirch, Conor Mcbride, and James Mckinna. 2005. Why dependent types matter. In *http://www.e-pig.org/downloads/ydtm.pdf*.

[2] N. Botta, A. Mandel, C. Ionescu, M. Hofmann, D. Lincke, S. Schupp, and C. Jaeger. 2011. A functional framework for agent-based models of exchange. *Appl. Math. Comput.* 218, 8 (Dec. 2011), 4025–4040. https://doi.org/10.1016/j.amc.2011.08.051

[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. 2013. Programming and Reasoning with Algebraic Effects and Dependent Types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2500365.2500581

[5] Edwin Brady. 2016. *State Machines All The Way Down - An Architecture for Dependently Typed Applications.* Technical Report. https://www.idris-lang.org/drafts/sms.pdf

[6] Edwin Brady. 2017. *Type-Driven Development with Idris.* Manning Publications Company. Google-Books-ID: eWzEjwEACAAJ.

[7] Edwin Brady and Kevin Hammond. 2010. Correct-by-Construction Concurrency: Using Dependent Types to Verify Implementations of Effectful Resource Usage Protocols. *Fundam. Inf.* 102, 2 (April 2010), 145–176. http://dl.acm.org/citation.cfm?id=1883634.1883636

[8] Edwin C. Brady. 2011. Idris âĂŤ systems programming meets full dependent types. In *In Proc. 5th ACM workshop on Programming languages meets program verification, PLPV âĂŹ11.* ACM, 43–54.

[9] Joshua M. Epstein and Robert Axtell. 1996. *Growing Artificial Societies: Social Science from the Bottom Up.* The Brookings Institution, Washington, DC, USA.

[10] Simon Fowler and Edwin Brady. 2014. Dependent Types for Safe and Secure Web Programming. In *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages (IFL '13)*. ACM, New York, NY, USA, 49:49–49:60. https://doi.org/10.1145/2620678.2620683

[11] W. O. Kermack and A. G. McKendrick. 1927. A Contribution to the Mathematical Theory of Epidemics. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 115, 772 (Aug. 1927), 700–721. https://doi.org/10.1098/rspa.1927.0118

[12] Ruth Meyer. 2014. Event-Driven Multi-agent Simulation. In *Multi-Agent-Based Simulation XV (Lecture Notes in Computer Science)*. Springer, Cham, 3–16. https://doi.org/10.1007/978-3-319-14627-0_1

[13] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics.* https://homotopytypetheory.org/book, Institute for Advanced Study.

[14] Neil Sculthorpe and Henrik Nilsson. 2009. Safe Functional Reactive Programming Through Dependent Types. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 23–34. https://doi.org/10.1145/1596550.1596558

[15] David Sorokin. 2015. *Aivika 3: Creating a Simulation Library based on Functional Programming.*

[16] Simon Thompson. 1991. *Type Theory and Functional Programming.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[17] Philip Wadler. 2015. Propositions As Types. *Commun. ACM* 58, 12 (Nov. 2015), 75–84. https://doi.org/10.1145/2699407

[18] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. 2000. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems.* Academic Press. Google-Books-ID: REzmYOQmHuQC.

Received March 2019