The Agent's new Cloths

Towards functional programming in Agent-Based Simulation

[Anonymised review copy]

Abstract: TODO: parallelism for free because all isolated e.g. running multiple replications or parameter-variations

TODO: it is paramount not to write against the established approach but for the functional approach. not to try to come up with arguments AGAINST the object-oriented approach but IN FAVOUR for the functional approach. In the end: dont tell the people that what they do sucks and that i am the saviour with my new method but: that i have a new method which might be of interest as it has a few nice advantages.

So far, the pure functional paradigm hasn't got much attention in Agent-Based Simulation (ABS) where the dominant programming paradigm is object-orientation, with Java, Python and C++ being its most prominent representatives. We claim that functional programming using Haskell is very well suited to implement complex, real-world agent-based models and brings with it a number of benefits. In this paper we will introduce the reader to the functional programming paradigm and explain how it can be applied to implementing ABS. Further we discuss benefits and advantages. As use-case we implemented the seminal Sugarscape model in Haskell.

Keywords: Agent-Based Simulation, Functional Programming, Haskell

Introduction

1.1 The traditional approach to Agent-Based Simulation (ABS) has so far always been object-oriented techniques, due to the influence of the seminal work of Epstein et al Epstein & Axtell (1996) in which the authors claim "[..] object-oriented programming to be a particularly natural development environment for Sugarscape specifically and artificial societies generally [..]" (p. 179). This work established the metaphor in the ABS community, that *agents map naturally to objects* (North & Macal 2007) which still holds up today.

1.2 In this paper we challenge this metaphor and explore ways of approaching ABS using the functional programming paradigm as in the language Haskell. By doing this we expect to leverage the benefits of it (Hudak et al. 2007) to become available when implementing ABS functionally: expressing *what* a system is instead of *how* it works through declarative code, being explicit about the interactions of the program with the real world, explicit data-centric programming resulting in less sources of bugs and a strong static type system making type-errors at run-time obsolete.

1.3 We show that these functional concepts result in an approach to implementing ABS where it is harder to make mistakes and allow to implement simulations which are guaranteed to be reproducible, have less sources of bugs, are easier to verify and thus more likely to be correct which is of paramount importance in high-impact scientific computing.

1.4 As a use-case throughout the paper we employ the well known SugarScape model (Epstein & Axtell 1996) to demonstrate our case-studies, because it can be seen as one of the most influential models in ABS and it laid the foundations of object-oriented implementation of agent-based models.

1.5 The aim of this paper is show *how* to implement ABS in functional programming as in Haskell and *why* it is of benefit of doing so. Further, we give the reader a good understanding of what functional programming is, what the challenges are in applying it to ABS and how we solve these in our approach.

1.6 The paper makes the following contributions:

- It is the first to *systematically* introduce the functional programming paradigm, as in Haskell, to ABS, identifying its benefits, difficulties and drawbacks.

- We show how functional ABS can be scaled up to massively large-scale without the problems of low level concurrent programming using Software Transactional Memory (STM). Although there exist STM implementations in non-functional languages like Java and Python, due to the nature of Haskells type-system, the use of STM has unique benefits in this setting.

- Further we introduce a powerful and very expressive approach to testing ABS implementations using property-based testing. This allows a much more powerful way of expressing tests, shifting from unit-testing towards specification-based testing. Although property-based testing has been brought to non-functional languages like Java and Python as well, it has its origins in Haskell and it is here where it truly shines.

Concepts of Functional Programming

2.1 In our research we are using the functional programming language Haskell. The paper of (Hudak et al. 2007) gives a comprehensive overview over the history of the language, how it developed and its features and is very interesting to read and get accustomed to the background of the language. A widely used introduction to programming in Haskell is (Hutton 2016), a more conceptual introduction to functional programming can be found in (MacLennan 1990). The main points why we decided to go for Haskell are:

- Rich Feature-Set - it has all fundamental concepts of the pure functional programming paradigm of which we explain the most important below.

- Real-World applications - the strength of Haskell has been proven through a vast amount of highly diverse real-world applications [1] Hudak et al. (2007), is applicable to a number of real-world problems O'Sullivan et al. (2008) and has a large number of libraries available.

- Modern - Haskell is constantly evolving through its community and adapting to keep up with the fast changing field of computer science. Further, the community is the main source of high-quality libraries.

2.2 The roots of functional programming lie in the Lambda Calculus which was first described by Alonzo Church (Church 1936). This is a fundamentally different approach to computation than imperative and object-oriented programming which roots lie in the Turing Machine (Turing 1937). Rather than describing *how* something is computed as in the more operational approach of the Turing Machine, due to the more declarative nature of the Lambda Calculus, code in functional programming describes *what* is computed.

2.3 As a motivating example we give an implementation of the factorial function in Haskell: factorial :: Integer -> Integer factorial 0 = 1 factorial n = n * factorial (n-1)

2.4 When looking at this function we can already see a few things:

1. Declarative - we describe *what* the factorial function is rather than how to compute it. This is supported by *pattern matching* which allows to give multiple equations for the same function, matching on its input.

2. Immutable data - in functional programming we don't have mutable variables - after a variable is assigned, it cannot change its contents. This also means that there is no destructive assignment operator which can re-assign values to a variable. To change values, we employ recursion.

3. Recursion - the function calls itself with a smaller argument and will eventually reach the case of 0. Recursion is the very meat of functional programming because they are the only way to implement loops in this paradigm due to immutable data.

4. Static Types - the first line indicates the name and the types of the function. In this case the function takes one Integer as input and returns an Integer as output. Types are static in Haskell which means that there can be no type-errors at run-time e.g. when one tries to cast one type into another because this is not supported by this kind of type-system.

5. Explicit input and output - all data which are required and produced by the function have to be explicitly passed in and out of it. There exists no global mutable data whatsoever and data-flow is always explicit.

[1] `https://wiki.haskell.org/Applications_and_libraries`

6. Purity - this function is pure, indicated by its type, which means that it can have no side-effects whatsoever. Calling this function with the same argument will *always* lead to the same result - this concept is called *referential transparency*. This means that when implementing this function one can not read from a file or open a connection to a server. In Haskell this is indicated in the types and ensured by the compiler.

2.5 It may seem that one runs into efficiency-problems in a pure functional programming language when using algorithms which are implemented in imperative languages through mutable data which allows in-place update of memory. The seminal work of (Okasaki 1999) showed that when approaching this problem from a functional mind-set this does not necessarily be the case. The author presents functional data structures which are asymptotically as efficient as the best imperative implementations and discusses the estimation of the complexity of lazy programs.

Side-Effects

Functional Reactive Programming

Related Research

2.6 TODO: paper by James Odell "Objects and Agents Compared"

A functional approach

3.1 Due to the fundamentally different approaches of pure Functional Programming (pure FP) an ABS needs to be implemented fundamentally different as well compared to traditional object-oriented approaches (OO). We face the following challenges:

1. How can we represent an Agent?
   In OO the obvious approach is to map an agent directly onto an object which encapsulates data and provides methods which implement the agents actions. Obviously we don't have objects in pure FP thus we need to find a different approach to represent the agents actions and to encapsulate its state.

2. How can we represent state in an Agent?
   In the classic OO approach one represents the state of an Agent explicitly in mutable member variables of the object which implements the Agent. As already mentioned we don't have objects in pure FP and state is immutable which leaves us with the very tricky question how to represent state of an Agent which can be actually updated.

3. How can we implement proactivity of an Agent?
   In the classic OO approach one would either expose the current time-delta in a mutable variable and implement time-dependent functions or ignore it at all and assume agents act on every step. At first this seems to be not a big deal in pure FP but when considering that it is yet unclear how to represent Agents and their state, which is directly related to time-dependent and reactive behaviour it raises the question how we can implement time-varying and reactive behaviour in a purely functional way.

4. How can we implement the agent-agent interaction?
   In the classic OO approach Agents can directly invoke other Agents methods which makes direct Agent interaction *very* easy. Again this is obviously not possible in pure FP as we don't have objects with methods and mutable state inside.

5. How can we represent an environment and its various types?
   In the classic OO approach an environment is almost always a mutable object which can be easily made dynamic by implementing a method which changes its state and then calling it every step as well. In pure FP we struggle with this for the same reasons we face when deciding how to represent an Agent, its state and proactivity.

Doi: 10.18564/jasss.xxxx

6. How can we implement the agent-environment interaction?
   In the classic OO approach agents simply have access to the environment either through global mechanisms (e.g. Singleton or simply global variable) or passed as parameter to a method and call methods which change the environment. Again we don't have this in pure FP as we don't have objects and globally mutable state.

7. How can we step the simulation?
   In the classic OO approach agents are run one after another (with being optionally shuffled before to uniformly distribute the ordering) which ensures mutual exclusive access in the agent-agent and agent-environment interactions. Obviously in pure FP we cannot iteratively mutate a global state.

Agent representation, state and proactivity

3.2 Whereas in imperative programming (the OO which we refer to in this paper is built on the imperative paradigm) the fundamental building block is the destructive assignment, in FP the building blocks are obviously functions which can be evaluated. Thus we have no other choice than to represent our Agents using a function which implements their behaviour. This function must be time-aware somehow and allow us to react to time-changes and inputs. Fortunately there exists already an approach to time-aware, reactive programming which is termed Functional Reactive Programming (FRP). This paradigm has evolved over the year and current modern FRP is built around the concept of a signal-function which transforms an input-signal into an output-signal. An input-signal can be seen as a time-varying value. Signal-functions are implemented as continuations which allows to capture local state using closures. Modern FRP also provides feedback functions which provides convenient methods to capture and update local state from the previous time-step with an initial state provided at time = 0.

3.3 - pure functions don't have a notion of communication as opposed to method calls in object-oriented languages like java

3.4 - time is represented using the FRP concept: Signal-Functions which are sampled at (fixed) time-deltas, the dt is never visible directly but only reflected in the code and read-only. - no method calls => continuous data-flow instead

3.5 Viewing agent-agent interaction as simple method calls implies the following: - it takes no time - it has a synchronous and transactional character - an agent gives up control over its data / actions or at least there is always the danger that it exposes too much of its interface and implementation details. - agents equals objects, which is definitely NOT true. Agents

3.6 data-flow synchronous agent transactions

3.7 - still need transactions between two agents e.g. trading occurs over multiple steps (makeoffer, accept/refuse, finalize/abort) -> exactly define what TX means in ABS -> exclusive between 2 agents -> state-changes which occur over multiple steps and are only visible to the other agents after the TX has commited -> no read/write access to this state is allowed to other agents while the TX is active -> a TX executes in a single time-step and can have an arbitrary number of tx-steps -> it is easily possible using method-calls in OOP but in our pure functional approach it is not possible -> parallel execution is being a problem here as TX between agents are very easy with sequential -> an agent must be able to transact with as many other agents as it wants to in the same time-step -> no time passes between transactions => what we need is a 'all agents transact at the same time' -> basically we can implement it by running the SFs of the agents involved in the TX repeatedly with dt=0 until there are no more active TXs -> continuations (SFs) are perfectly suited for this as we can 'rollback' easily by using the SF before the TX has started

Environment representation and interaction

3.8 no global shared mutable environment, having different options: - non-active read-only (SIR): no agent, as additional argument to each agent - pro-active read-only (?): environment as agent, broadcast environment updates as data-flow - non-active read/write (?): no agent, StateT in agents monad stack - pro-active read/write (Sugarscape): environment as agent, StateT in agents monad stack

3.9 care must be taken in case of agent-transactions: when aborting/refusing all changes to the environment must be rolled back => instead of StateT use a transactional monad which allows us to revert changes to a save point at the start of the TX. if we drag the environment through all agents then we could easily revert changes but that then requires to hard-code the environment concept deep into the simulation scheduling/stepping which brings lots of inconveniences, also it would need us to fold the resulting multiple environments back into a single. If we had an environment-centric view then probably this is what we want but in ABS the focus is on the agents

3.10 question is if the TX sf runs in the same monad aw the agent or not. i opt for identity monad which prevents modification of the Environment in a transaction

3.11 also need to motivate the dt=0 in all TX processing: conceptually it all happens instantaneously (although arbitration is sequential) but agents must act time-sensitive

3.12 for environment we need transactional and shared state behaviour where we can have mutual exclusive access to shared data but also roll back changes we made. it should run deterministic when running agents not truly parallel. solution: run environment in a transactional state monad (TX monad). although the agents are executed in parallel in the end it (map) runs sequentially. this passes a mutable state through all agents which can act on it an roll back actions e.g. in case of a failed agent TX. if we dont need transactional behaviour then just use StateT monad. this ensures determinism. pro active environment is also easily possible by writing to the state. this approach behaves like sequential transactional although the agents run in parallel but how is this possible when using mapMSF ?

Stepping the simulation

3.13 - parallel update only, sequential is deliberately abandoned due to: -> reality does not behave this way -> if we need transactional behaviour, can use STM which is more explicit -> it is translates directly to a map which is very easy to reason about (sequential is basically a fold which is much more difficult to reason about) -> is more natural in functional programming -> it exists for 'transactional' reasons where we need mutual exclusive access to environment / other agents -> we provide a more explicit mechanism for this: Agent Transactions

References

Church, A. (1936). An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, *58*(2), 345–363. doi:10.2307/2371045
URL `http://dx.doi.org/10.2307/2371045`

Epstein, J. M. & Axtell, R. (1996). *Growing Artificial Societies: Social Science from the Bottom Up*. Washington, DC, USA: The Brookings Institution

Hudak, P., Hughes, J., Peyton Jones, S. & Wadler, P. (2007). A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, (pp. 12–1–12–55). New York, NY, USA: ACM. doi:10.1145/1238844.1238856
URL `http://dx.doi.org/10.1145/1238844.1238856`

Hutton, G. (2016). *Programming in Haskell*. Cambridge University Press. Google-Books-ID: 1xHPDAAAQBAJ

MacLennan, B. J. (1990). *Functional Programming: Practice and Theory*. Addison-Wesley. Google-Books-ID: JqhQAAAAMAAJ

North, M. J. & Macal, C. M. (2007). *Managing Business Complexity: Discovering Strategic Solutions with Agent-Based Modeling and Simulation*. Oxford University Press, USA. Google-Books-ID: gRATDAAAQBAJ

Okasaki, C. (1999). *Purely Functional Data Structures*. New York, NY, USA: Cambridge University Press

O'Sullivan, B., Goerzen, J. & Stewart, D. (2008). *Real World Haskell*. O'Reilly Media, Inc., 1st edn.

Turing, A. M. (1937). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, *s2-42*(1), 230–265. doi:10.1112/plms/s2-42.1.230
URL `http://dx.doi.org/10.1112/plms/s2-42.1.230`