Programming Paradigms in Agent-Based Simulation

Jonathan Thaler

February 20, 2017

Abstract

three very different programming languages Java, Haskell and Scala and compare their suitability to implement the strategies. The major finding is that depending on the kind of game one can expect very different results when selecting different update-strategies and that the language for implementing the ABS should be considered as well, reflecting its suitability to implement the update-strategy.

Keywords

Agent-Based Simulation, Parallelism, Concurrency, Haskell, Actors, Prisoners Dilemma, Heroes and Cowards

1 Introduction

Because the selection of an update-strategy has profound implications for the implementation of an ABS we investigate the three different programming-paradigms of *object-orientation* (OO), *pure functional* and *multi-paradigm* in the form of the programming languages Java, Haskell and Scala in their suitability of implementing each update-strategy. As it turns out the paradigms can't capture all the update-strategies equally well thus one should be careful when selecting the implementation language for the ABS, reflecting its suitability for implementing the selected updates-strategy.

2 Background

2.1 Related Research

The amount of research on using the pure functional paradigm using Haskell in the field of ABS has been moderate so far. Though there exist a few papers which look into Haskell and ABS [2], [9], [6] they focus primarily on how to specify agents. A library for Discrete Event Simulation (DES) and System Dynamics (SD) in Haskell called *Aivika 3* is described in [8]. It also comes with very basic features for ABS but only allows to specify simple state-based agents with timed transitions. This papers is investigating Haskell in a different way by looking into its suitability in implementing update-strategies in ABS, something not looked at in the ABS community so far, presenting an original novelty.
There already exists research using the Actor Model [1] for ABS in the context of Erlang [10], [3], [4], [7] but we feel that they barely scratched the surface. We want to renew the interest in this direction of research by incorporating Scala with using the Actor-library in our research because we will show that one update-strategy maps directly to the Actor Model.

3 Programming paradigms and ABS

In this section we give a brief overview of comparing the suitability of three fundamentally different languages to implement the different update-strategies. We wanted to cover a wide range of different types of languages and putting emphasis on each languages strengths without abusing language constructs to

recreate features it might seem to lack. An example would be to rebuild OO constructs in pure functional languages which would be a abuse of the language, something we explicitly avoided although it resulted in a few limitations as noted below. We implemented both the *Prisoners Dilemma* game on a 2D grid and the *Heroes & Cowards* game in all three languages with all four update-strategies. [1] TODO: reference table:

3.1 OO: Java

This language is included as the benchmark of object-oriented (OO) imperative languages as it is extremely popular in the ABS community and widely used in implementing their models and frameworks. It comes with a comprehensive programming library and powerful synchronization primitives built in at language-level.

Ease of Use We found that implementing all the strategies was straight-forward and easy thanks to the languages features. Especially parallelism and concurrency is convenient to implement due to elegant and powerful built-in synchronization primitives.

Benefits We experienced quite high-performance even for a large number of agents which we attributed to the implicit side-effects using aliasing through references. This prevents massive copying like Haskell but comes at the cost of explicit data-flow.

Deficits A downside is that one must take care when accessing memory in case of *parallel* or *concurrent strategy*. Due to the availability of aliasing and side-effects in the language it can't be guaranteed by Java's type-system that access to memory happens only when its safe. So care must be taken when accessing references sent by messages to other agents, accessing references to other agents or the infrastructure of an agent itself e.g. the message-box.

[1] Code available under `https://github.com/thalerjonathan/phd/tree/master/coding/papers/iteratingABM/`

We found that implementing the *actor strategy* was not possible when using thousands of agents because Java can't handle this number of threads. For implementing the *parallel* and *concurrent* ones we utilized the ExecutorService to submit a task for each agent which runs the update and finishes then. The tasks are evenly distributed between the available threads using this service where the service is backed by the number of cores the CPU has. This approach does not work for the *actor strategy* because there an agent runs constantly within its thread making it not possible to map to the concept of a task as this task would not terminate. The ExecutorService would then start n tasks (where n is the number of threads in the pool) and would not start new ones until those have finished, which will not occur until the agent would shut itself down. Also yielding or sleeping does not help either as not all threads are started but only n.

Natural Strategy We found that the *sequential strategy* with immediate message-handling is the most natural strategy to express in Java due to its heavy reliance on side-effects through references (aliases) and shared thread of execution. Also most of the models work this way making Java a save choice for implementing ABS.

3.2 Pure functional: Haskell

This language is included to put to test whether a pure functional, declarative programming language is suitable for full-blown ABS. What distinguishes it is its complete lack of implicit side-effects, global data, mutable variables and objects. The central concept is the function into which all data has to be passed in and out explicitly through statically typed arguments and return values: data-flow is completely explicit. For a nice overview on the features and strengths of pure functional programming see the classical paper [5].

Ease of Use We initially thought that it would be suitable best for implementing the *parallel strategy* only due the inherent data-parallel nature of pure functional languages. After having implementing all

strategies we had to admit that Haskell is very well suited to implement all of them faithfully. We think this stems from the fact that it has no implicit side-effects which reduces bugs considerably and results in completely explicit data-flow. Not having objects with data and methods, which can call between each other meant that we needed some different way of representing agents. This was done using a struct-like type to carry data and a transformer function which would receive and process messages. This may seem to look like OO but it is not: agents are not carried around but messages are sent to a receiver identified by an id.

Benefits Haskell has a very powerful static type-system which seems to be restrictive in the beginning but when one gets used to it and knows how to use it for ones support, then it becomes rewarding. Our major point was to let the type-system prevent us from introducing side-effects. In Haskell this is only possible in code marked in its types as producing side-effects, so this was something we explicitly avoided and were able to do so throughout the whole implementation. This means a user of this approach can be guided by the types and is prevented from abusing them. In essence, the lesson learned here is *if one tries to abuse the types or work around, then this is an indication that the update-strategy one has selected does not match the semantics of the model one wants to implement.* If this happens in Java, it is much more easier to work around by introducing global variables or side-effects but this is not possible in Haskell. Also we claim that when using Haskell one arrives at a much safer version in the case of Parallel or Concurrent Strategies than in Java.

Parallelism and Concurrency is extremely convenient to implement in Haskell due to its complete lack of implicit side-effects. Adding hardware-parallel execution in the *parallel strategy* required the adoption of only 5 lines of code and no change to the existing agent-code at all (e.g. no synchronization, as there are no implicit side-effects). For implementing the *concurrent strategy* we utilized the programming model of Software-Transactional-Memory (STM). The approach is that one optimistically runs agents which introduce explicit side-effects in parallel where each agent executes in a transaction and then to simply retry the transaction if another agent has made concurrent side-effect modifications. This frees one from thinking in terms of synchronization and leaves the code of the agent nearly the same as in the *sequential strategy*. Spawning thousands of threads in the *actor strategy* is no problem in Haskell due to its lightweight handling of threads internal in the runt-time system, something which Java seems to be lacking. We have to note that each agents needs to explicitly yield the execution to allow other agent-threads to be scheduled, something when omitted will bring the system to a grind.

Deficits Performance is an issue. Our Haskell solution could run only about 2000 agents in real-time with 25 updates per second as opposed to 50.000 in our Java solution, which is not very fast. It is important though to note, that being beginners in Haskell, we are largely unaware of the subtle performance-details of the language so we expect to achieve a massive speed-up in the hands of an experienced programmer.

Another thing is that currently only homogeneous agents are possible and still much work needs do be done to capture large and complex models with heterogeneous agents. For this we need a more robust and comprehensive surrounding framework, which is already existent in the form of functional reactive programming (FRP). Our next paper is targeted on combining our Haskell solution with an FRP framework like Yampa (see Further Research).

Our solution so far is unable to implement the *sequential strategy* with immediate message-handling. This is where OO really shines and pure functional programming seems to be lacking in convenience. A solution would need to drag the collection of all agents around which would make state-handling and manipulation very cumbersome. In the end it would have meant to rebuild OO concepts in a pure functional language, something we didn't wanted to do. For now this is left as an open, unsolved issue and we hope that it could be solved in our approach with FRP (see future research).

Natural Strategy The most natural strategy is the *parallel strategy* as it lends itself so well to the concepts of pure functional programming where things are evaluated virtually in parallel without side-effects on each other - something which resembles exactly the semantics of the *parallel strategy*. We argue that with slightly more effort, the *concurrent strategy* is also very natural formulated in Haskell due to the availability of STM, something only possible in a language without implicit side-effects as otherwise retries of transactions would not be possible.

3.3 Multi-paradigm: Scala

This multi-paradigm functional language which sits in-between Java and Haskell and is included to test the usefulness of the *actor strategy* for implementing ABS. The language comes with an Actor-library inspired by [1] and resembles the approach of Erlang which allows a very natural implementation of the strategy.

Ease of Use We were completely new to Scala with Actors although we have some experience using Erlang which was of great use. We found that the language has some very powerful mixed-paradigm features which allow to program in a very flexible way without inducing too much restrictions on one.

Benefits Implementing agent-behaviour is extremely convenient, especially for simple state-driven agents. The Actor-language has a built-in feature which allows to change the behaviour of an agent on message-reception where the agent then simply switches to a different message-handler, allowing elegant implementation of state-dependent behaviour.
Performance is very high. We could run simulations in real-time with about 200.000 agents concurrently, thanks to the transparent handling in the run-time system. Also it is very important to note that one can use the framework Akka to build real distributed systems using Scala with Actors so there are potentially no limits to the size and complexity of the models and number of agents one wants to run with it.

Deficits Care must be taken not to send references and mutable data, which is still possible in this mixed-paradigm language.

Natural Strategy The most natural strategy would be of course the *actor strategy* and we only used this strategy in this language to implement our models. Note that the *actor strategy* is the most general one and would allow to capture all the other strategies using the appropriate synchronization mechanisms.

4 Conclusion and future research

In this paper we presented general properties of ABS, derived four general update-strategies and discussed their implications. Again we cannot stress enough that selecting the right update-strategy is of most importance and must match the semantics of the model one wants to simulate. We also argued that the ABS community needs a unified terminology of speaking about update-strategies otherwise confusions arise and reproducibility suffers. We proposed such a unified terminology on the basis of the general update-strategies and hope it will get adopted. We put our theoretical considerations to a practical test by implementing case-studies using three very different kind of languages to see how each of them performed in comparison with each other in implementing the update-strategies. To summarize, we can say that Java is the gold-standard due to convenient synchronization primitives built in the language. Haskell really surprised us as it allowed us to faithfully implement all strategies equally well, something we didn't anticipate in the beginning of our research. We hope that our work convinces researches and developers in the field of ABS to give Haskell a try and dig deeper into it, as we feel it will be highly rewarding. We explicitly avoided using the functional reactive programming (FRP) paradigm to keep our solution simple but could only build simple models with homogeneous agents. The next step would be to fusion ABS with FRP using the library Yampa

Table 1: Language Comparisons

	Java	Haskell	Scala
Sequential	++	–	? (+)
Parallel	+	++	? (+)
Concurrent	+	+	? (+)
Actor	–	+	++

for leveraging both approaches from which we hope to gain the ability to develop much more complex models with heterogeneous agents. If one can live with the non-determinism of Scala with the Actors-library it is probably the most interesting and elegant solution to implement ABS. We attribute this to the closeness of Actors to the concept of agents, the powerful concurrency abstraction and language-level support. We barely scratched the surface on this topic but we find that the Actor Model should get more attention in ABS. We think that this research-field is nowhere near exhaustion and we hope that more research is going into this topic as we assume that the Actor-Model has a bright future ahead due to the ever increasing availability of massively parallel computing machinery. We showed that the the *Prisoners Dilemma* game on a 2D-grid can only be simulated correctly when using the *parallel strategy* and that the other strategies lead to a break-down of the emergent pattern reported in the original paper. On the other hand using the *Heroes & Cowards* game we showed that there exist models whose emergent patterns exhibit a stability under varying update-strategies. Intuitively we can say that this is due to the nature of the model specification which does not require specific orderings of actions but it would be interesting to put such intuitions on firm theoretical grounds.

References

[1] Agha, G. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.

[2] De Jong, T. Suitability of Haskell for Multi-Agent Systems. Tech. rep., University of Twente, 2014.

[3] Di Stefano, A., and Santoro, C. Using the Erlang Language for Multi-Agent Systems Implementation. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology* (Washington, DC, USA, 2005), IAT '05, IEEE Computer Society, pp. 679–685.

[4] Di Stefano, A., and Santoro, C. eXAT: an Experimental Tool for Programming Multi-Agent Systems in Erlang. Tech. rep., 2007.

[5] Hughes, J. Why Functional Programming Matters. *Comput. J. 32*, 2 (Apr. 1989), 98–107.

[6] Jankovic, P., and Such, O. Functional Programming and Discrete Simulation. Tech. rep., 2007.

[7] Sher, G. I. *Agent-Based Modeling Using Erlang Eliminating The Conceptual Gap Between The Programming Language & ABM.* 2013.

[8] Sorokin, D. Aivika 3: Creating a Simulation Library based on Functional Programming, 2015.

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

[10] Varela, C., Abalde, C., Castro, L., and Gulías, J. On Modelling Agent Systems with Erlang. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Erlang* (New York, NY, USA, 2004), ERLANG '04, ACM, pp. 65–70.