THE AGENTS NEW CLOTHS?
TOWARDS PURE FUNCTIONAL PROGRAMMING IN ABS

Jonathan Thaler
Peer Olaf Siebers

School Of Computer Science
University of Nottingham
Nottingham, United Kingdom
{jonathan.thaler,peer-olaf.siebers}@nottingham.ac.uk

ABSTRACT

The established approach to implement and engineer Agent-Based Simulations (ABS) has primarily been object-oriented with Python and Java being the most popular languages. In this paper we explore a different approach to this problem and investigate the pure functional programming paradigm, using the language Haskell. We give a high level introduction into the core features of pure functional programming and show how they can be made of use to implement ABS in a case-study of a full implementation of the seminal Sugarscape model. With this case-study we are able to show that pure functional programming as in Haskell has a valid place in building clean, robust and maintainable ABS implementations. Further we show that we can directly leverage the benefits of pure functional programming to ABS: we have strong guarantees of reproducibility already at compile time, can easily exploit data-parallelism and concurrency is easier to get right. The main drawback is a lower performance than established performances but this was not the main focus of research and we hope to improve on this in the future.

Keywords: Agent-Based Simulation, Functional Programming.

1 INTRODUCTION

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 and Axtell 1996) in which the authors claim *"[..] object-oriented programming (OOP) 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 and Macal 2007) which still holds up today.

Although we agree that OOP is a good fit for ABS, in this paper we present ways of implementing ABS in the functional programming paradigm using the language Haskell (Hudak, Hughes, Peyton Jones, and Wadler 2007). We believe that functional programming has its place in ABS as well because of ABS' *scientific computing* nature where results need to be reproducible and correct up to some specification, while simulations should be able to exploit parallelism and concurrency as well. We claim that by using functional programming for implementing ABS it is less difficult to add parallelism and correct concurrency,

the resulting simulations are easier to test and verify, guaranteed to be reproducible already at compile-time, have fewer potential sources of bugs and are ultimately more likely to be correct.

To substantiate our claims, we first present fundamental concepts and advanced features of functional programming and then show how they can be used to engineer clean, maintainable and reusable ABS implementations. Further we discuss how the well known benefits of functional programming in general are applicable in ABS. We discuss this in the context of a practical case-study we conducted, in which we implemented the *full* SugarScape model (Epstein and Axtell 1996) in Haskell.

The aim and contribution of this paper is to introduce the functional programming paradigm using Haskell to ABS on a *conceptual* level, identifying benefits, difficulties and drawbacks. This is done through the above mentioned case-study which introduces general implementation techniques applicable to ABS and investigates benefits and drawbacks leveraged from functional programming. To the best of our knowledge, we are the first to do so within the ABS community.

The structure of the paper is as follows. In Section 2 we present related work on functional programming in ABS. In Section 3 we introduce the functional programming paradigm, motivate why we chose Haskell, establish key features and briefly discuss its type system and side-effects. In Section 4 we present our approach in the context of our Sugarscape case-study together with the challenges encountered and its benefits and drawbacks. In Section 5 we discuss our initial claims in the light of the case-study. In Section 6 we conclude and point out further research.

2 RELATED WORK

The amount of research on using FP with Haskell in the field of ABS has been moderate so far. Most of the papers are related to the field of Multi Agent Systems and look into how agents can be specified using the belief-desire-intention paradigm (De Jong 2014, Sulzmann and Lam 2007, Jankovic and Such 2007).

A multi-method simulation library in Haskell called *Aivika 3* is described in the technical report (Sorokin 2015). It supports implementing Discrete Event Simulations (DES), System Dynamics and comes with basic features for event-driven ABS which is realised using DES under the hood. Further it provides functionality for adding GPSS to models and supports parallel and distributed simulations. It runs within the IO effect type for realising parallel and distributed simulation but also discusses generalising their approach to avoid running in IO.

In his masterthesis (Bezirgiannis 2013) the author investigated Haskells parallel and concurrency features to implement (amongst others) *HLogo*, a Haskell clone of the NetLogo simulation package, focusing on using Software Transactional Memory for a limited form of agent-interactions. *HLogo* is basically a re-implementation of NetLogos API in Haskell where agents run within the IO effect type.

Using functional programming for DES was discussed in (Jankovic and Such 2007) where the authors explicitly mention the paradigm of Functional Reactive Programming (FRP) to be very suitable to DES.

A domain-specific language for developing functional reactive agent-based simulations was presented in (Schneider, Dutchyn, and Osgood 2012, Vendrov, Dutchyn, and Osgood 2014). This language called FRABJOUS is human readable and easily understandable by domain-experts. It is not directly implemented in FRP/Haskell but is compiled to Haskell code which they claim is also readable. This supports that FRP is a suitable approach to implement ABS in Haskell. Unfortunately, the authors do not discuss their mapping of ABS to FRP on a technical level, which would be of most interest to functional programmers.

The authors of (Botta, Mandel, and Ionescu 2010) discuss the problem of advancing time in message-driven agent-based socio-economic models. They formulate purely functional definitions for agents and

their interactions through messages. Our architecture for synchronous agent-interaction as discussed in the case-study was not directly inspired by their work but has some similarities: the use of messages and the problem of when to advance time in models with arbitrary number synchronised agent-interactions.

3 FUNCTIONAL PROGRAMMING

Functional programming (FP) is called *functional* because it makes functions the main concept of programming, promoting them to first-class citizens: functions can be assigned to variables, they can be passed as arguments to other functions and they can be returned as values from functions. The roots of FP lie in the Lambda Calculus which was first described by Alonzo Church (Church 1936). This is a fundamentally different approach to computing than imperative programming (which includes established OOP) which roots lie in the Turing Machine (Turing 1937). Rather than describing *how* something is computed as in the operational approach of the Turing Machine, due to the more *declarative* nature of the Lambda Calculus, code in FP describes *what* is computed.

In our research we are using the *pure* FP language Haskell. The paper of (Hudak, Hughes, Peyton Jones, and Wadler 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. The main points why we decided to go for Haskell are:

- Rich Feature-Set - it has all fundamental concepts of the pure FP paradigm included, of which we explain the most important ones below. Further, Haskell has influenced a large number of languages, underlining its importance and influence in programming language design.
- Real-World applications - the strength of Haskell has been proven through a vast amount of highly diverse real-world applications, is applicable to a number of real-world problems (O'Sullivan, Goerzen, and Stewart 2008) and has a large number of libraries available [1].
- 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.

3.1 Fundamentals

To explain the central concepts of FP, we give an implementation of the factorial function in Haskell:

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

When looking at this function we can identify the following:

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 FP 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.

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

3. Recursion - the function calls itself with a smaller argument and will eventually reach the base-case of 0. Recursion is the very meat of FP because it is the only way to implement loops in this paradigm due to immutable data.
4. Static Types - the first line indicates the name and the type 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.
6. Referential transparency - calling this function with the same argument will *always* lead to the same result, meaning one can replace this function by its value. This means that when implementing this function one can not read from a file or open a connection to a server. This is also known as *purity* and is indicated in Haskell in the types which means that it is also guaranteed by the compiler.

It may seem that one runs into efficiency-problems in Haskell 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 with 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.

For an excellent and widely used introduction to programming in Haskell we refer to (Hutton 2016). Other, more exhaustive books on learning Haskell are (Lipovaca 2011, Allen and Moronuki 2016). For an introduction to programming with the Lambda-Calculus we refer to (Michaelson 2011). For more general discussion of FP we refer to (Hughes 1989, MacLennan 1990).

3.2 Side-Effects

One of the fundamental strengths of Haskell is its way of dealing with side-effects in functions. A function with side-effects has observable interactions with some state outside of its explicit scope. This means that its behaviour depends on history and that it loses its referential transparency character, which makes understanding and debugging much harder. Examples for side-effects are (amongst others): modifying state, await an input from the keyboard, read or write to a file, open a connection to a server, drawing random-numbers,...

Obviously, to write real-world programs which interact with the outside world we need side-effects. Haskell allows to indicate in the *type* of a function that it does or does not have side-effects. Further there are a broad range of different effect types available, to restrict the possible effects a function can have to only required types. The compiler then ensures that these constraints are not violated e.g. a program which tries to read a file in a function which only allows drawing random-numbers will fail to compile. Haskell also provides mechanisms to combine multiple effects e.g. one can define a function which can draw random-numbers *and* modify some state. The most common side-effect types are: *IO* allows all kind of I/O related side-effects: reading/writing a file, creating threads, write to the standard output, read from the keyboard, opening network-connections, mutable references; *Rand* allows drawing random-numbers; *Reader / Writer / State* allows to read / write / both from / to an environment context.

A function without any side-effect type is called *pure*, and the *factorial* function is indeed pure. Below we give an example of a function which is not pure. The *queryUser* function constructs and returns a function with the *IO* side-effect which, when executed, asks the user for its user-name and compares it with a given

user-configuration. In case the user-name matches, it returns True, and False otherwise after printing a corresponding message.

```haskell
queryUser :: String -> IO Bool
queryUser username = do
  -- print text to console
  putStr "Type in user-name: "
  -- wait for user-input
  str <- getLine
  -- check if input matches user-name
  if str == username
    then do
      putStrLn "Welcome!"
      return True
    else do
      putStrLn "Wrong user-name!"
      return False
```

The *IO* in the first line indicates that the function runs in the IO effect and can thus (amongst others) print to the console and read input from it. What is striking is that this looks very much like imperative code - this is no accident and intended. When we are dealing with side-effects, ordering becomes important, thus Haskell introduced the so-called *do-notation* which emulates an imperative style of programming. Whereas in imperative programming languages like C, commands are chained or composed together using the semicolon (;) operator, in FP this is done using function composition: feeding the output of a function directly into the next function. The machinery behind the do-notation does exactly this and desugars this imperative-style code into function compositions which run custom code between each line, depending on the type of effect the computation runs in. This approach of function composition with custom code in between each function allows to emulate a broad range of imperative-style effects, including the above mentioned ones. For a technical, in-depth discussion of the concept of side-effects and how they are implemented in Haskell using Monads, we refer to the papers (Moggi 1989, Wadler 1992, Wadler 1995, Wadler 1997, Jones 2002).

Although it might seem very restrictive at first, we get a number of benefits from making the type of effects we can use in the function explicit. First we can restrict the side-effects a function can have to a very specific type which is guaranteed at compile time. This means we can have much stronger guarantees about our program and the absence of potential errors already at compile-time which implies that we don't need test them with e.g. unit-tests. Second, to actually execute effectful functions, *effect runners* are used, which are themselves *pure* functions [2], taking arguments to the respective effect-context e.g. the mutable initial state. This means that we can execute effectful functions in a very controlled way by making the effect-context explicit in the parameters to the effect execution. This allows a much easier approach to isolated testing because the history of the system is made explicit and has to be provided during execution.

Further, this type system allows Haskell to make a very clear distinction between parallelism and concurrency. Parallelism is always deterministic and thus pure without side-effects because although parallel code runs concurrently, it does by definition not interact with data of other threads. This can be indicated through types: we can run pure functions in parallel because for them it doesn't matter in which order they are executed, the result will always be the same due to the concept of referential transparency. Concurrency is potentially non-deterministic because of non-deterministic interactions of concurrently running threads through shared data. For a technical, in-depth discussion on parallelism and concurrency in Haskell we refer

[2] Except in the case of the *IO* effect, this can only be executed from within the *main* program entry-point which is itself a function within the *IO* effect context.

to the following books and papers: (Marlow 2013, O'Sullivan, Goerzen, and Stewart 2008, Harris, Marlow, Peyton-Jones, and Herlihy 2005, Marlow, Peyton Jones, and Singh 2009).

4 CASE-STUDY: PURE FUNCTIONAL SUGARSCAPE

To explore how to approach ABS based on pure FP concepts as introduced before, we did a full implementation [3] of the seminal Sugarscape model (Epstein and Axtell 1996). We chose this model because it is quite well known in the ABS community, it was highly influential in sparking the interest in ABS, it is quite complex with non-trivial agent-interactions and it used object-oriented techniques and explicitly advocates them as a good fit to ABS, which begged the question whether and how well we could do an FP implementation of it.

Our goal was first to develop techniques and concepts to show *how* to engineer a clean, maintainable and robust ABS in Haskell. The second step was then to discuss *why* and why not one would follow such an approach by identifying benefits and drawbacks. Absolutely paramount in our research was being *pure*, avoiding the *IO* effect type under all circumstances. More specifically, in this case-study:

- We developed techniques for engineering a clean, maintainable, general and reusable *sequential* implementation in Haskell. Those techniques are directly applicable to other ABS implementations as well and are discussed in this paper.
- We explored techniques for a *concurrent* implementation using Software Transactional Memory in Haskell (Harris, Marlow, Peyton-Jones, and Herlihy 2005) and data-parallelism to speed up the execution in our sequential implementation. Due to lack of space, we refer to (TODO: cite our own TOMACS paper) for an in-depth discussion of this work.
- We explored ways of fully validating our implementation using code-testing. We showed how to apply property-based testing from Haskell (Claessen and Hughes 2000) to ABS for a specification based checking of the correctness of the simulation. Due to lack of space, we refer to (TODO: cite our other submission to this conference) for a more detailed discussion of this work.

4.1 A Functional View

Due to the fundamentally different approaches of FP, an ABS needs to be implemented fundamentally different, compared to established OOP approaches. We face the following challenges:

1. How can we represent an Agent, its local state and its interface?
2. How can we implement direct agent-to-agent interactions?
3. How can we implement an environment and agent-to-environment interactions?

4.1.1 Agent representation

The fundamental building blocks to solve these problems are *recursion* and *continuations*. In recursion a function is defined in terms of itself: in the process of computing the output it *might* call itself with changed input data. Continuations are functions which allow to encapsulate the execution state of a program by capturing local variables (known as closure) and pick up computation from that point later on by returning a new function. As an illustratory example, we implement a continuation in Haskell which sums up integers and stores the sum locally as well as returning it as return value for the current step:

[3] The code is freely accessible from https://github.com/thalerjonathan/phd/tree/master/public/towards/SugarScape

```haskell
-- define the type of the continuation: it takes an arbitrary type a
-- and returns a type a with a new continuation
newtype Cont a = Cont (a -> (a, Cont a))

-- an instance of a continuation with type a fixed to Int
-- takes an initial value x and sums up the values passed to it
-- note that it returns adder with the new sum recursively as
-- the new continuation
adder :: Int -> Cont Int
adder x = Cont (\x' -> (x + x', adder (x + x')))

-- this function runs the given continuation for a given number of steps
-- and always passes 1 as input and prints the continuations output
runCont :: Int -> Cont Int -> IO ()
runCont 0 _ = return () -- finished
runCont n (Cont cont) = do -- pattern match to extract the function
  -- run the continuation with 1 as input, cont' is the new continuation
  let (x, cont') = cont 1
  print x
  -- recursive call, run next step
  runCont (n-1) cont'

-- main entry point of a Haskell program
-- run the continuation adder with initial value of 0 for 100 steps
main :: IO ()
main = runCont 100 (adder 0)
```

We implement an agent as a continuation: this lets us encapsulate arbitrary complex agent-state which is only visible and accessible from within the continuation - the agent has exclusive access to it. This allows also to switch behaviour dynamically e.g. switching from one mode of behaviour to another like in a state-machine, simply by returning new functions which encapsulate the new behaviour. If no change in behaviour should occur, the continuation simply recursively returns itself with the new state captured as seen in the example above.

The fact that we design an agent as a function, raises the question of the interface of it: what are the inputs and the output? Note that the type of the function has to stay the same (type *a* in the example above) although we might switch into different continuations - our interface needs to capture all possible cases of behaviour. The way we define the interface is strongly determined by the direct agent-agent interaction. In case of Sugarscape, agents need to be able to conduct two types of direct agent-agent interaction: 1. one-directional, where agent A sends a message to agent B without requiring agent B to synchronously reply to that message e.g. repaying a loan or inheriting money to children; 2. bi-directional, where two agents negotiate over multiple steps e.g. accepting a trade, mating or lending. Thus it seems reasonable to define as input type an enumeration (algebraic data-type in Haskell, see example below) which defines all possible incoming messages the agent can handle. The agents continuation is then called every time the agent receives a message and can process it, update its local state and might change its behaviour.

As output we define a data-structure which allows the agent to communicate to the simulation kernel 1. whether it wants to be removed from the system, 2. a list of new agents it wants to spawn, 3. a list of messages the agent wants to send to other agents. Further because the agents data is completely local, it also returns a data-structure which holds all *observable* information the agent wants to share with the outside world. Together with the continuation this guarantees that the agent is in full control over its local state, no one can mutate or access from outside. This also implies that information can only get out of the agent by

actually running its continuation. It also means that the output type of the function has to cover all possible input cases - it cannot change or depend on the input.

```
type AgentId    = Int
data Message    = Tick Int | MatingRequest AgentGender ...
data AgentState = AgentState { agentAge :: Int, ... }
data Observable = Observable { agentAgeObs :: Int, ... }
data AgentOut   = AgentOut
  { kill       :: Bool
  , observable :: Observable
  , messages   :: [(AgentId, Message)] -- list of messages with receiver
  }
-- agent continuation has different types for input and output
newtype AgentCont inp out = AgentCont (in -> (out, AgentCont inp out))
-- taking the initial AgentState as input and returns the continuation
sugarscapeAgent :: AgentState -> AgentCont (AgentId, Message) AgentOut
sugarscapeAgent asInit = AgentCont (\ (sender, msg) ->
  case msg of
    agentCont (sender, Tick t) = ... handle tick
    agentCont (sender, MatingRequest otherGender) = ... handle mating request)
```

4.1.2 Stepping the simulation

The simulation kernel keeps track of the existing agents and the message-queue and processes the queue one element at a time. The new messages of an agent are inserted *at the front* of the queue, ensuring that synchronous bi-directional messages are possible without violating resources constraints. The Sugarscape model specifies that in each tick or time-step all agents run in random order and do their thing. Thus to start the agent-behaviour in a new time-step, the core inserts a Tick message to each agent in random order which then results in them being executed and emitting new messages. The current time-step has finished when all messages in the queue have been processed. See algorithm 4.1.2 for the pseudo-code for the simulation stepping.

input : All agents *as*
input : List of agent observables
shuffle all agents as;
messageQueue = schedule Tick to all agents;
agentObservables = empty List;
while *messageQueue not empty* **do**
    msg = pop message from messageQueue;
    a = lookup receiving agent in as;
    (out, a') = runAgent a msg;
    update agent with continuation a' in as;
    add agent observable from out to agentObservables;
    add messages of agent at front of messageQueue;
end
return agentObservables;

Algorithm 1: Stepping the simulation.

4.1.3 Environment and agent-environment interaction

Obviously the agents in the Sugarscape are located in a discrete 2d environment where they move around and harvest resources, which means the need to read and write data of environment. This is conveniently implemented by adding a State side-effect type to the agent continuation function. Further we also add a Random effect type because dynamics in most ABS in general and Sugarscapes in particular are driven by random number streams, so our agent needs to have access to one as well. All of this low level continuation plumbing exists already as a high quality library called Dunai, based on research on Functional Reactive Programming (Hudak, Courtney, Nilsson, and Peterson 2003) and Monadic Stream Functions (Perez, Baerenz, and Nilsson 2016, Perez 2017).

5 DISCUSSION

Probably the biggest strength is that we can guarantee reproducibility at compile time: given identical initial conditions, repeated runs of the simulation will lead to same outputs. This is of fundamental importance in simulation and addressed in the Sugarscape model: *"... when the sequence of random numbers is specified ex ante the model is deterministic. Stated yet another way, model output is invariant from run to run when all aspects of the model are kept constant including the stream of random numbers."* (page 28, footnote 16) - we can guarantee that in our pure functional approach already *at compile time*.

Refactoring is very convenient and quickly becomes the norm: guided by types (change / refine them) and relying on the compiler to point out problems, results in very effective and quick changes without danger of bugs showing up at run-time. This is not possible in Python because of its lack of compiler and types, and much less effective in Java due to its dynamic type-system which is only remedied through strong IDE support.

Adding data-parallelism is easy and often requires simply swapping out a data-structure or library function against its parallel version. Concurrency, although still hard, is less painful to address and add in a pure functional setting due to immutable data and explicit side-effects. Further, the benefits of implementing concurrent ABS based on STM has been shown (TODO: cite my own TOMACS paper) which underlines the strength of Haskell for concurrent ABS due to its strong guarantees about retry-semantics.

Testing in general allows much more control and checking of invariants due to the explicit handling of effects - together with the strong static type system, nothing slips the testing-code as everything is explicit. Property-based testing in particular is a perfect match to testing ABS due to the random nature in both and because it supports convenient expressing of specifications. Thus we can conclude that in a pure functional setting, testing is very expressive and powerful and supports working towards an implementation which is very likely to be correct.

5.1 Issues

Haskell is notorious for its space-leaks due to lazy evaluation: data is only evaluated when required. Even for simple programs one can be hit hard by a serious space-leak where unevaluated code pieces (thunks) build up in memory until they are needed, leading to dramatically increased memory usage for a problem which could be solved using a fraction.

It is no surprise that our highly complex Sugarscape implementation initially suffered severely from space-leaks, piling up about 40 MByte / second. In simulation this is a big issue, threatening the value of the whole implementation despite its other benefits: because simulations might run for a (very) long time or conceptually forever, one must make absolutely sure that the memory usage stays somewhat constant. As a

remedy, Haskell allows to add so-called strictness pragmas to code-modules which forces strict evaluation of all data even if it is not used. Carefully adding this conservatively file-by file applying other techniques of forcing evaluation removed most of the memory leaks.

The main drawback of our approach is performance, which at the moment does not come close to OO implementations. There are two main reasons for it: first, FP is known for being slower due to higher level of abstractions, which are bought by slower code in general and second, updates are the main bottleneck due to immutable data requiring to copy the whole (or subparts) of a data structure in cases of a change. The first one is easily addressable through the use of data-parallelism and concurrency as we have done in our STM paper (TODO cite). The second reason can be addressed by the use of linear types (Bernardy, Boespflug, Newton, Jones, and Spiwack 2017), which allow to annotate a variable with how often it is used within a function. From this a compiler can derive aggressive optimisations, resulting in imperative-style performance but retaining the declarative nature of the code.

6 CONCLUSIONS

Our results strongly hint that our claim that pure FP has indeed its place in ABS is valid but for now we conclude that it is still too early and it is only so in cases of high-impact and large-scale simulations which results might have far-reaching consequences e.g. influence policy decisions. The reason is that engineering a proper implementation of a complex ABS model takes substantial effort in pure FP due to different techniques required. It is only there where the high requirements for reproducibility, robustness and correctness provided by FP will pay off. Still, we plan on distilling the developed techniques of the case-study into a general purpose ABS library. This should make implementing models much easier and quicker, making a pure FP approach an attractive alternative for prototyping.

6.1 Further Research

Often, ABS models build on an underlying DES core, especially when they follow an event-driven approach (Meyer 2014) where they need to schedule actions into the future. Due to the time-driven approach of the Sugarscape model, we didn't implement this in our case-study but our pure functional simulation core is conceptually already very close to a DES approach. We plan on extending it to allow true event-based ABS, which we want to feed into our general purpose ABS library as well.

Due to the recursive nature of FP we believe that it is also a natural fit to implement recursive simulations as the one discussed in (Gilmer and Sullivan 2000). In recursive ABS agents are able to halt time and 'play through' an arbitrary number of actions, compare their outcome and then to resume time and continue with a specifically chosen action e.g. the best performing or the one in which they haven't died. More precisely, an agent has the ability to run the simulation recursively a number of times where the number is not determined initially but can depend on the outcome of the recursive simulation. So recursive ABS gives each Agent the ability to run the simulation locally from its point of view to project its actions into the future and change them in the present. Due to controlled side-effects and referential transparency, combined with the recursive nature of pure FP, we think that implementing a recursive simulation in such a setting should be straight-forward.

This research is only a first step towards an even more extensive use of pure FP systems in ABS. The next step is to investigate the use of dependent types in ABS, using the pure functional language Idris (Brady 2013). Dependent types allow to express even stronger guarantees at compile time, going as far as programs being valid proofs for the correctness of the implemented feature.

ACKNOWLEDGMENTS

The authors would like to thank J. Hey and M. Handley for valuable feedback and discussions.

REFERENCES

Allen, C., and J. Moronuki. 2016, July. *Haskell Programming from First Principles*. Allen and Moronuki Publishing. Google-Books-ID: 5FaXDAEACAAJ.

Bernardy, J.-P., M. Boespflug, R. R. Newton, S. P. Jones, and A. Spiwack. 2017, December. "Linear Haskell: practical linearity in a higher-order polymorphic language". *Proceedings of the ACM on Programming Languages* vol. 2 (POPL), pp. 1–29. arXiv: 1710.09756.

Bezirgiannis, N. 2013. *Improving Performance of Simulation Software Using Haskells Concurrency & Parallelism*. Ph. D. thesis, Utrecht University - Dept. of Information and Computing Sciences.

Botta, N., A. Mandel, and C. Ionescu. 2010. "Time in discrete agent-based models of socio-economic systems". Documents de travail du Centre d'Economie de la Sorbonne 10076, Université Panthéon-Sorbonne (Paris 1), Centre d'Economie de la Sorbonne.

Brady, E. 2013. "Idris, a general-purpose dependently typed programming language: Design and implementation". *Journal of Functional Programming* vol. 23 (05), pp. 552–593.

Church, A. 1936, April. "An Unsolvable Problem of Elementary Number Theory". *American Journal of Mathematics* vol. 58 (2), pp. 345–363.

Claessen, K., and J. Hughes. 2000. "QuickCheck - A Lightweight Tool for Random Testing of Haskell Programs". In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pp. 268–279. New York, NY, USA, ACM.

De Jong, T. 2014. "Suitability of Haskell for Multi-Agent Systems". Technical report, University of Twente.

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

Gilmer, Jr., J. B., and F. J. Sullivan. 2000. "Recursive Simulation to Aid Models of Decision Making". In *Proceedings of the 32Nd Conference on Winter Simulation*, WSC '00, pp. 958–963. San Diego, CA, USA, Society for Computer Simulation International.

Harris, T., S. Marlow, S. Peyton-Jones, and M. Herlihy. 2005. "Composable Memory Transactions". In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pp. 48–60. New York, NY, USA, ACM.

Hudak, P., A. Courtney, H. Nilsson, and J. Peterson. 2003. "Arrows, Robots, and Functional Reactive Programming". In *Advanced Functional Programming*, edited by J. Jeuring and S. L. P. Jones, Number 2638 in Lecture Notes in Computer Science, pp. 159–187. Springer Berlin Heidelberg.

Hudak, P., J. Hughes, S. Peyton Jones, and P. Wadler. 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.

Hughes, J. 1989, April. "Why Functional Programming Matters". *Comput. J.* vol. 32 (2), pp. 98–107.

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

Jankovic, P., and O. Such. 2007. "Functional Programming and Discrete Simulation". Technical report.

Jones, S. P. 2002. "Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell". In *Engineering theories of software construction*, pp. 47–96, Press.

Lipovaca, M. 2011, April. *Learn You a Haskell for Great Good!: A Beginner's Guide*. 1 edition ed. San Francisco, CA, No Starch Press.

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

Marlow, S. 2013. *Parallel and Concurrent Programming in Haskell*. O'Reilly. Google-Books-ID: k0W6AQAACAAJ.

Marlow, S., S. Peyton Jones, and S. Singh. 2009. "Runtime Support for Multicore Haskell". In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pp. 65–78. New York, NY, USA, ACM.

Meyer, R. 2014, May. "Event-Driven Multi-agent Simulation". In *Multi-Agent-Based Simulation XV*, Lecture Notes in Computer Science, pp. 3–16, Springer, Cham.

Michaelson, G. 2011. *An Introduction to Functional Programming Through Lambda Calculus*. Courier Corporation. Google-Books-ID: gKvwPtvsSjsC.

Moggi, E. 1989. "Computational Lambda-calculus and Monads". In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pp. 14–23. Piscataway, NJ, USA, IEEE Press.

North, M. J., and C. M. Macal. 2007, March. *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., J. Goerzen, and D. Stewart. 2008. *Real World Haskell*. 1st ed. O'Reilly Media, Inc.

Perez, I. 2017, October. *Extensible and Robust Functional Reactive Programming*. Doctoral Thesis, University Of Nottingham, Nottingham.

Perez, I., M. Baerenz, and H. Nilsson. 2016. "Functional Reactive Programming, Refactored". In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, pp. 33–44. New York, NY, USA, ACM.

Schneider, O., C. Dutchyn, and N. Osgood. 2012. "Towards Frabjous: A Two-level System for Functional Reactive Agent-based Epidemic Simulation". In *Proceedings of the 2Nd ACM SIGHIT International Health Informatics Symposium*, IHI '12, pp. 785–790. New York, NY, USA, ACM.

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

Sulzmann, M., and E. Lam. 2007. "Specifying and Controlling Agents in Haskell". Technical report.

Turing, A. M. 1937. "On Computable Numbers, with an Application to the Entscheidungsproblem". *Proceedings of the London Mathematical Society* vol. s2-42 (1), pp. 230–265.

Vendrov, I., C. Dutchyn, and N. D. Osgood. 2014, April. "Frabjous A Declarative Domain-Specific Language for Agent-Based Modeling". In *Social Computing, Behavioral-Cultural Modeling and Prediction*, edited by W. G. Kennedy, N. Agarwal, and S. J. Yang, Number 8393 in Lecture Notes in Computer Science, pp. 385–392. Springer International Publishing.

Wadler, P. 1992. "The Essence of Functional Programming". In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pp. 1–14. New York, NY, USA, ACM.

Wadler, P. 1995. "Monads for Functional Programming". In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pp. 24–52. London, UK, UK, Springer-Verlag.

Wadler, P. 1997, September. "How to Declare an Imperative". *ACM Comput. Surv.* vol. 29 (3), pp. 240–263.