Functional Pearl: Pure functional epidemics

An Agent-Based Approach

Jonathan Thaler
Thorsten Altenkirch
Peer-Olaf Siebers
jonathan.thaler@nottingham.ac.uk
thorsten.altenkirch@nottingham.ac.uk
peer-olaf.siebers@nottingham.ac.uk
University of Nottingham
Nottingham, United Kingdom

Abstract

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. In this functional pearl we develop an elegant agent-based implementation of the SIR model of epidemiology, which allows to simulate the spreading of an infectious disease through a population, using using Functional Reactive Programming and Monadic Stream Functions.

Keywords Functional Reactive Programming, Monadic Stream Functions, Agent-Based Simulation

ACM Reference Format:
Jonathan Thaler, Thorsten Altenkirch, and Peer-Olaf Siebers. 2018. Functional Pearl: Pure functional epidemics: An Agent-Based Approach. In *Proceedings of Haskell Symposium (HS18).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [1] 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* [4] which still holds up today.

In this paper we fundamentally challenge this metaphor and explore ways of approaching ABS in a pure functional way using Haskell. By doing this we expect to leverage the benefits of pure functional programming [2]: higher expressivity through declarative code, being polymorph and explicit about side-effects through monads, more robust and less susceptible for bugs due to explicit data flow and lack of implicit side-effects.

As use case we introduce the simple SIR model of epidemiology with which one can simulate epidemics, that is the spreading of an infectious disease through a population, in a realistic way.

Over the course of four steps, we derive all necessary concepts required for a full agent-based implementation. We start from a very simple solution running in the Random Monad which has all general concepts already there and then refine it in various ways, making the transition to Functional Reactive Programming (FRP) [9] and to Monadic Stream Functions (MSF) [6].

The aim of this paper is to show how ABS can be done in *pure* Haskell 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.

The contributions of this paper are:

- To the best of our knowledge, we are the first to *systematically* introduce the concepts of ABS to the *pure* functional programming paradigm in a step-by-step approach. It is also the first paper to show how to apply Arrowized FRP to ABS on a technical level, presenting a new field of application to FRP.
- Our approach shows how robustness can be achieved through purity which guarantees reproducibility at compile time, something not possible with traditional object-oriented approaches.
- The result of using Arrowized FRP is a conceptually much cleaner approach to ABS than traditional imperative object-oriented approaches. It allows expressing continuous time-semantics in a much clearer, compositional and declarative way, without having to deal with low-level details related to the progress of time.

HS18, 2018, 09
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Section ?? discusses related work. In section ?? we introduce functional reactive programming, arrowized programming and monadic stream functions, because our approach builds heavily on these concepts. Section ?? defines agent-based simulation. In section ?? we introduce the SIR model of epidemiology as an example model to explain the concepts of ABS. The heart of the paper is section ?? in which we derive the concepts of a pure functional approach to ABS in four steps, using the SIR model. Finally, we draw conclusions and discuss issues in section 2 and point to further research in section ??.

1.1 Generalising to Monadic Stream Functions

A part of the library Dunai is BearRiver, a wrapper which re-implements Yampa on top of Dunai, which should allow us to easily replace Yampa with MSFs. This will enable us to run arbitrary monadic computations in a signal function, which we will need in the next step when adding an environment.

1.1.1 Identity Monad

We start by making the transition to BearRiver by simply replacing Yampas signal function by BearRivers' which is the same but takes an additional type parameter *m* indicating the monadic context. If we replace this type-parameter with the Identity Monad we should be able to keep the code exactly the same, except from a few type-declarations, because BearRiver re-implements all necessary functions we are using from Yampa. We simply re-define our agent signal function, introducing the monad stack our SIR implementation runs in:

```
type SIRMonad    = Identity
type SIRAgent    = SF SIRMonad [SIRState] SIRState
```

1.1.2 Random Monad

Using the Identity Monad does not gain us anything but it is a first step towards a more general solution. Our next step is to replace the Identity Monad by the Random Monad which will allow us to get rid of the RandomGen arguments to our functions and run the whole simulation within the Random Monad *again* just as we started but now with the full features functional reactive programming. We start by re-defining the SIRMonad and SIRAgent:

```
type SIRMonad g = Rand g
type SIRAgent g = SF (SIRMonad g) [SIRState] SIRState
```

The question is now how to access this Random Monad functionality within the MSF context. For the function *occasionally*, there exists a monadic pendant *occasionallyM* which requires a MonadRandom type-class. Because we are now running within a MonadRandom instance we simply replace *occasionally* with *occasionallyM*.

1.1.3 Discussion

So far making the transition to MSFs does not seem as compelling as making the move from the Random Monad to FRP in the beginning. Running in the Random Monad within FRP is convenient but we could achieve the same with passing RandomGen around as we already demonstrated. In the next step we introduce the concept of a read/write environment which we realise using a StateT monad. This will show the real benefit of the transition to MSFs as without it, implementing a general environment access would be quite cumbersome.

(a) von Neumann **(b)** Moore

Figure 1. Common neighbourhoods in discrete 2D environments of Agent-Based Simulation.

1.2 Adding an environment

In this step we will add an environment in which the agents exist and through which they interact with each other. This is a fundamental different approach to agent interaction but is as valid as the approach in the previous steps.

In ABS agents are often situated within a discrete 2D environment [1] which is simply a finite *NxM* grid with either a Moore or von Neumann neighbourhood (Figure 1). Agents are either static or can move freely around with cells allowing either single or multiple occupants.

We can directly map the SIR model to a discrete 2D environment by placing the agents on a corresponding 2D grid with an unrestricted neighbourhood. The behaviour of the agents is the same but they select their interactions directly from the environment. Also instead of feeding back the states of all agents as inputs, agents now communicate through the environment by revealing their current state to their neighbours by placing it on their cell. Agents can read the states of all their neighbours which tells them if a neighbour is infected or not. This allows us to implement the infection mechanism as in the beginning. For purposes of a more interesting approach, we restrict the neighbourhood to Moore (Figure 1b).

1.2.1 Implementation

We start by defining our discrete 2D environment for which we use an indexed two dimensional array. In each cell the agents will store their current state, thus we use the *SIRState* as type for our array data:

```
type Disc2dCoord = (Int, Int)
type SIREnv      = Array Disc2dCoord SIRState
```

Next we redefine our monad stack and agent signal function. We use a StateT transformer on top of our Random Monad from the previous step with *SIREnv* as type for the

state. Our agent signal function now has unit input and output type, which indicates that the actions of the agents are only visible through side-effects in the monad stack they are running in.

```
type SIRMonad g = StateT SIREnv (Rand g)
type SIRAgent g = SF (SIRMonad g) () ()
```

The implementation of a susceptible agent is now a bit different and a mix between previous steps. The agent directly queries the environment for its neighbours and randomly selects one of them. The remaining behaviour is similar:

```
susceptibleAgent :: RandomGen g => Disc2dCoord -> SIRAgent g
susceptibleAgent coord
    = switch susceptible (const (infectedAgent coord))
  where
    susceptible :: RandomGen g
      => SF (SIRMonad g) () ((), Event ())
    susceptible = proc _ -> do
      makeContact <- occasionallyM (1 / contactRate) () -< ()
      if not (isEvent makeContact)
        then returnA -< ((), NoEvent)
        else (do
          env <- arrM_ (lift get) -< ()
          let ns = neighbours env coord agentGridSize moore
          s <- drawRandomElemS -< ns
          case s of
            Infected -> do
              infected <- arrM_
                (lift $ lift $ randomBoolM infectivity) -< ()
              if infected
                then (do
                  arrM (put . changeCell coord Infected) -< env
                  returnA -< ((), Event ()))
                else returnA -< ((), NoEvent)
            _        -> returnA -< ((), NoEvent))
```

Querying the neighbourhood is done using the *neighbours :: SIREnv -> Disc2dCoord -> Disc2dCoord -> [Disc2dCoord] -> [SIRState]* function. It takes the environment, the coordinate for which to query the neighbours for, the dimensions of the 2D grid and the neighbourhood information and returns the data of all neighbours it could find. Note that on the edge of the environment, it could be the case that fewer neighbours than provided in the neighbourhood information will be found due to clipping.

The behaviour of an infected agent is nearly the same as in the previous step, with the difference that upon recovery the infected agent updates its state in the environment from Infected to Recovered.

Running the simulation with MSFs works slightly different. The function *embed* we used before is not provided by BearRiver but by Dunai which has important implications. Dunai does not know about time in MSFs, which is exactly what BearRiver builds on top of MSFs. It does so by adding a ReaderT Double which carries the Δt. This is the reason why we need lifts e.g. in case of getting the environment. Thus *embed* returns a computation in the ReaderT Double Monad which we need to peel away using *runReaderT*. This then results in a StateT computation which we evaluate by using *evalStateT* and an initial environment as initial state. This then results in another monadic computation of the Random Monad type which we evaluate using *evalRand* which delivers the final result. Note that instead of returning agent states we simply return a list of environments, one for each step. The agent states can then be extracted from each environment.

```
runSimulation :: RandomGen g => g -> Time -> DTime
  -> SIREnv -> [(Disc2dCoord, SIRState)] -> [SIREnv]
runSimulation g t dt env as = evalRand esRand g
  where
    steps    = floor (t / dt)
    dts      = replicate steps ()
    -- initial SFs of all agents
    sfs      = map (uncurry sirAgent) as
    -- running the simulation
    esReader = embed (stepSimulation sfs) dts
    esState  = runReaderT esReader dt
    esRand   = evalStateT esState env
```

Due to the different approach of returning the SIREnv in every step, we implemented our own MSF:

```
stepSimulation :: RandomGen g
  => [SIRAgent g] -> SF (SIRMonad g) () SIREnv
stepSimulation sfs = MSF (\_ -> do
  -- running all SFs with unit input
  res <- mapM (`unMSF` ()) sfs
  -- extracting continuations, ignore output
  let sfs' = fmap snd res
  -- getting environment of current step
  env <- get
  -- recursive continuation
  let ct = stepSimulation sfs'
  return (env, ct))
```

1.2.2 Results

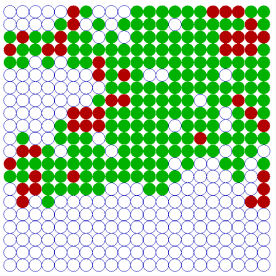
We implemented rendering of the environments using the gloss library which allows us to cycle arbitrarily through the steps and inspect the spreading of the disease over time visually as seen in Figure 2.

Note that the dynamics of the spatial SIR simulation which are seen in Figure 2b look quite different from the SD dynamics of Figure **??**. This is due to a much more restricted neighbourhood which results in far fewer infected agents at a time and a lower number of recovered agents at the end of the epidemic, meaning that fewer agents got infected overall.

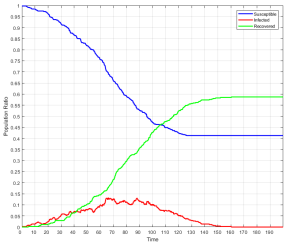
1.2.3 Discussion

At first the environment approach might seem a bit overcomplicated and one might ask what we have gained by using an unrestricted neighbourhood where all agents can contact all others. The real win is that we can introduce arbitrary restrictions on the neighbourhood as shown with the Moore neighbourhood.

Of course an environment is not restricted to be a discrete 2D grid and can be anything from a continuous N-dimensional space to a complex network - one only needs



(a) $t = 100$



(b) Dynamics over time

Figure 2. Simulating the agent-based SIR model on a 21x21 2D grid with Moore neighbourhood (Figure 1b), a single infected agent at the center and same SIR parameters as in Figure **??**. Simulation run until $t = 200$ with fixed $\Delta t = 0.1$. Last infected agent recovers shortly after $t = 160$. The susceptible agents are rendered as blue hollow circles for better contrast.

to change the type of the StateT monad and provide corresponding neighbourhood querying functions. The ability to place the heterogeneous agents in a generic environment is also the fundamental advantage of an agent-based over the SD approach and allows to simulate much more realistic scenarios.

2 Conclusions

Our approach is radically different from traditional approaches in the ABS community. First it builds on the already quite powerful FRP paradigm. Second, due to our hybrid approach, it forces one to think properly of time-semantics of the model and how small Δt should be. Third it requires to think about agent interactions in a new way instead of being just method-calls.

Because no part of the simulation runs in the IO Monad and we do not use unsafePerformIO we can rule out a serious class of bugs caused by implicit data-dependencies and side-effects which can occur in traditional imperative implementations.

Also we can statically guarantee the reproducibility of the simulation. Within the agents there are no side effects possible which could result in differences between same runs. Every agent has access to its own random-number generator or the Random Monad, allowing randomness to occur in the simulation but the random-generator seed is fixed in the beginning and can never be changed within an agent. This means that after initialising the agents, which *could* run in the IO Monad, the simulation itself runs completely deterministic.

Determinism is also ensured by fixing the Δt and not making it dependent on the performance of e.g. a rendering-loop or other system-dependent sources of non-determinism as described by [7]. Also by using FRP we gain all the benefits from it and can use research on testing, debugging and exploring FRP systems [7], [5].

Issues

Unfortunately, the hybrid approach of SD/ABS amplifies the performance issues of agent-based approaches, which requires much more processing power compared to SD, because each agent is modelled individually in contrast to aggregates in SD [3]. With the need to sample the system with high frequency, this issue gets worse. We haven't investigated how to optimize the performance by using efficient functional data structures, hence in the moment our program performs much worse than an imperative implementation that exploits in-place updates.

Despite the strengths and benefits we get by leveraging on FRP, there are errors that are not raised at compile-time, e.g. we can still have infinite loops and run-time errors. This was for example investigated by [8] who use dependent types to avoid some runt-time errors in FRP. We suggest that one could go further and develop a domain specific type system for FRP that makes the FRP based ABS more predictable and that would support further mathematical analysis of its properties.

We can conclude that the main difficulty of a pure functional approach evolves around the communication and interaction between agents. This is straight-forward in object-oriented programming, where it is achieved using method-calls mutating the internal state of the agent, but that comes at the cost of a new class of bugs due to implicit data flow. In pure functional programming these data flows are explicit but our current approach of feeding back the states of all agents as inputs is not very general and we have added further mechanisms of agent interaction which we had to omit due to lack of space. We hypothesise that MSFs allow us to conveniently express agent communication but but leave this for further research.

We started with high hopes for the pure functional approach and hypothesized that it will be truly superior to existing traditional object-oriented approaches but we come to the conclusion that this is not so. The single real benefit is the lack of implicit side-effects and reproducibility guaranteed at compile time. Still, our research was not in vain as we see it as an intermediary step towards using dependent types. Moving to dependent types would pose a unique benefit over the object-oriented approach and should allow us to express and guarantee properties at compile time which is not possible with imperative approaches. We leave this for further research.

Acknowledgments

The authors would like to thank I. Perez, H. Nilsson, J. Greensmith, M. Baerenz, H. Vollbrecht, S. Venkatesan and J. Hey

and the referees of Haskell Symposium 2018 for constructive feedback, comments and valuable discussions.

References

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

[2] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip 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)*. ACM, New York, NY, USA, 12–1–12–55. https://doi.org/10.1145/1238844.1238856

[3] Charles M. Macal. 2010. To Agent-based Simulation from System Dynamics. In *Proceedings of the Winter Simulation Conference (WSC '10)*. Winter Simulation Conference, Baltimore, Maryland, 371–382. http://dl.acm.org/citation.cfm?id=2433508.2433551

[4] Michael J. North and Charles M. Macal. 2007. *Managing Business Complexity: Discovering Strategic Solutions with Agent-Based Modeling and Simulation.* Oxford University Press, USA. Google-Books-ID: gRATDAAAQBAJ.

[5] Ivan Perez. 2017. Back to the Future: Time Travel in FRP. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. ACM, New York, NY, USA, 105–116. https://doi.org/10.1145/3122955.3122957

[6] Ivan Perez, Manuel Baerenz, and Henrik Nilsson. 2016. Functional Reactive Programming, Refactored. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, New York, NY, USA, 33–44. https://doi.org/10.1145/2976002.2976010

[7] Ivan Perez and Henrik Nilsson. 2017. Testing and Debugging Functional Reactive Programming. *Proc. ACM Program. Lang.* 1, ICFP (Aug. 2017), 2:1–2:27. https://doi.org/10.1145/3110246

[8] 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

[9] Zhanyong Wan and Paul Hudak. 2000. Functional Reactive Programming from First Principles. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 242–252. https://doi.org/10.1145/349299.349331

Received March 2018