Pure Functional Epidemics
An Agent-Based Approach

Jonathan Thaler Thorsten Altenkirch Peer-Olaf Siebers

University of Nottingham, United Kingdom

IFL 2018

Research Question(s)

- **How** can we implement Agent-Based Simulation (ABS) in (pure) functional programming?
- **What** are the benefits and drawbacks?

Agent-Based Simulation (ABS)

Example

Simulate *the spread of an infectious disease in a city.*
What are the ***dynamics*** *(peak, duration of disease)?*

1. Start with population → Agents
2. Situated in City → Environment
3. Interacting with each other → local interactions
4. Creating dynamics → emergent system behaviour
5. Therefore ABS → bottom-up approach

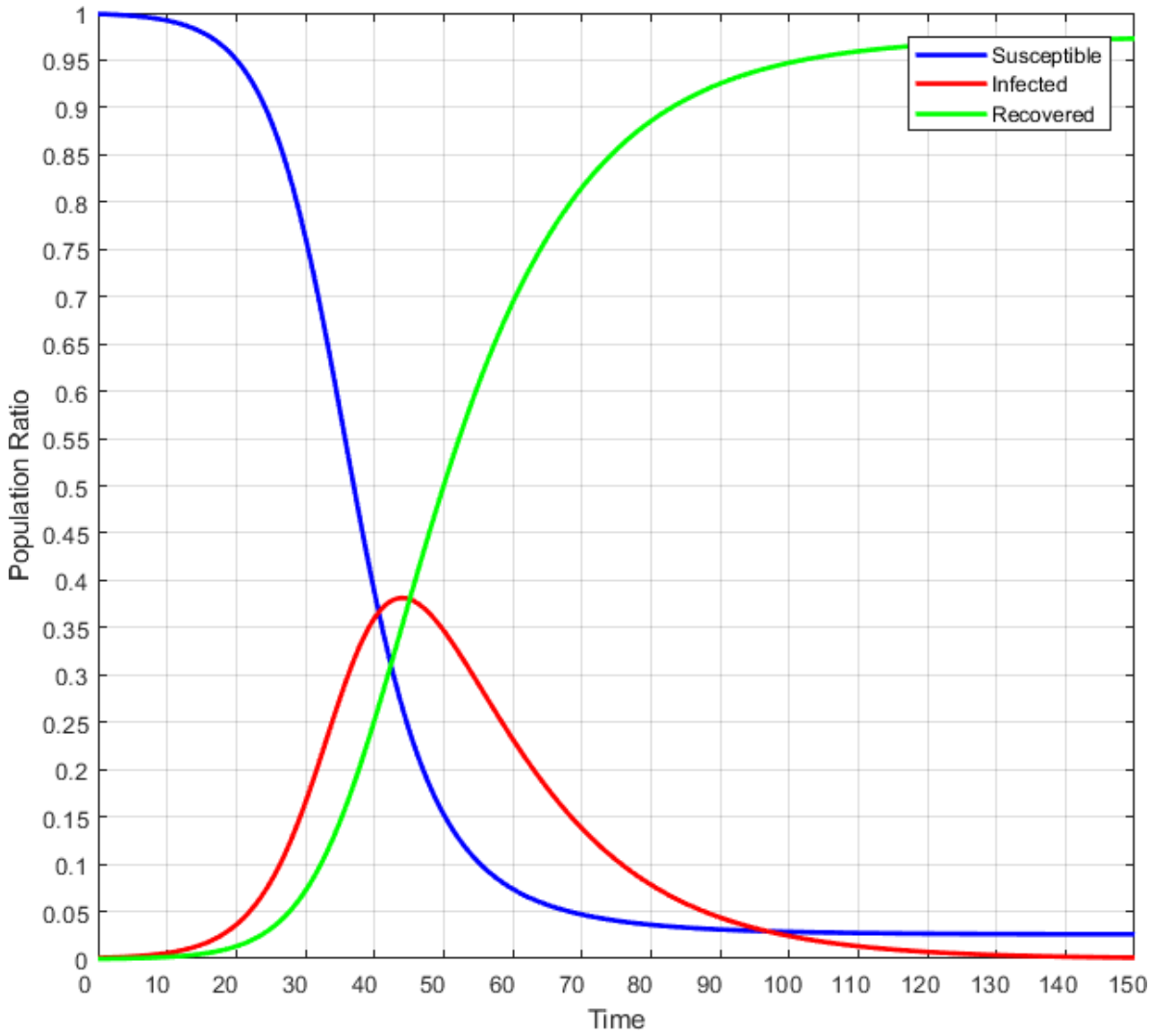
SIR Model

- Population size $N = 1,000$
- Contact rate $\beta = 0.2$
- Infection probability $\gamma = 0.05$
- Illness duration $\delta = 15$
- 1 initially infected agent

System Dynamics

Top-Down, formalised using Differential Equations, give rise to dynamics.

SIR Model Dynamics



How-To implement ABS?

Established, state-of-the-art approach in ABS

Object-Oriented Programming in Python, Java,...

We want (pure) functional programming

Purity, explicit about side-effects, declarative, reasoning, parallelism, concurrency, property-based testing,...

How can we do it?

Functional Reactive Programming

Functional Reactive Programming (FRP)

- Continuous- & discrete-time systems in functional programming
- Signal Function (SF)
  $\Rightarrow$ process over time, maps signal to signal
- Events (deterministic & stochastic)
- Random-number streams
- *Arrowized* FRP using the *Yampa* library

Signal Function

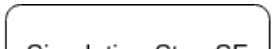
TODO: Signal Function conceptually explained TODO: add diagrams and explain FRP and arrowized notation on high level

FRP Combinators

TODO: switch TODO: occasionally TODO: after TODO: noiseR

Update Semantics

TODO
TODO: sequential is wrong semantics because all agents act at the sa.e time, if we impose an ordering it could be the case that e.g. the infection spreads from top left corner to bottom right but not the other way round. for many ab models which run sequentially the established approach is thus to shuffle the agents in each step to avoid these kind of semantic problems. In FP we can enforce such an iteation strategy already in the types. the flaw reflects exactly my iterating paper message: the strategy needs to reflect the semantics of the model
TODO: make clear that ABS often runs agents sequentially and shuffles them. there is no agreed "true" way as we have shown as itvresults in different semantics but in functional programming the parallel approach is the best fit. and for SIR its the only correct one



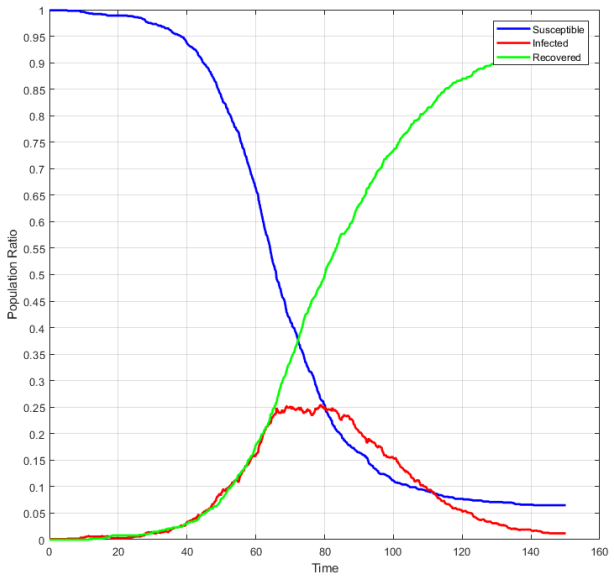
Some Types...

```
1  data SIRState = Susceptible | Infected | Recovered
2
3  type SIRAgent = SF [SIRState] SIRState
4
5  sirAgent :: RandomGen g => g -> SIRState -> SIRAgent
6  sirAgent g Susceptible = susceptibleAgent g
7  sirAgent g Infected    = infectedAgent g
8  sirAgent _ Recovered   = recoveredAgent
9
10 recoveredAgent :: SIRAgent
11 recoveredAgent = arr (const Recovered)
```

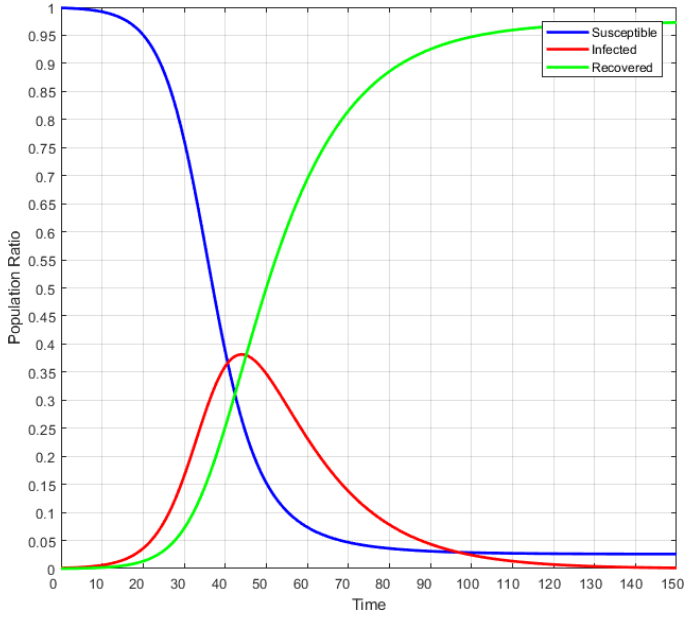
Susceptible Agent

```haskell
susceptibleAgent :: RandomGen g => g -> SIRAgent
susceptibleAgent g
    = switch
      -- delay switching by 1 step to prevent against transition
      -- from Susceptible to Recovered within one time-step
      (susceptible g >>> iPre (Susceptible, NoEvent))
      (const (infectedAgent g))
  where
    susceptible :: RandomGen g
      => g -> SF [SIRState] (SIRState, Event ())
    susceptible g = proc as -> do
      makeContact <- occasionally g (1 / contactRate) () -< ()
      if isEvent makeContact
        then (do
          -- draw random element from the list
          a <- drawRandomElemSF g -< as
          case a of
            Infected -> do
              -- returns True with given probability
              i <- randomBoolSF g infectivity -< ()
              if i
                then returnA -< (Infected, Event ())
                else returnA -< (Susceptible, NoEvent)
            _        -> returnA -< (Susceptible, NoEvent))
        else returnA -< (Susceptible, NoEvent)
```

Dynamics $\Delta t = 0.1$

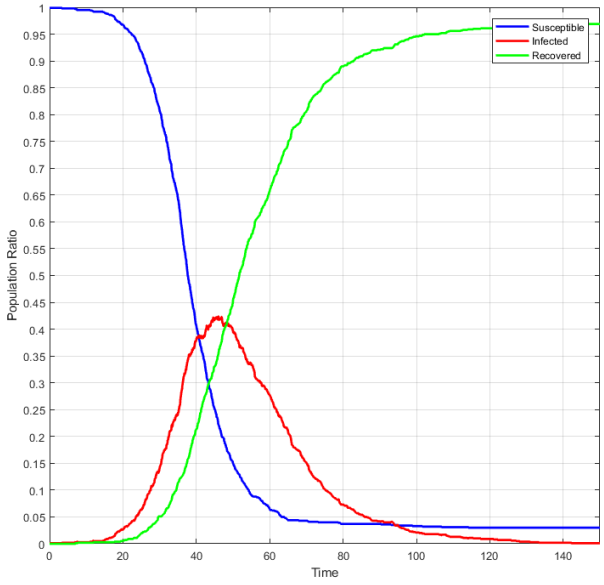


(a) Agent-Based approach

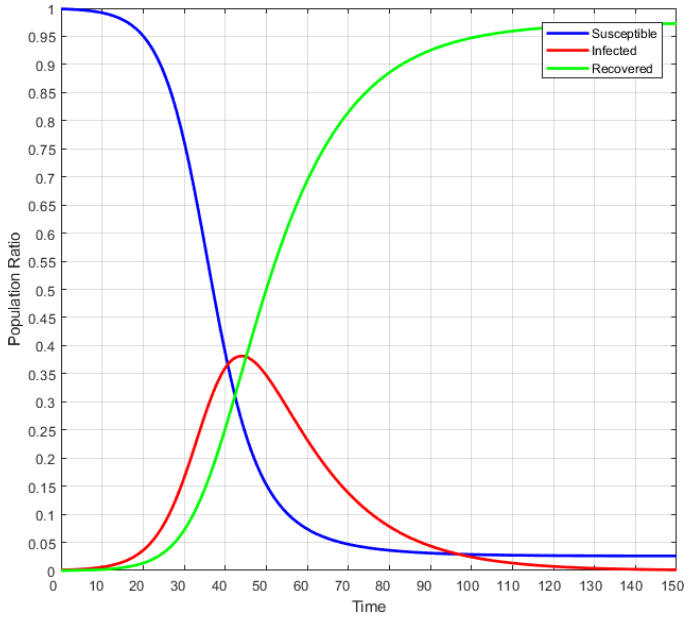


(b) System Dynamics

Dynamics $\Delta t = 0.01$



(a) Agent-Based approach



(b) System Dynamics

Reflection

- **Time**
- **Agents**
- **Feedback**
- **Stochasticity**
- **Deterministic**
- **Parallel, lock-step semantics**
- Problem: Correlated Random Numbers !!
- Missing: Spatiality

Solving Random Number Correlation

Elegant Approach

Random Monad

Problem

Yampa not monadic

Solution

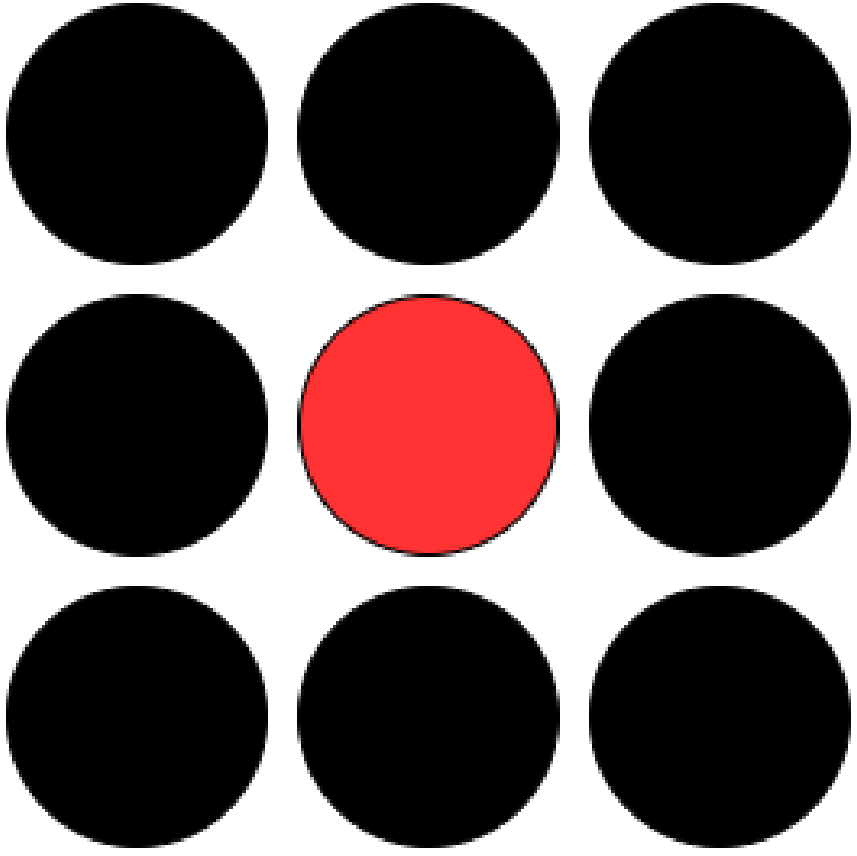
Monadic Stream Functions (MSFs)
$\Rightarrow$ Signal Functions with monadic context

Monadic Stream Functions

TODO

Where is the Environment?

TODO



Re-defining Some Types

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

type SIRAgent g = SF (Rand g) SIREnv SIRState

neighbours :: SIREnv -> Disc2dCoord -> Disc2dCoord -> [Disc2dCoord] -> [SIRState]

simulationStep :: RandomGen g => [(SIRAgent g, Disc2dCoord)]
               -> SIREnv -> SF (Rand g) () SIREnv
simulationStep sfsCoords env = MSF (\_ -> do
  let (sfs, coords) = unzip sfsCoords
  -- run agents sequentially but with shared, read-only environment
  ret <- mapM (`unMSF` env) sfs
  -- construct new environment from all agent outputs for next step
  let (as, sfs') = unzip ret
      env' = foldr (\ (a, coord) envAcc -> updateCell coord a envAcc)
               env (zip as coords)

      sfsCoords' = zip sfs' coords
      cont       = simulationStep sfsCoords' env'
  return (env', cont))

updateCell :: Disc2dCoord -> SIRState -> SIREnv -> SIREnv
```

Dynamics with environment

Conclusion

- Purity guarantees reproducibility at compile time
- Enforce and guarantee update semantics at compile time
- Performance :(
- Agent-Identity a bit lost
- Agent-Interaction is main difficulty

Thank You!