

# Pure Functional Epidemics An Agent-Based Approach

Jonathan Thaler   Thorsten Altenkirch   Peer-Olaf Siebers

University of Nottingham, United Kingdom

IFL 2018

## Research Questions

- **How** can we implement Agent-Based Simulation in (pure) functional programming e.g. Haskell?
- **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

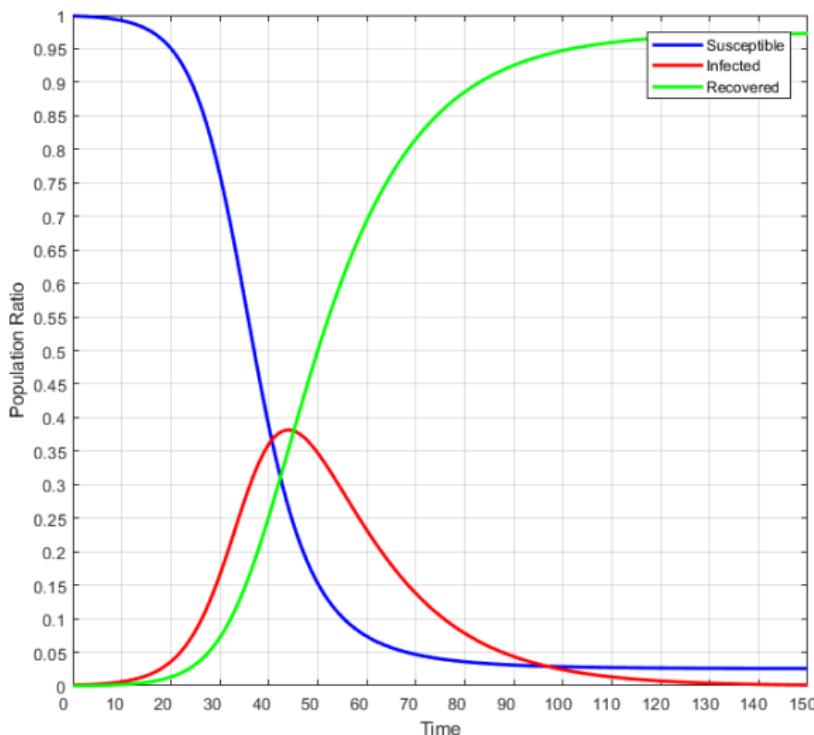


- Population size  $N = 1,000$
- Contact rate  $\beta = 0.2$
- Infection probability  $\gamma = 0.05$
- Illness duration  $\delta = 15$
- 1 initially infected agent

## 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 FP
- Signal Function
- Events
- Random-number streams
- *Arrowized FRP using the Yampa library*

## Signal Function (SF)

### Process over time

$$\begin{aligned} SF \alpha \beta &\approx Signal \alpha \rightarrow Signal \beta \\ Signal \alpha &\approx Time \rightarrow \alpha \end{aligned}$$

### Agents as Signal Functions

- Clean interface (input / output)
- Pro-activity by perceiving time

## FRP combinators

### Dynamic change of behaviour

```
switch :: SF inp (out, Event e) -> (e -> SF inp out) -> SF inp out
```

### Stochastic event source

```
occasionally :: RandomGen g => g -> Time -> b -> SF a (Event b)
```

### Random number stream

```
noiseR :: (RandomGen g, Random b) => (b, b) -> g -> SF a b
```

### Infinitesimal delay (1 step)

```
iPre :: a -> SF a a
```

# Arrowized Programming

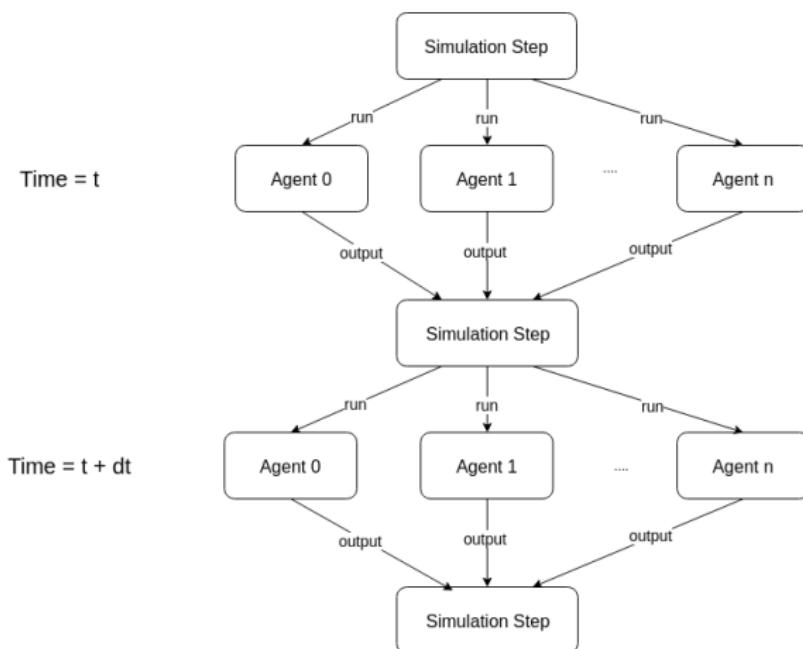
## Monads

```
do
    out1 ← comp1
    out2 ← comp2 out1
    return out2
```

## Arrows

```
proc input do
    out1 ← comp1 ⤵ input
    out2 ← comp2 ⤵ out1
    returnA ⤵ out2
```

# Update Semantics



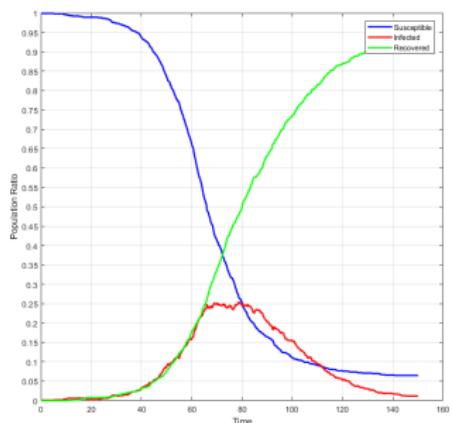
## Some Types...

```
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)
```

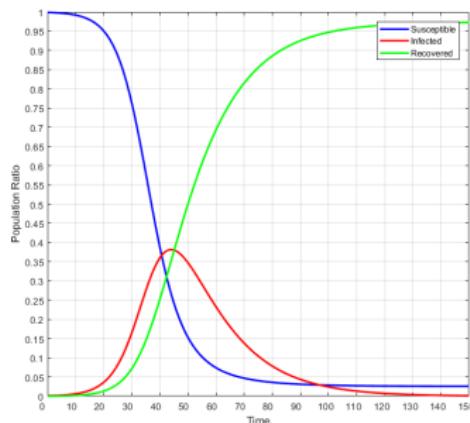
# Susceptible Agent

```
1  susceptibleAgent :: RandomGen g => g -> SIRAgent
2  susceptibleAgent g
3      = switch
4          -- delay switching by 1 step to prevent against transition
5          -- from Susceptible to Recovered within one time-step
6          (susceptible g >>> iPre (Susceptible, NoEvent))
7          (const (infectedAgent g))
8  where
9      susceptible :: RandomGen g => g -> SF [SIRState] (SIRState, Event ())
10     susceptible g = proc as -> do
11         -- generate an event on average with given rate
12         makeContact <- occasionally g (1 / contactRate) () -< ()
13         if isEvent makeContact
14             then (do
15                 -- draw random contact
16                 a <- drawRandomElemSF g -< as
17                 case a of
18                     -- contact with infected => get infected with prob.
19                     Infected -> do
20                         -- returns True with given probability
21                         i <- randomBoolSF g infectivity -< ()
22                         if i
23                             -- got infected => infection event => transition to infected
24                             then returnA -< (Infected, Event ())
25                             else returnA -< (Susceptible, NoEvent)
26                         -> returnA -< (Susceptible, NoEvent))
27             else returnA -< (Susceptible, NoEvent)
```

## Dynamics $\Delta t = 0.1$

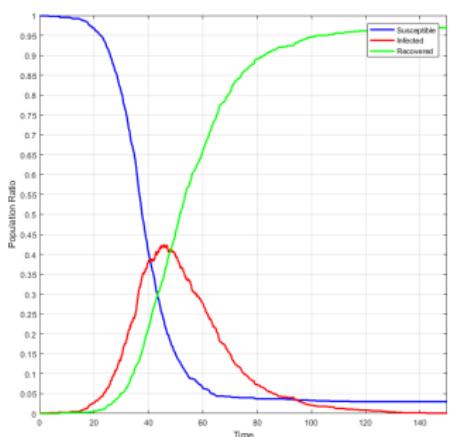


(a) Agent-Based approach

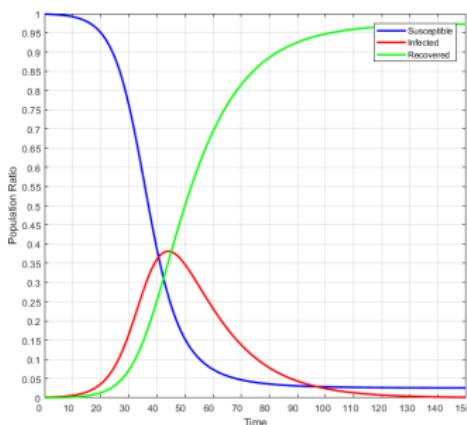


(b) System Dynamics

## Dynamics $\Delta t = 0.01$



**(a)** Agent-Based approach



**(b)** System Dynamics

# Solving Random Number Correlation

## Elegant Approach

*Random Monad*

## Problem

*Yampa not monadic*

## Solution

Monadic Stream Functions

# Monadic Stream Functions (MSFs)

## Concept

- Signal Functions + monadic context
- *Dunai* - Perez et al
- *BearRiver* - *Yampa* built on top of *Dunai*

## Definition

```
newtype MSF m a b = MSF {unMSF :: MSF m a b -> a -> m (b, MSF m a b) }

arrM :: Monad m -> (a -> m b) -> MSF m a b
arrM_ :: Monad m -> m b -> MSF m a b
```

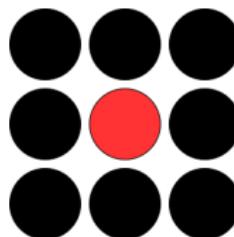
## Monadic Stochastic Event Source

```
occasionallyM :: MonadRandom m => Time -> b -> SF m a (Event b)
```

## Monadic Agent Signal Function

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

# Defining Spatiality



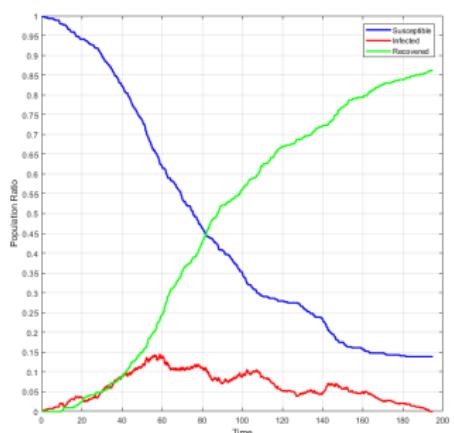
Moore Neighbourhood

## Some types...

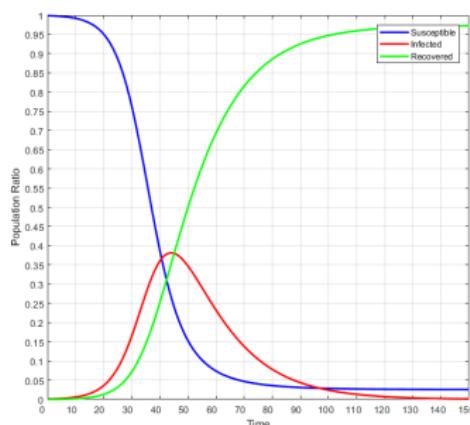
```
type Disc2dCoord = (Int, Int)
type SIREnv      = Array Disc2dCoord SIRState

type SIRAgent g  = SF (Rand g) SIREnv SIRState
```

# Spatial Dynamics



(a) Agent-Based



(b) System Dynamics

# Spatial Visualisation

# Conclusion

- Purity guarantees reproducibility
- Enforce and guarantee update semantics
- Performance :(

Thank You!