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

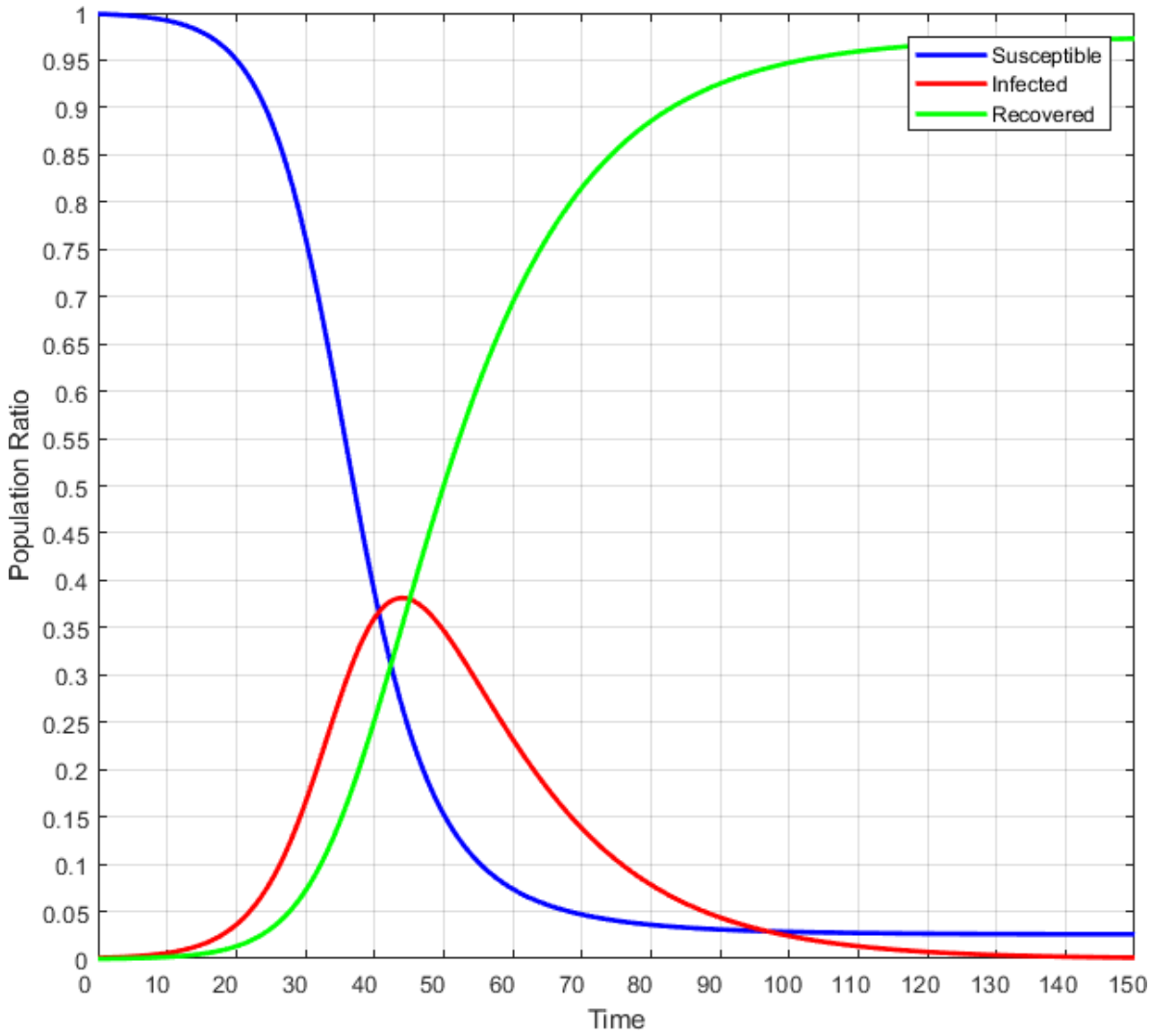
SIR Model

- 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

$$\mathit{SF}\ \alpha\ \beta \approx \mathit{Signal}\ \alpha \to \mathit{Signal}\ \beta$$

$$\mathit{Signal}\ \alpha \approx \mathit{Time} \to \alpha$$

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)
```

Deterministic event source

```
after :: Time -> b -> SF a (Event b)
```

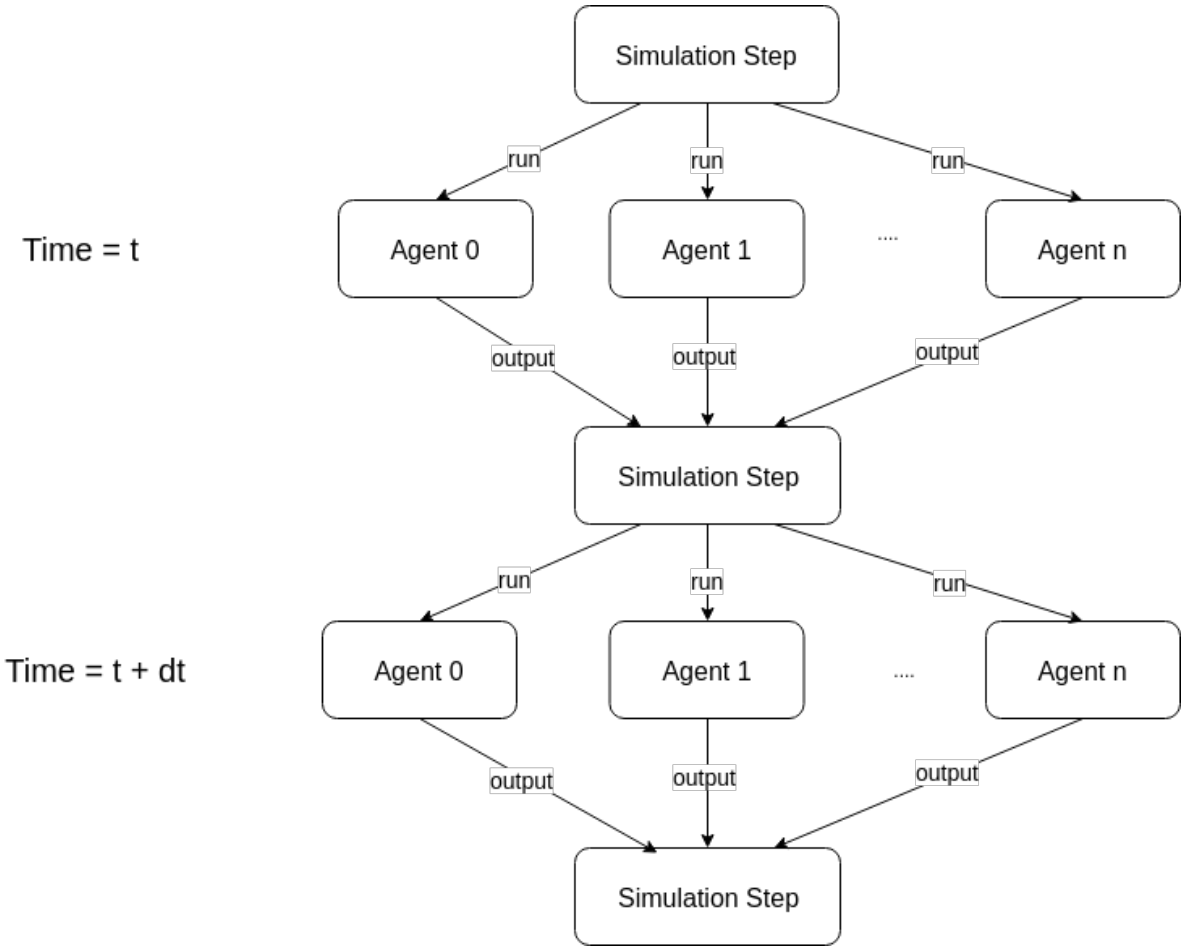
Random number stream

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

Infinitesimal delay (1 step)

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

Update Semantics



Arrowized Programming

Monads

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

Arrows

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

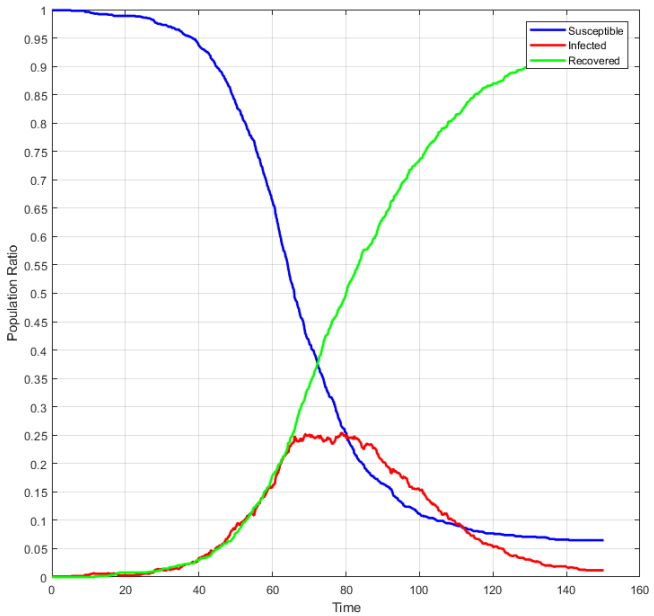
Some Types...

```haskell
data SIRState = Susceptible | Infected | Recovered

type SIRAgent = SF [SIRState] SIRState

sirAgent :: RandomGen g => g -> SIRState -> SIRAgent
sirAgent g Susceptible = susceptibleAgent g
sirAgent g Infected    = infectedAgent g
sirAgent _ Recovered   = recoveredAgent

recoveredAgent :: SIRAgent
recoveredAgent = arr (const Recovered)
```

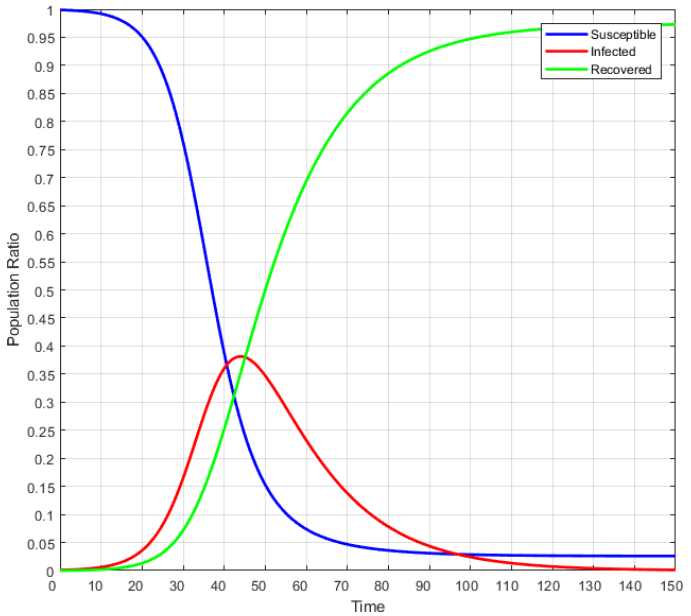
Susceptible Agent

```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
      -- generate an event on average with given rate
      makeContact <- occasionally g (1 / contactRate) () -< ()
      if isEvent makeContact
        then (do
          -- draw random contact
          a <- drawRandomElemSF g -< as
          case a of
            -- contact with infected => get infected with prob.
            Infected -> do
              -- returns True with given probability
              i <- randomBoolSF g infectivity -< ()
              if i
                -- got infected => infection event => transition to infected
                then returnA -< (Infected, Event ())
                else returnA -< (Susceptible, NoEvent)
            _        -> returnA -< (Susceptible, NoEvent))
        else returnA -< (Susceptible, NoEvent)
```

Dynamics $\Delta t = 0.1$

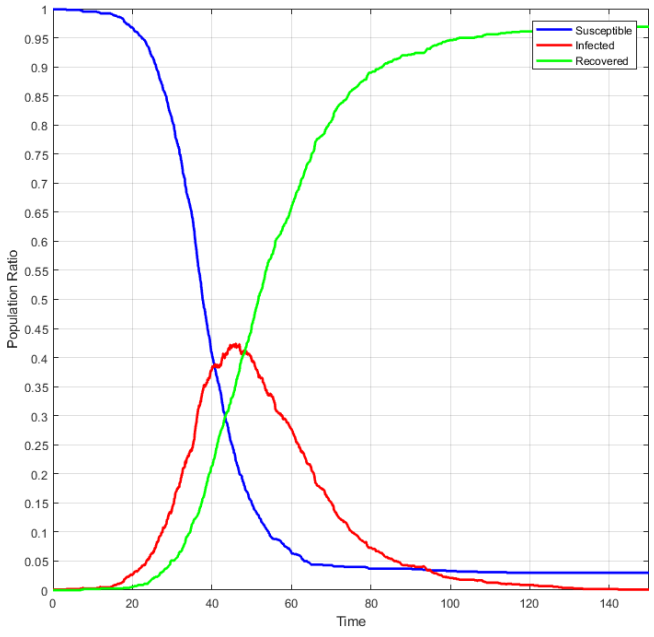


(a) Agent-Based approach

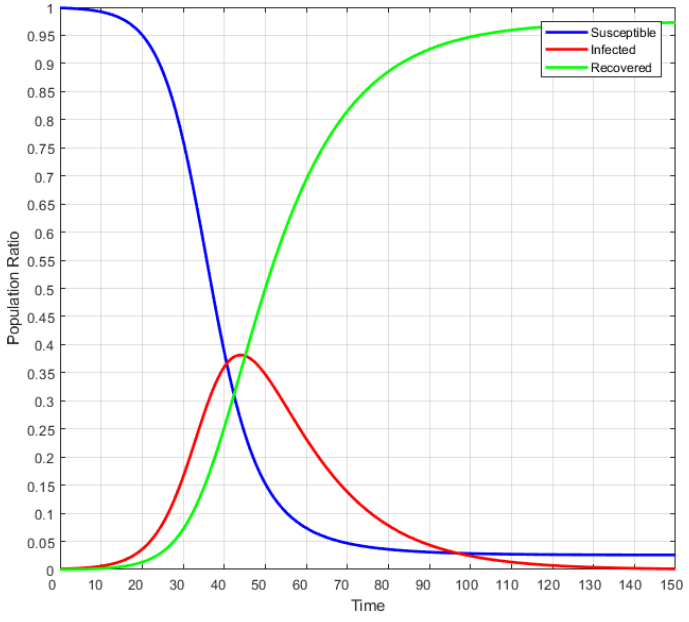


(b) System Dynamics

Dynamics $\Delta t = 0.01$



(a) Agent-Based approach



(b) System Dynamics

Reflection

So far...

- Agents with stochastic behaviour
- Time & Feedback
- Deterministic dynamics
- Parallel, lock-step semantics

Whats next?

- Problem - correlated random numbers
- Spatiality - structured environment

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
```

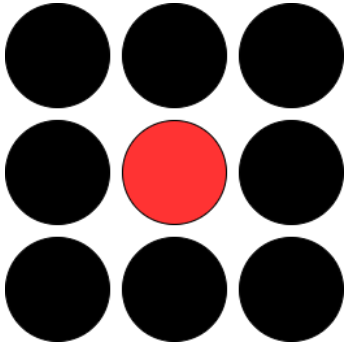
Monadic Stochastic Event Source

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

Monadic Agent Signal Function

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

Defining Spatiality

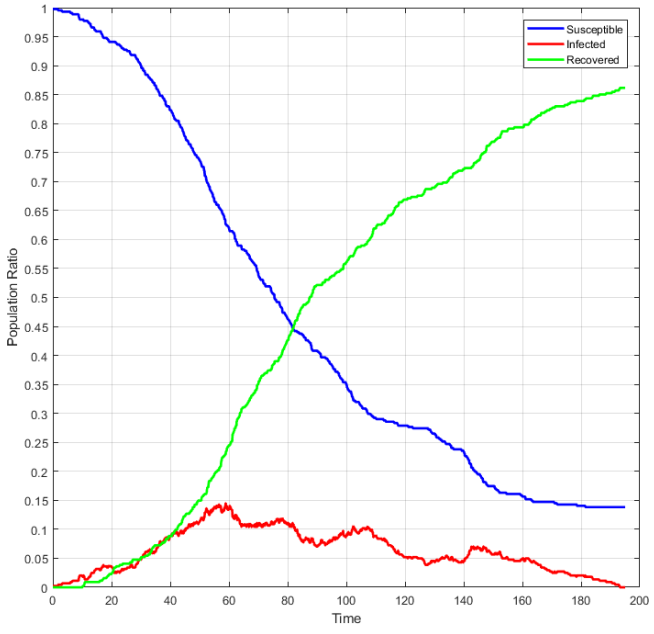


Moore Neighbourhood

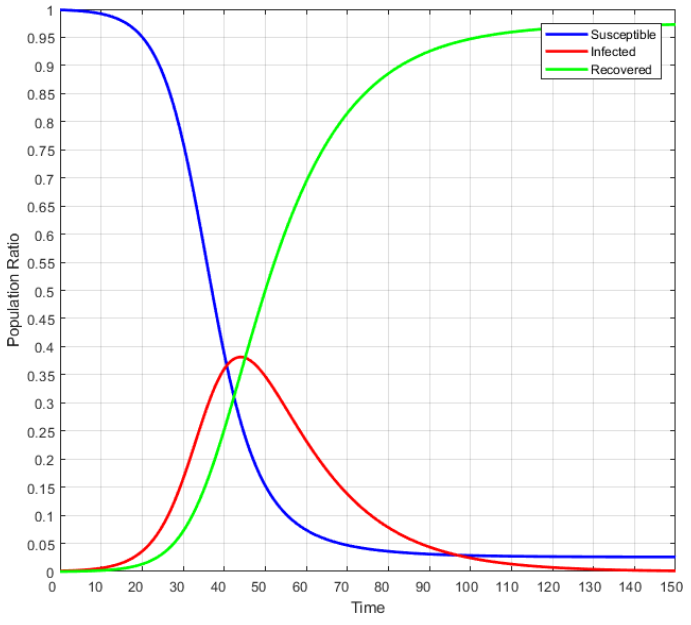
Some types...

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

Thank You!