Pure Functional Epidemics
An Agent-Based Approach

Jonathan Thaler Thorsten Altenkirch Peer-Olaf Siebers

University of Nottingham, United Kingdom

IFL 2018

Outline

Research Question(s)

- **How** can we implement Agent-Based Simulation (ABS) in (pure) functional programming?
- **Why** would we do that?
- **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 $\rightarrow$ Agents
2. Situated in City $\rightarrow$ Environment
3. Interacting with each other $\rightarrow$ local interactions
4. Creating dynamics $\rightarrow$ emergent system behaviour
5. Therefore ABS $\rightarrow$ bottom-up approach

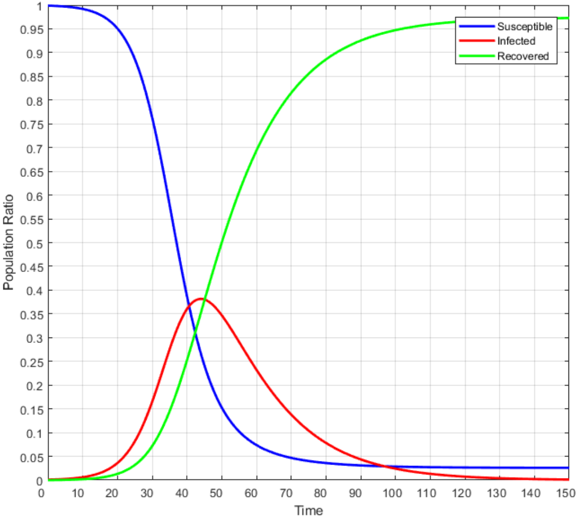
SIR Model

- 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 (on the rise!), Java,...

What we want is (pure) functional programming

- Reproducibility guaranteed at compile time
- Declarative, reasoning, parallelism, concurrency, property-based testing,...

How can we do it?

Functional Reactive Programming

Functional Reactive Programming (FRP)

- Implement continuous- & discrete-time systems in functional programming
- Central concept: Signal Function (SF), process over time, maps signal to signal
- Events (deterministic & stochastic)
- Random-number streams
- Running signal-functions in pure way
- *Arrowized* FRP using the *Yampa* library

First Steps

```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 (susceptible g) (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)
```

Infected Agent

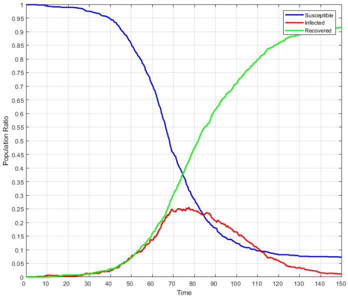
```
infectedAgent :: RandomGen g => g -> SIRAgent
infectedAgent g = switch infected (const recoveredAgent)
  where
    infected :: SF [SIRState] (SIRState, Event ())
    infected = proc _ -> do
      recEvt <- occasionally g illnessDuration () -< ()
      let a = event Infected (const Recovered) recEvt
      returnA -< (a, recEvt)
```

Running the Simulation

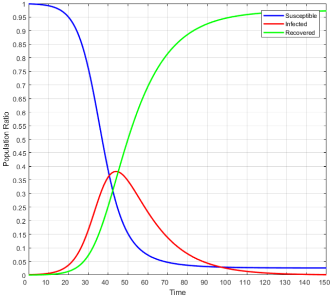
```haskell
runSimulation :: RandomGen g => g -> Time -> DTime -> [SIRState] -> [[SIRState]]
runSimulation g t dt as = embed (stepSimulation sfs as) ((), dts)
  where
    steps      = floor (t / dt)
    dts        = replicate steps (dt, Nothing)
    n          = length as
    (rngs, _)  = rngSplits g n [] -- unique rngs for each agent
    sfs        = zipWith sirAgent rngs as

stepSimulation :: [SIRAgent] -> [SIRState] -> SF () [SIRState]
stepSimulation sfs as =
    dpSwitch
      -- feeding the agent states to each SF
      (\_ sfs' -> (map (\sf -> (as, sf)) sfs'))
      -- the signal functions
      sfs
      -- switching event, ignored at t = 0
      (switchingEvt >>> notYet)
      -- recursively switch back into stepSimulation
      stepSimulation
  where
    switchingEvt :: SF ((), [SIRState]) (Event [SIRState])
    switchingEvt = arr (\ (_, newAs) -> Event newAs)
```

Dynamics $\Delta t = 0.1$



(a) Agent-Based approach



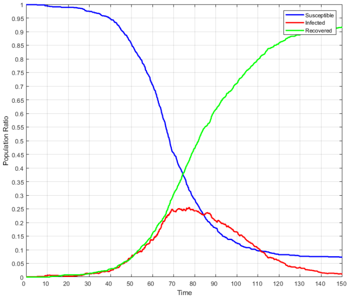
(b) System Dynamics

Sampling Issues

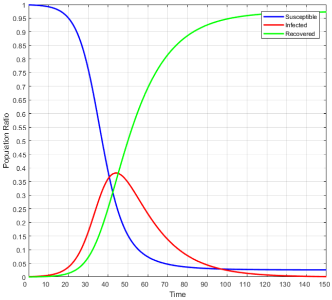
Undersampling

Supersampling

Dynamics $\Delta t = 0.1$

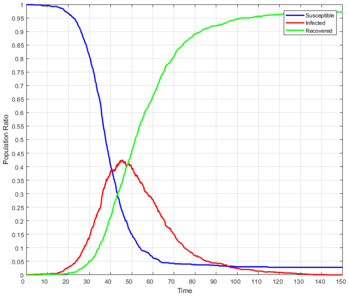


(a) Agent-Based approach

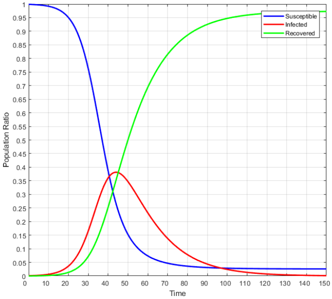


(b) System Dynamics

Dynamics $\Delta t = 0.01$



(a) Agent-Based approach



(b) System Dynamics

Reflection

- **Time** - simulation over virtual time, modelled explicitly, divided into *fixed* Δt, at each step all agents executed.
- **Agents** - implemented as an individual, behaviour depending on internal state.
- **Feedback** - output state of agent in time-step t is input for next time-step $t + \Delta t$.
- **Environment** - fully-connected network (complete graph).
- **Stochasticity** - inherently stochastic simulation.
- **Deterministic** - repeated runs with same initial random-number generator result in same dynamics.

Where is the benefit?

Same as System Dynamics

Where is the Environment?

Adding an Environment

- Running SFs happens in *'parallel'* => multiple copies of environment
- State Monad elegant solution

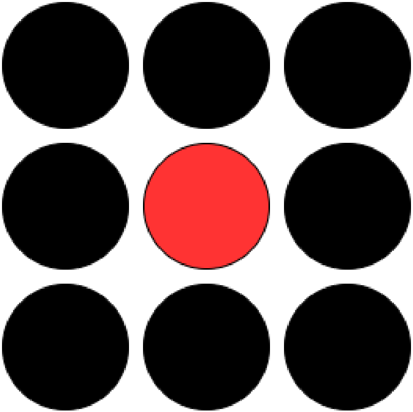
Problem

Yampa not monadic

Solution

Use Monadic Stream Functions (MSFs)
Conceptually, Signal Functions with monadic context

The environment: Moore neighbourhood



Re-defining Types

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

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

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

moore :: [Disc2dCoord]
moore = [ topLeftDelta,    topDelta,     topRightDelta,
          leftDelta,                     rightDelta,
          bottomLeftDelta, bottomDelta,  bottomRightDelta ]

topLeftDelta :: Disc2dCoord
topLeftDelta      = (-1, -1)

topDelta :: Disc2dCoord
topDelta          = ( 0, -1)
...
```

Susceptible Agent revisited

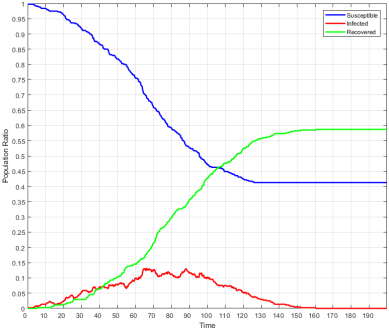
```haskell
susceptibleAgent :: RandomGen g => Disc2dCoord -> SIRAgent g
susceptibleAgent coord = switch susceptible (const (infectedAgent coord))
  where
    susceptible :: RandomGen g => SF (SIRMonad g) () ((), Event ())
    susceptible = proc _ -> do
      makeContact <- occasionallyM (1 / contactRate) () -< ()
      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))
```

Running the Simulation revisited

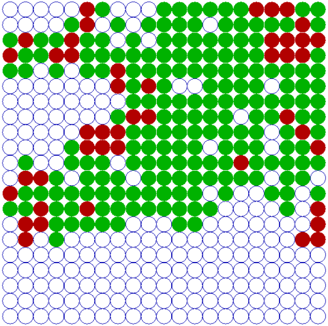
```haskell
runSimulation :: RandomGen g
              => g -> Time -> DTime -> SIREnv -> [(Disc2dCoord, SIRState)] -> [SIREnv]
runSimulation g t dt env as = evalRand esRand g
  where
    steps    = floor (t / dt)
    dts      = replicate steps ()
    -- initial SFs of all agents
    sfs      = map (uncurry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

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))
```

Dynamics with environment



(a) Dynamics over $t = 300$



(b) Visualisation at $t = 100$

Performance

Experiment

Spatial Agent-Based SIR, 51x51 Grid (2,601 agents), $t = 100$, $\Delta t = 0.1$, avg. 8 runs

Haskell

100.3 sec

RePast ($\Delta t = ?$)

10.8 sec

STM Haskell

8.6 sec on Amazon S2, 16 cores

Conclusion

- Purity guarantees reproducibility at compile time
- Performance :(
- Agent-Identity a bit lost
- Agent-Interaction is main difficulty

Future Work

Property-based testing for verification in ABS

- Completely untouched (property-based testing exists in Java, Python)
- Might offer huge benefits e.g. formalising hypotheses

Dependent Types in ABS (using Idris)

- Safe environment access
- Safe state-transitions
- Safe Agent-Interactions
- Equilibrium of Model & Totality of Implementation

Thank You!