Pure functional programming in Agent-Based Simulation

Jonathan Thaler

University of Nottingham, Nottingham, United Kingdom

Sandtable 21st June 2019

The Metaphor

- "[..] object-oriented programming is a particularly natural development environment for Sugarscape specifically and artificial societies generally [..]" (Epstein et al 1996)
- "agents map naturally to objects" (North et al 2007)

Outline

- Defining Agent-Based Simulation (ABS)
- What is *pure* Functional Programming (FP)?
- How can we do ABS + FP?
- ABS + FP = ?
- Erlang = Future of ABS?
- Conclusions

Defining Agent-Based Simulation (ABS)

What is Agent-Based Simulation?

Agent-Based Simulation is a methodology to **model** and **simulate** a system, where the **global behaviour** may be **unknown** but the behaviour and **interactions** of the **parts** making up the system **is known.** Those parts, called **agents**, are modelled and simulated, out of which then the aggregate **global behaviour** of the whole system **emerges**.

Defining ABS cont'd

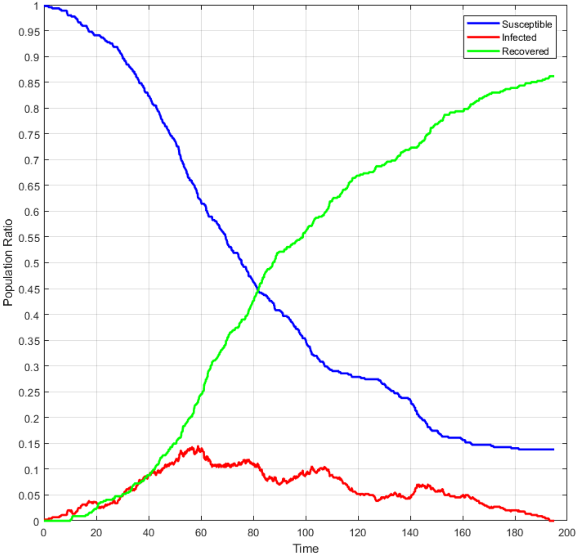
We informally assume the following about our agents [1, 2, 3, 4]:

1. They are **uniquely addressable** entities with some **internal state** over which they have full, **exclusive control**.
2. They are **pro-active**, which means they can initiate actions on their own e.g. change their internal state, send messages, create new agents, terminate themselves.
3. They are situated in an **environment** and can **interact** with it.
4. They can **interact** with **other agents** situated in the same environment by means of **messaging**.

Agent-Based Spatial SIR Model

- Population size $N = 1,000$
- Contact rate $\beta = 5$
- Infection probability $\gamma = 0.05$
- Illness duration $\delta = 15$
- 1 initially infected agent
- On a 2D grid with Moore Neighbourhood

Agent-Based Spatial SIR Model Dynamics



What is pure functional programming?

Functions as first class citizens

Passed as arguments, returned as values and assigned to variables.

```
map :: (a -> b) -> [a] -> [b]

const :: a -> (b -> a)
const a = (\_ -> a)
```

What is pure functional programming cont'd?

Immutable data

Variables can not change, functions return new copy.
Data-Flow oriented programming.

```
let x   = [1..10]
    x'  = drop 5 x
    x'' = x' ++ [10..20]
```

What is pure functional programming cont'd?

Recursion

To iterate over and change data.

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

What is pure functional programming cont'd?

Declarative style

Describe *what* to compute instead of *how*.

```
mean :: [Double] -> Double
mean xs = sum xs / length xs
```

What is pure functional programming cont'd?

Explicit about Side-Effects

Distinguish between side-effects of a function *in its type*.

```
readFromFile        :: String -> IO String
randomExponential   :: Double -> Rand Double
statefulAlgorithm   :: State Int (Maybe Double)
produceData         :: Writer [Double] ()
```

How can we do ABS + FP?

How can we represent an Agent, its local state and its interface?

We don't have objects and mutable state...

How can we implement direct agent-to-agent interactions?

We don't have method calls and mutable state...

How can we implement an environment and agent-to-environment interactions?

We don't have method calls and mutable state...

Solution

Functional Reactive Programming +
Monadic Stream Functions

Arrowized Functional Reactive Programming (AFRP)

- Continuous- & discrete-time systems in FP
- Signal Function
- Events
- Effects like random-numbers, global state, concurrency
- *Arrowized* FRP using the *Dunai* library

Monadic Stream Functions (MSF)

Process over time

$$\textit{SF}\ \alpha\ \beta \approx \textit{Signal}\ \alpha \rightarrow \textit{Signal}\ \beta$$
$$\textit{Signal}\ \alpha \approx \textit{Time} \rightarrow \alpha$$

Agents as Signal Functions

- Clean interface (input / output)
- Pro-activity by perceiving time
- Closures + Continuations = very simple immutable objects

What are closures and continuations?

```haskell
-- continuation type-definition
newtype Cont a = Cont (a -> (a, Cont a))

-- A continuation which sums up inputs.
-- It uses a closure to capture the input
adder :: Int -> Cont Int
adder x = Cont (\x' -> (x + x', adder (x + x')))
```

Recovered Agent

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

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

type SIRAgent    = SF Rand SIREnv SIRState

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

Infected Agent

```
infectedAgent :: Double -> SIRAgent
infectedAgent delta
    = switch infected (const recoveredAgent)
  where
    infected :: SF Rand SIREnv (SIRState, Event ())
    infected = proc _ -> do
      recovered <- occasionally delta () -< ()
      if isEvent recovered
        then returnA -< (Recovered, Event ())
        else returnA -< (Infected, NoEvent)
```

Susceptible Agent

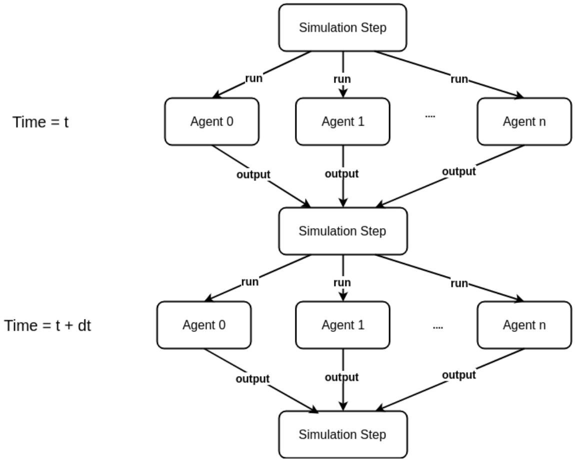
```
susceptibleAgent coord beta gamma delta
    = switch susceptible (const (infectedAgent delta))
  where
    susceptible :: SF Rand SIREnv (SIRState, Event ())
    susceptible = proc env -> do
      makeContact <- occasionally (1 / beta) () -< ()
      if isEvent makeContact
        then (do
          s <- randomNeighbour coord env -< as
          case s of
            Just Infected -> do
              i <- arrM_ (lift (randomBoolM gamma)) -< ()
              if i
                then returnA -< (Infected, Event ())
                else returnA -< (Susceptible, NoEvent)
            _        -> returnA -< (Susceptible, NoEvent))
        else returnA -< (Susceptible, NoEvent)
```

ABS + FP = Type Saftey

Purity guarantees reproducibility at compile time

"... when the sequence of random numbers is specified ex ante the model is deterministic. Stated yet another way, model output is invariant from run to run when all aspects of the model are kept constant including the stream of random numbers." Epstein et al (1996)

ABS + FP = Enforce Update Semantics



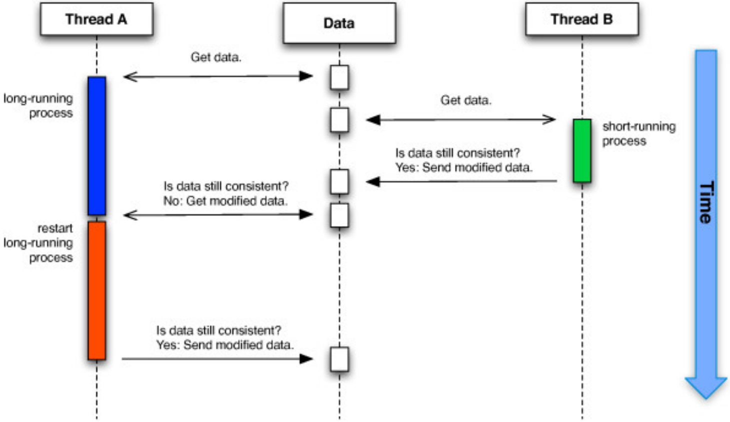
ABS + FP = Software Transactional Memory

- Concurrency in ABS difficult.
- Synchronisation using locks.
- $\Rightarrow$ error prone
- $\Rightarrow$ mixing of concurrency and model related code.
- New approach in Haskell: Software Transactional Memory.

Software Transactional Memory (STM)

- Lock free concurrency.
- Run STM actions concurrently and rollback / retry.
- Haskell first language to implement in core.
- Haskell type system guarantees retry-semantics.

Software Transactional Memory (STM)



Software Transactional Memory (STM)

- Tremendous performance improvement.
- Substantially outperforms lock-based implementation.
- STM semantics retain guarantees about non-determinism.

ABS + FP = Property-Based Testing

- Express specifications directly in code.
- QuickCheck library generates random test-cases.
- Developer can express expected coverage.
- Random Property-Based Testing + Stochastic ABS = ♡♡♡

QuickCheck

List Properties

```
-- the reverse of a reversed list is the original list
reverse_reverse :: [Int] -> Bool
reverse_reverse xs
  = reverse (reverse xs) == xs

-- concatenation operator (++) is associative
append_associative :: [Int] -> [Int] -> [Int] -> Bool
append_associative xs ys zs
  = (xs ++ ys) ++ zs == xs ++ (ys ++ zs)

-- reverse is distributive over concatenation (++)
reverse_distributive :: [Int] -> [Int] -> Bool
reverse_distributive xs ys
  = reverse (xs ++ ys) == reverse xs ++ reverse ys
```

QuickCheck cont'd

Running the tests...

```
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
*** Failed! Falsifiable (after 3 tests and 1 shrink):
[1]
[0]
```

QuickCheck cont'd

Labeling

```
reverse_reverse_label :: [Int] -> Property
reverse_reverse_label xs
  = label ("length of list is " ++ show (length xs))
          (reverse (reverse xs) == xs)
```

Running the tests...

```
+++ OK, passed 100 tests:
 5% length of list is 27
 5% length of list is 15
 5% length of list is 0
 4% length of list is 4
 4% length of list is 19
 ...
```

QuickCheck cont'd

Coverage

```
reverse_reverse_cover :: [Int] -> Property
reverse_reverse_cover xs  = checkCoverage
  cover 15 (length xs >= 50) "length of list at least 50"
  (reverse (reverse xs) == xs)
```

Running the tests...

```
+++ OK, passed 12800 tests
    (15.445% length of list at least 50).
```

Property-Based Testing Conclusion

- Test agent specification.
- Test simulation invariants.
- Validate dynamics against real world data.
- Exploratory models: hypotheses tests about dynamics.
- Explanatory models: validate against formal specification.

ABS + FP = Drawbacks!

- Direct bi-directional / sync Agent-interactions are *very* cumbersome.
- STM not applicable to direct agent-interactions.
- SF/MSFs with many effects are *terribly* slow!
- Steep learning curve: learning Haskell *is* hard.
- (Good) Haskell programmers are a *very* scarce resource.
- Strong, static type-system sometimes a burden as we want to be more dynamic.

Is (pure) functional ABS a dead end?

On the contrary, it is just the beginning... enter Erlang!

Erlang = Future of ABS?

- Functional language; dynamically strongly typed; *not* pure.
- Concurrent actor-based messaging with shared-nothing semantics.
- extreme robust, has proven itself in the industry (1.7 million lines of code in TODO switch of Eriksson, with downtime of 2 hours over 40 years)
- property-based testing available, can even detect races with quickcheck-statemachine
- emulate oo: encapsulation (obviously), polymorphism (same interface, different behaviour), basiclly like a immutable, single method object, which transitions into a new object upon reception of a message, sync method-call possible (send and receive)
- philosophy: fail fast, seems to work as proven in highly complex industry

Erlang + ABS

- maps very naturally to ABS which center around complex agent-interactions, and can exploit concurrency really well
- easy emulation of data-parallelism (not statically guaranteed)
- performance: substantially faster than pure functional Haskell approach, still slower than a sequential java approach BUT it should scale much better to large number of agents
- think about adistributed, always-online, distributed simulations which can be upgraded (e.g. introduce new agents) while the simulation is running
- Process calculi are directly applicable (CPS, CCS, pi-calculus) and provide a means to compensate for the lack of static typing
- First prototypes done by me (SIR, Sugarscape) are promising.

Conclusion

Have we done ABS implementation wrong?

No, but we missed out on a lot of potential!

I propose Erlang as the future of ABS implementation

But... who is going to take the risk?

Thank You!

MACAL, C. M.
Everything you need to know about agent-based modelling and simulation.
Journal of Simulation 10, 2 (May 2016), 144–156.

ODELL, J.
Objects and Agents Compared.
Journal of Object Technology 1, 1 (May 2002), 41–53.

SIEBERS, P.-O., AND AICKELIN, U.
Introduction to Multi-Agent Simulation.
arXiv:0803.3905 [cs] (Mar. 2008).
arXiv: 0803.3905.

WOOLDRIDGE, M.
An Introduction to MultiAgent Systems, 2nd ed.
Wiley Publishing, 2009.