Pure functional programming in Agent-Based Simulation

Jonathan Thaler

University of Nottingham, Ningbo, China

AIOP Seminar 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

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

What is 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 $\beta = 5$
- 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

Defining Spatiality

Moore Neighbourhood

Spatial Dynamics

Agent-Based

System Dynamics

Spatial Visualisation

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 with 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

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

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')))

-- Runs a continuation for n steps and
-- prints the output in each step
runCont :: Int -> Cont Int -> IO ()
runCont 0 _ = return ()
runCont n (Cont cont) = do
  let (x, cont') = cont 1
  print x
  runCont (n-1) cont'
```

Recovered Agent

```
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: agents acting at same time, interacting with each other and with environment.
- Synchronisation using locks: semaphors, monitors, mutex,...
- $\Rightarrow$ error prone (deadlocks, correctness, composability,...)
- $\Rightarrow$ mixing of concurrency and model related code.
- New approach in Haskell: Software Transactional Memory

Software Transactional Memory (STM)

- Lock free concurrency.
- Run STM actions optimistically 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
- Guarantees that the non-determinism comes only from concurrency within STM and nothing else

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]
```

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.

Conclusion

- The direction is towards an simulation which is more likely to be correct with the ultimate goal being a correct-by-construction implementation (up to some specification).
- A correct-by-construction implementation does NOT relief us from actually running the simulation!
-

Thank You!