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

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

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

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

Agents as Signal Functions

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

What are closures and continuations?

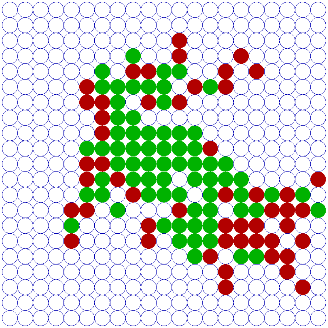
```
-- continuation type-definition
newtype Cont i o = Cont (i -> (o, Cont i o))

-- 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')))
```

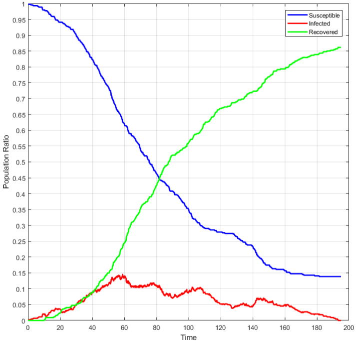
Agent-Based Spatial SIR Model

- 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



(a) Environment at $t = 50$



(b) Dynamics over time

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

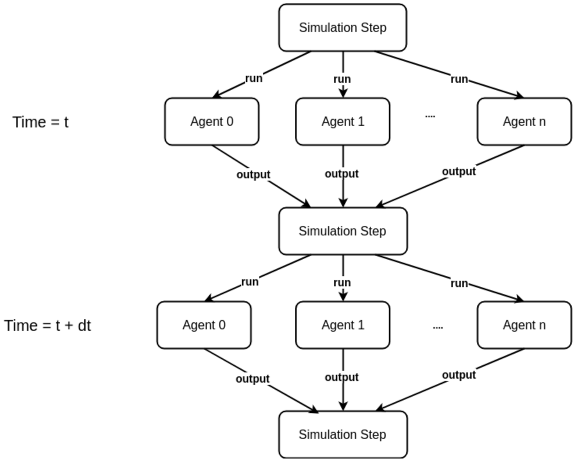
```
susceptibleAgent coord beta gamma delta
    = switch susceptible (const (infectedAgent delta))
  where
    susceptible :: SF Rand SIREnv (SIRState, Event ())
    susceptible = proc env -> do
      makeContact <- occasionally (1 / beta) () -< ()
      if isEvent makeContact
        then (do
          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



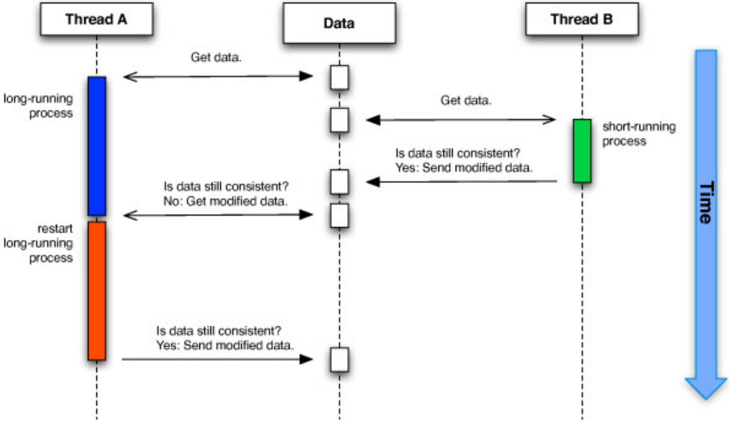
ABS + FP = Software Transactional Memory

- Concurrency in ABS difficult.
- 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.
- No pollution of code with locking semantics.Â´
- 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 ABS example: SIR invariants

```
prop_sir_invariants :: Positive Int       -- ^ contact rate
                    -> Probability        -- ^ infectivity (0,1)
                    -> Positive Double    -- ^ illness duration
                    -> TimeRange          -- ^ duration
                    -> [SIRState]         -- ^ population
                    -> Property
prop_sir_invariants
    (Positive cor) (P inf) (Positive ild) (T t) as
  = property (do
    -- total agent count
    let n = length ss
    -- run the SIR simulation with a new RNG
    ret <- genSimulationSIR ss cor inf ild t
    -- check invariants and return result
    return (sirInvariants n ret)
```

Property-Based Testing ABS example: SIR invariants

```haskell
sirInvariants :: Int                     -- ^ N total number of agents
              -> [(Time,(Int,Int,Int))] -- ^ output each step: (Time, (S,I,R))
              -> Bool
sirInvariants n aos = timeInc && aConst && susDec && recInc && infInv
  where
    (ts, sirs)  = unzip aos
    (ss, _, rs) = unzip3 sirs

    -- 1. time is monotonic increasing
    timeInc = allPairs (<=) ts
    -- 2. number of agents N stays constant in each step
    aConst = all agentCountInv sirs
    -- 3. number of susceptible S is monotonic decreasing
    susDec = allPairs (>=) ss
    -- 4. number of recovered R is monotonic increasing
    recInc = allPairs (<=) rs
    -- 5. number of infected I = N - (S + R)
    infInv = all infectedInv sirs

    agentCountInv :: (Int,Int,Int) -> Bool
    agentCountInv (s,i,r) = s + i + r == n

    infectedInv :: (Int,Int,Int) -> Bool
    infectedInv (s,i,r) = i == n - (s + r)

    allPairs :: (Ord a, Num a) => (a -> a -> Bool) -> [a] -> Bool
    allPairs f xs = all (uncurry f) (pairs xs)

    pairs :: [a] -> [(a,a)]
    pairs xs = zip xs (tail xs)
```

Property-Based Testing Conclusion

- Matching the constructive and exploratory nature of ABS.
- Test agent specification.
- Test simulation invariants.
- 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.
- MSFs can become *terribly* slow!
- Steep learning curve: learning Haskell *is* hard.
- (Good) Haskell programmers are a *very* scarce resource.
- Strong, static type-system burden, sometimes 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.
- Actor Model: message-based concurrency, shared-nothing semantics.
- Extremely robust and mature: 1.7 million lines of code in Eriksson telecom switch, 2 hours downtime in 40 years.
- Property-based testing: detect races and deadlocks.
- STM behaviour: can be emulated or use Erlangs Mnesia.
- Philosophy: fail fast!

Erlang + ABS

- Prototypes (SIR, Sugarscape) look promising.
- Performance is promising.
- Maps naturally to models with complex agent-interactions with the need to scale up.
- Easy emulation of data-parallelism.
- Use Process Calculi (CPS, CCS, pi-calculus) for specification and algebraic reasoning.

The Future?

Agent-interaction heavy model with huge populations of computationally expensive agents, needing a distributed always online approach, which can be updated while simulation is running (e.g. introduce new agents).

Conclusion

Have we done ABS implementation wrong?

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

I hypothesise that Erlang could be the future of ABS

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

Thank You!