Reproducible parallel simulation experiments via pure functional programming

Tom Warnke and Adelinde M. Uhrmacher
Institute for Visual and Analytic Computing
University of Rostock

Motivation

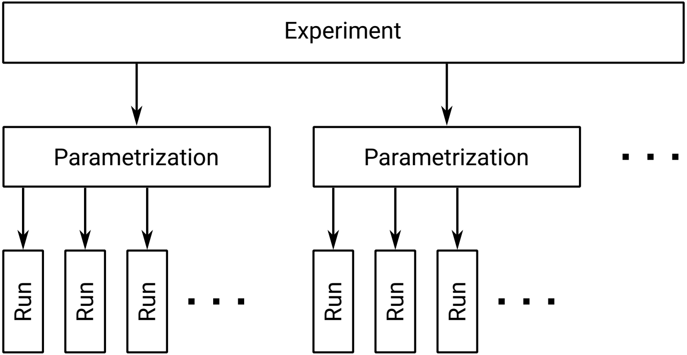
Simulation-based research suffers from a “reproducibility crisis”.

We propose to make results from simulation experiments reproducible by expressing them as pure functions.

A pure function

- is deterministic and
- has no side effects

Execution of a simulation experiment



Execution of a simulation experiment

Imperative implementation:

```
initRNG(seed)
for (p : parametrizations) {
  for (i : 1..runNumber) {
    s = randomInt()
    result[p,i] = run(p,s)
  }
}
```

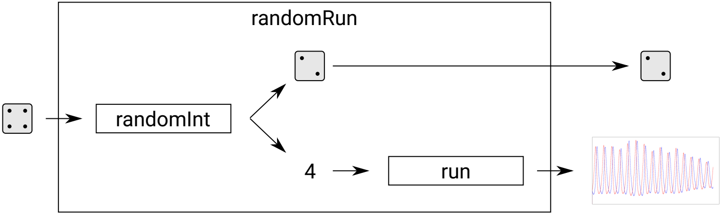
Purely functional implementation:

```
rng = initRNG(seed)
e = parametrizations.traverse { p =>
  randomInt.map { s =>
    run(p,s)
  }.replicateA(runNumber)
}
result = e.run(rng)
```

- implicit vs. explicit RNG state
- when parallelized, can determinism be affected by race conditions?

Sequential execution

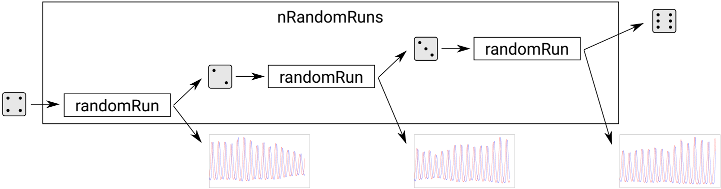
Expressing a single simulation run



```
randomRun = randomInt.map(s => run(s))
randomRun : RNG => (RNG, Result)
```

Sequential execution

Combining runs

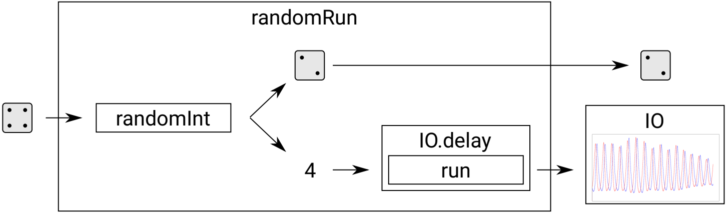


```
nRandomRuns = randomRun.replicateA(3)
nRandomRuns : RNG => (RNG, List[Result])
```

Concurrent execution

Expressing a single simulation run

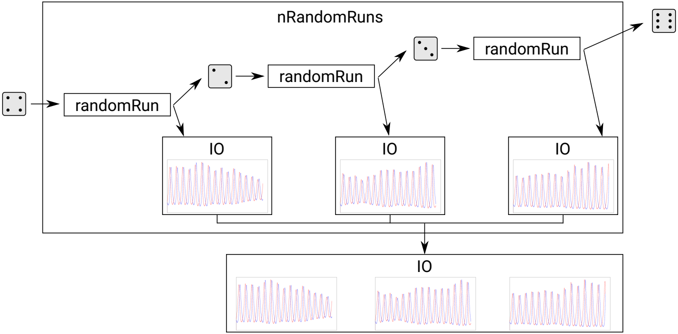
The execution of the run is suspended in an asynchronous effect monad `IO`.



```
randomRun = randomInt.map(s => IO.delay(run(s)))
randomRun : RNG => (RNG, IO[Result])
```

Concurrent execution

Combining runs

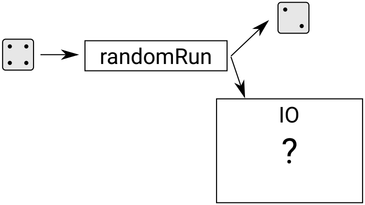


```
nRandomRuns = randomRun.replicateA(3).map(_.parSequence)
nRandomRuns : RNG => (RNG, IO[List[Result]])
```

Types and complex experiments

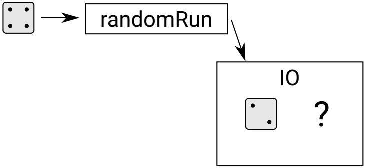
Interaction of concurrency and random number generation

`RNG => (RNG, IO[?])`



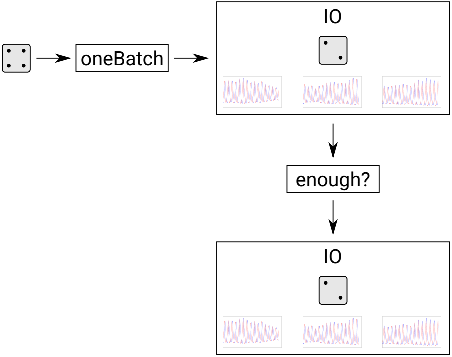
- parallel execution
- need to know how many RNs are needed in the beginning

`RNG => IO[(RNG, ?)]`

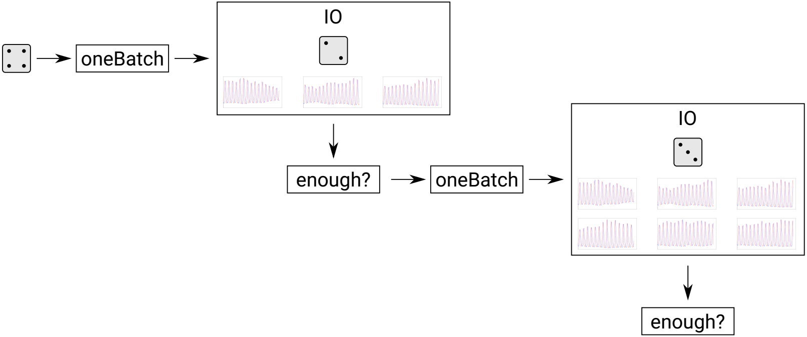


- sequential execution
- can decide to draw new RNs based on intermediate results

Dynamic replication conditions



Dynamic replication conditions



Example: Statistical model checking with NetLogo

Sequential probability ratio test

```
SPRT.check(
  run(randomInt)(s =>
    NetLogo.run(model       = ExampleModel.contents,
                stopCond    = nlBoolean("ticks > 50"),
                observables = List(obs),
                params      = Map("acceleration" -> 0.01,
                                  "deceleration" -> 0.01),
                seed        = s)
  ),
  batchSize = 4,
  property  = redCarNeverStops,
  p         = 0.8,
  alpha     = 0.05,
  beta      = 0.05,
  delta     = 0.05
)
```

Conclusion

Pure functional programming is one elegant way to express deterministic, parallel simulation experiments.

- It guarantees determinism by design.
- Diverse types of simulation experiments can be implemented.
- Supported by established FP libraries.