Comp 311
Functional Programming

Eric Allen, Two Sigma Investments
Robert “Corky” Cartwright, Rice University
Sagnak Tasirlar, Two Sigma Investments
Review:
Pseudo-Random Number Generation

- There are many approaches to generating a pseudo-random stream of \texttt{Int} values

- One common approach is to define a linear congruential generator (LCG):

  \[
  X_{n+1} = (aX_n + c) \mod m
  \]

- The pseudo-random numbers are the elements of this recurrence
Linear Congruential Generators

- LCGs can produce generators capable of passing formal tests for randomness
- The quality of the results is highly dependent on the initial values selected
- Poor statistical properties
- Not well suited for cryptographic purposes
A Linear Congruent Generator (C++11 minstd_rand)

def makeRandomGenerator(): () => Int = {
    val a = 48271
    val b = 0
    val m = Int.MaxValue
    var seed = 3

    def inner() = {
        seed = (a*seed + b) % m
        seed
    }
    inner
}

A Linear Congruent Generator
(C++11 minstd_rand)

val g = makeRandomGenerator()<E> ↦
val g =
< def inner() = {
    seed = (a*seed + b) % m
    seed

} ,
val a = 48271
val b = 0
val m = Int.MaxValue
var seed = 3 >
```scala
\( g(E) \mapsto \)
< def inner() = {
  seed = (a*seed + b) \mod m
  seed
}
val a = 48271
val b = 0
val m = Int.MaxValue
var seed = 3
\( >() \mapsto \)
```
seed = (a*seed + b) % m
seed,
< val a = 48271
  val b = 0
  val m = Int.MaxValue
  var seed = 3 >
⇒
seed = (48271*2 + 0) % Int.MaxValue
seed,
< val a = 48271
  val b = 0
  val m = Int.MaxValue
  var seed = 3 >
⇒
seed, <val a = 48271
val b = 0
val m = Int.MaxValue
var seed = 96542>

⇒

96542
And now the environment closing over generator $g$ binds seed to 96542.
Purely Functional State
Rolling a Die

• Suppose we want to implement a function that simulates the rolling of a six-sided die

• The result of calling the function should be a random number from 1 to 6
def rollDie: Int = {
  val rng = new scala.util.Random
  rng.nextInt(6) + 1
}

The call to nextInt will return a value from 0 to 5, not 1 to 6.
Stateful Programs and Debugging

• Because of the state encapsulated in our random number generator:
  • Repeatability of testing is hard
  • Bugs are difficult to reduce

• We would like to use effects when necessary without losing the benefits of referential transparency
trait RandomNumberGenerator {
  def nextInt: (Int, RandomNumberGenerator)
}

Purely Functional Random Number Generation
case class SimpleRNG(seed: Int) extends RandomNumberGenerator {
  val a = 48271
  val b = 0
  val m = Int.MaxValue

  def nextInt: (Int, RandomNumberGenerator) = {
    val newSeed = (a*seed + b) % m
    val newRNG = SimpleRNG(newSeed)
    (newSeed, newRNG)
  }
}
Threading State Through a Sequence of Statements

```scala
val rng = SimpleRNG(3)
val (n, rng2) = rng.nextInt(n + 1, rng2)
```
Transforming Stateful APIs to Functional APIs

trait Foo {
  private var s: State = MyState
  def bar: Bar
  def baz: Int
}

becomes

trait Foo {
  def bar: (Bar, Foo)
  def baz: (Int, Foo)
}
A Better API for State Actions

• Explicitly threading state from one function application to the next is tedious and error prone

• We would like to define combinators that pass the state from one application to the next automatically

• For now, we consider the state of our program to be defined entirely by the state of our random number generator
A Dream

val rng = SimpleRNG(3)

veryHelpfulFunction (  
  val n = rng.nextInt,  
  n + 1
)


A Dream

val rng = SimpleRNG(3)

veryHelpfulFunction {
    val n = rng.nextInt,
    n + 1
}
A Dream

val rng = SimpleRNG(3)

veryHelpfulFunction {
    val n = rng.nextInt,
        n + 1
} ↦ (4, rng1)
val rng = SimpleRNG(3)

veryHelpfulFunction {
  rng.nextInt,
  (n: Int) => n + 1
}

→
(4, rng1)
A More Realistic Dream

```scala
val rng = SimpleRNG(3)

def run() = veryHelpfulFunction {
  _.nextInt,
  (n: Int) => n + 1
}

run(rng) ↦ (4, rng1)
```
Defining a Type Alias for State Actions

```typescript
type StateAction[+A] = RandomNumberGenerator => (A, RandomNumberGenerator)
```
A Simple State Action

val nextInt: StateAction[Int] = _.nextInt
Transforming State Actions
With the Map Combinator

def veryUsefulFunction[A, B](action: StateAction[A],
               f: A => B): StateAction[B] =

          state => {
            val (a, state2) = action(state)
            (f(a), state2)
          }
Transforming State Actions
With the Map Combinator

def map[A,B](action: StateAction[A],
             f: A => B): StateAction[B] =
state => {
    val (a, state2) = action(state)
    (f(a), state2)
}
Transforming State Actions
With the Map Combinator

case class StateAction[S,+A](run: S => (A,S)) extends Function1[S,(A,S)] {
  def apply(s:S) = run(s)

    val (a, s2) = run(s)
    (f(a), s2)
  }
}
Reformulating `nextInt` as a State Action

```scala
val nextInt = StateAction {
  (rng: RandomNumberGenerator) => rng.nextInt
}
```
A Simple State Action

```scala
val nextInt = StateAction(_.nextInt)
```
A More Realistic Dream

```scala
val rng = SimpleRNG(6)

def run() = {
    nextInt.map
        ((n: Int) => n + 1)
}
```
A More Realistic Dream

```scala
val rng = SimpleRNG(6)

def run() = {
    for {
        n <- nextInt
        yield n + 1
    }
}
```
A More Realistic Dream

```scala
val rng = SimpleRNG(6)

def run() = {
  for {
    n <-nextInt
  }
  yield n + 1
}

run()(rng)
```
def nonNegativeInt = {
    for {
        n <- nextInt
    } yield if (n == Int.MinValue) 0 else if (n < 0) -n else n
}
Using Map

def nonNegativeEven: StateAction[Int] = for {
    i <- nonNegativeInt
} yield i - (i % 2)
Random Non-Negative Numbers in a Range (Attempt 1)

// INCORRECT
def nonNegativeLessThan(n: Int): StateAction[Int] =
  for {
    i <- nonNegativeInt
  } yield i % n

This definition skews the results because Int.MaxValue might not be divisible by n.
Random Non-Negative Numbers in a Range (Attempt 2)

// INCORRECT
val nonNegativeLessThan(n: Int): StateAction[Int] = {
for {
  i <- nonNegativeInt
} yield {
  val mod = i % n
  if (i + (n - 1) - mod >= 0) mod
  else nonNegativeLessThan(n)
}

But this version does not pass type checking!
Random Non-Negative Numbers in a Range (Attempt 2)

- The problem with our Attempt 2 is that the recursive call to `nonNegativeLessThan` produces a `StateAction[Int]`

- Our map combinator expects an `Int` result from the mapped function, not a `StateAction[Int]`

- To get a better idea as to how to define `nonNegativeLessThan`, let us try defining it without combinators
Random Non-Negative Numbers in a Range
(Attempt 3)

def nonNegativeLessThan(n: Int): StateAction[Int] = {
  rng =>
  val (i, rng2) = nonNegativeInt(rng)
  val mod = i % n
  if (i + (n - 1) - mod >= 0) (mod, rng2)
  else nonNegativeLessThan(n)(rng2)
}

This version works, but now we are back to threading state explicitly.

We need a new combinator.
case class StateAction[S,+A](run: S => (A,S)) extends Function1[S,(A,S)] {
    def apply(s:S) = run(s)

    def map[B](f: A => B): StateAction[S,B] = StateAction { s =>
        val (a, s2) = run(s)
        (f(a), s2)
    }

    def flatMap[B](f: A => StateAction[S,B]): StateAction[S,B] = StateAction { s =>
        val (a, s2) = run(s)
        f(a)(s2)
    }
}
Every Partial Application of the StateAction Type Defines a Monad

type RNGStateAction[A] = StateAction[RandomNumberGenerator, A]
Random Non-Negative Numbers in a Range (Attempt 4)

```scala
def nonNegativeLessThan(n: Int): StateAction[Int] = {
  nonNegativeInt.flatMap { i =>
    val mod = i % n
    if (i + (n - 1) - mod >= 0) (mod, _)
    else nonNegativeLessThan(n)
  }
}
```

We have almost completely eliminated state threading, except for one underscore.
Random Non-Negative Numbers in a Range (Attempt 4)

• We now have the inverse of our earlier problem:

• Our flatMap combinator expects a `StateAction[Int]` result from the mapped function, not an Int

• We can address this problem by wrapping part of the flatMapped function in an application of the unit constructor for `StateActions`
A “No-Op” Abstraction Over State Actions

def unit[A](a: A): StateAction[A] = 
  rng => (a, rng)

def rngUnit[A](a: A): RngStateAction[A] = 
  StateAction(s => (a, s))
Random Non-Negative Numbers in a Range (Attempt 5)

def nonNegativeLessThan4point5(n: Int):
    StateAction[RandomNumberGenerator,Int] = {
        nonNegativeInt.flatMap { i =>
            val result = i % n
            if (i + (n - 1) - result >= 0) unit(result)
            else nonNegativeLessThan5(n)
        }
    }
Random Non-Negative Numbers in a Range (Attempt 5)

def nonNegativeLessThan4point5(n: Int):
    StateAction[RandomNumberGenerator,Int] = {
        nonNegativeInt.flatMap { i =>
            val result = i % n
            if (i + (n - 1) - result >= 0) unit(result)
            else nonNegativeLessThan5(n)
        } map (j => j)
    }

A trailing map of the identity function defines an equivalent function.
Using For-Expression Syntax

def nonNegativeLessThan(n: Int): RngStateAction[Int] = {
  for {
    i <- nonNegativeInt
    result <- {
      val randN = i % n
      if (i + (n - 1) - randN >= 0) unit(randN)
      else nonNegativeLessThan(n)
    }
  }
  yield result
}
case class StateAction[S,+A](run: S => (A,S)) extends Function1[S,(A,S)] {
  def apply(s:S) = run(s)

  def map[B](f: A => B): StateAction[S,B] = StateAction { s =>
    val (a, s2) = run(s)
    (f(a), s2)
  }

  def flatMap[B](f: A => StateAction[S,B]): StateAction[S,B] = StateAction { s =>
    val (a, s2) = run(s)
    f(a)(s2)
  }
}
Revisiting RollDie

def rollDie: StateAction[Int] = nonNegativeLessThan(6)
Revisiting RollDie

def rollDie: StateAction[Int] =
    map(nonNegativeLessThan(6))( _ + 1)
Revisiting RollDie

def rollDie =
    for {
        i <- nonNegativeLessThan(6)
    }
    yield (i + 1)