Comp 311
Functional Programming

Eric Allen, PhD
Vice President, Engineering
Two Sigma Investments, LLC

Changing the State of Variables

Changing the State of Variables

- Thus far, we have focused solely on purely functional programs
- This approach has gotten us remarkably far
- Sometimes, it is difficult to structure a program without some notion of stateful variables:
  - I/O, GUIs
  - Modeling a stateful system in the world

Assignment and Local State

- We view the world as consisting of objects with state that changes over time
- It is often natural to model physical systems with computational objects with state that changes over time

Assignment and Local State

- If we choose to model the flow of time in the system by elapsed time in the computation, we need a way to change the state of objects as a program runs

- If we choose to model state using symbolic names in our program, we need an assignment operator to allow for changing the value associated with a name

Modeling an Address Book

```
class AddressBook() {
  val addresses: Map[String,String] = Map()

  def put(name: String, address: String) = {
    …
  }

  def lookup(name: String) = addresses(name)
}
```

Modeling an Address Book

```
class AddressBook() {
  var addresses: Map[String,String] = Map()

  def put(name: String, address: String) = {
    addresses = addresses + (name -> address)
  }

  def lookup(name: String) = addresses(name)
}
```

Sameness and Change

- In the context of assignment, our notion of equality becomes far more complex

```
val petersAddressBook = new AddressBook()
val paulsAddressBook = new AddressBook()
```

```
val petersAddressBook = new AddressBook()
val paulsAddressBook = paulsAddressBook
```

Sameness and Change

- Effectively assignment forces us to view names as referring not to values, but to *places* that store values

Referential Transparency

- The notion that equals can be substituted for equals in an expression without changing the value of the expression is known as *referential transparency*
- Referential transparency is one of the distinguishing aspects of functional programming
- It is lost as soon as we introduce assignment

Referential Transparency

- Without referential transparency, the notion of what it means for two objects to be “the same” is far more difficult to explain
- One approach:
  - Modify one object and see whether the other object has changed in the same way

Referential Transparency

- One approach:
  - Modify one object and see whether the other object has changed in the same way
  - But that involves observing a single object twice
  - How do we know we are observing the same object both times?

Pitfalls of Imperative Programming

- The order of updates to variables is a classic source of bugs

```scala
def factorial(n: Int) = {
  var product = 1
  var counter = 1
  def iter(): Int = {
    if (counter > n) {
      product
    }
    else {
      product = product * counter
      counter = counter + 1
      iter()
    }
  }
  iter()
}
```

```
def factorial(n: Int) = {
  var product = 1
  var counter = 1
  def iter(): Int = {
    if (counter > n) {
      product
    }
    else {
      product = product * counter
      counter = counter + 1
      iter()
    }
  }
  iter()
}
```

What if the order of these updates were reversed?

Review: The Environment Model of Evaluation

- Environments map names to values
- Every expression is evaluated in the context of an environment

The Environment Model of Reduction

- To evaluate a name, simply reduce to the value it is mapped to in the environment

The Environment Model of Reduction

- To evaluate a function, reduce it to a *closure,* which consists of two parts:
    - The body of the function
    - The environment in which the body occurs

The Environment Model of Reduction

- Objects are also modeled as closures
  - What is the environment?
  - What corresponds to the body of the function?

The Environment Model of Reduction

- To evaluate an application of a closure
  - Extend the environment of the closure, mapping the function’s parameters to argument values
  - Evaluate the body of the closure in this new environment

Variable Rebinding in the Environment Model

- The environment model provides us with the necessary machinery to model stateful variables
- To evaluate a variable *v* assignment:
  - Rebind the value *v* maps to in the environment in which the assignment occurs

Rebinding a Variable in an Environment

- The rebound value of *v* is then used in all subsequent reductions involving the same environment
  - Includes closures involving that environment
- This model of variable assignment pushes the notion of state out to environments
- The “places” referred to by variables are simply components of environments

Example: Pseudo-Random Number Generation

- There are many approaches to generating a pseudo-random stream of `Int` values

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

$$X_{n+1} = (aX_n + c) \bmod m$$

- 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 >
```

```
g()<E> ↦
< def inner() = {
        seed = (a*seed + b) % m
        seed
  } ,
  val a = 48271
  val b = 0
  val m = Int.MaxValue
  var seed = 3 >()<E> ↦
```

```
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
```

```
seed, <val a = 48271
       val b = 0
       val m = Int.MaxValue
       var seed = 96542>
```

$\mapsto$

96542

And now the environment closing over generator g binds seed to 96542.

Mutable Data Structures

Mutable Data Structures

- Thus far, we have explored only *variable* assignment
- It is often preferable to construct data structures with state that changes over time

Modeling an Address Book

```
class AddressBook() {
  var addresses: Map[String,String] = Map()

  def put(name: String, address: String) = {
    addresses = addresses + (name -> address)
  }

  def lookup(name: String) = addresses(name)
}
```

It would be nice to simply use a put operation to insert data into an existing map.

Mutable Data Structures

- We already know how to build mutable data structures:
  - Define classes with local variables
  - Note that our AddressBooks are themselves mutable data, given the `var` modifier on the `addresses` field
- Consequently, the environment model is all that is needed to model not only variable assignment, but arbitrary mutable data

Equality in Scala

Equality in Scala

- The method **eq** checks that two objects exist in the same place

Equality in Scala

- The method == checks the “natural” equality relation on a type

```
final def ==(that: Any): Boolean =
  if (null eq this) null eq that
  else this equals that
```

Equality in Scala

- The inherited `equals` method is the same as `eq`
- We can override the inherited definition
- Case classes override automatically

Pitfalls in Overriding Equals

- Wrong signature
- Not defining an equivalence relation
- Overriding on mutable datatypes
- Not overriding `hashCode`

Wrong Signature

```
def equals(that: Any): Boolean
```

Not Defining an Equivalence Relation

- Equivalence relations are:
  - Reflexive
  - Symmetric
  - Transitive
- To respect symmetry, we are forced to check that the *dynamic types* of two objects are identical

Not Defining an Equivalence Relation

```
class Point(val x: Int, val y: Int) {
  override def equals(that: Any): Boolean = …
}

class ColoredPoint(red: Int, blue: Int, green: Int, x: Int, y: Int)
extends Point(x,y)
```

Not Defining an Equivalence Relation

```
class Point(val x: Int, val y: Int) {
  override def equals(that: Any): Boolean = {
    if (this.getClass != that.getClass) false
    else {
      val _point = that.asInstanceOf[Point]
      (_point.x == x) && (_point.y == y)
    }
  }
}

class ColoredPoint(red: Int, blue: Int, green: Int, x: Int, y: Int)
extends Point(x,y)
```

Overriding on Mutable Datatypes

Just say no.

Memoization

Fibonacci Numbers

```
def fib(n: Int): Int = {
  require (n >= 0)
  if (n == 0) 0
  else if (n == 1) 1
  else fib(n - 1) + fib(n - 2)
} ensuring (_ >= 0)
```

Fibonacci Numbers

```
val memoFib: Int => Int =
  memoize {
    (n: Int) => {
      require (n >= 0)
      if (n == 0) 0
      else if (n == 1) 1
      else memoFib(n - 1) + memoFib(n - 2)
    } ensuring (_ >= 0)
  }
```

Memoize

```
def memoize(f: Int => Int) = {
  val table = mutable.Map[Int,Int]()
  (n: Int) =>
    table.getOrElse(n, {
      val result = f(n)
      table += (n -> result)
      result
    })
}
```

Impact of Effects on the Design Recipe

Impact of Effects on the Design Recipe

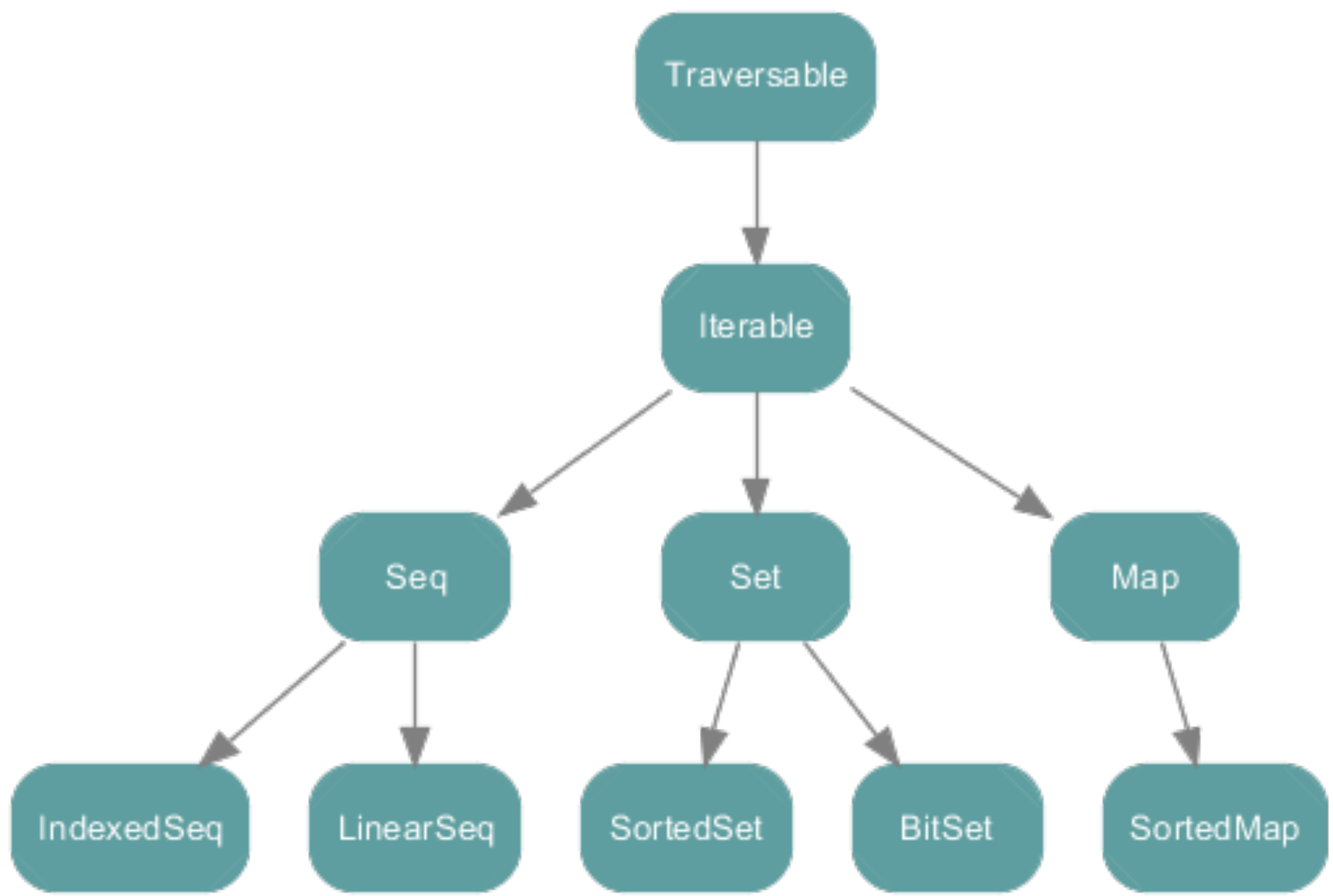
- Now that functions have effects:
  - The documentation should discuss the observable effects
  - Examples should include observable effects
  - Tests should check that effects occur as expected

Testing Effects

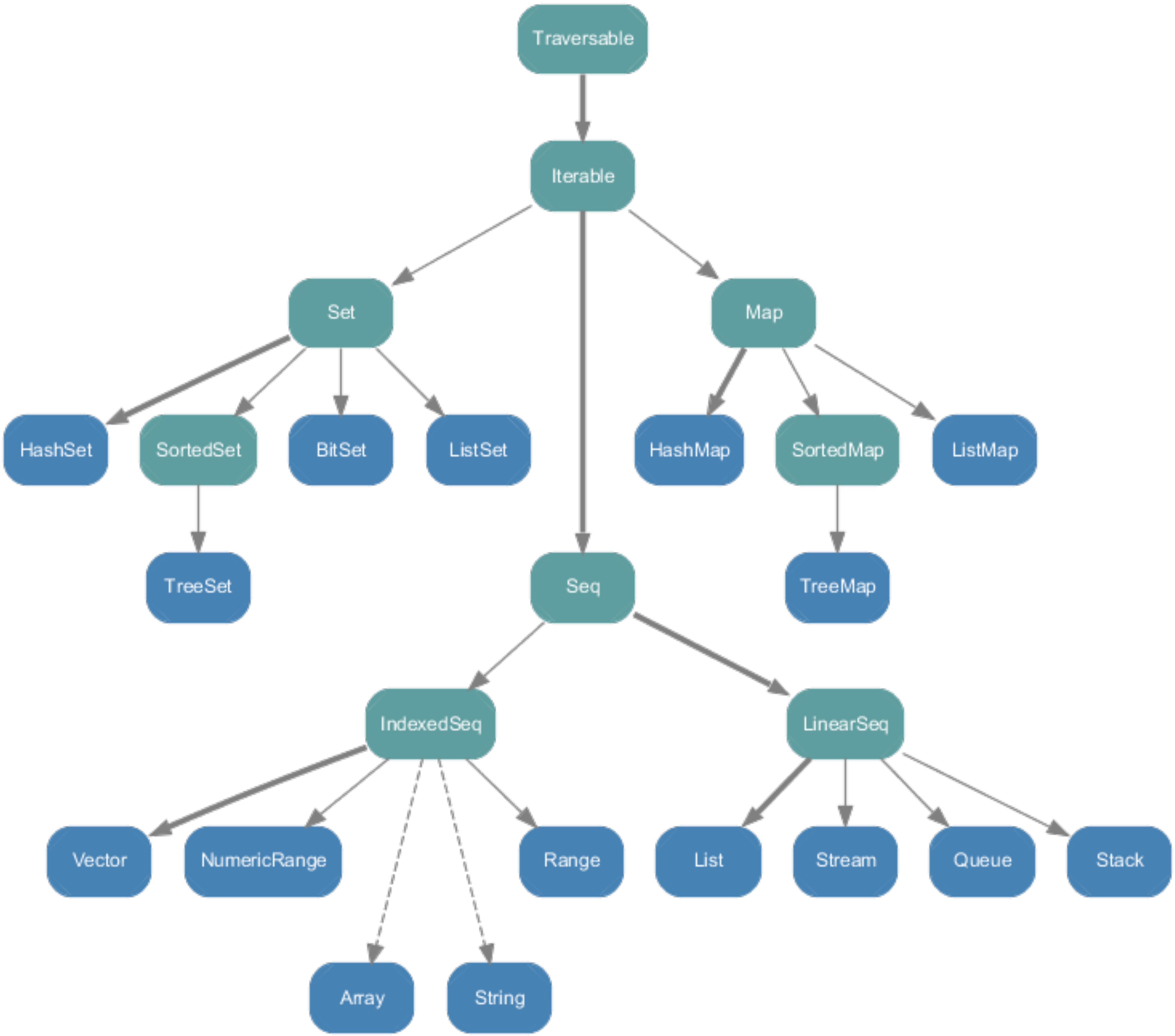
- A common approach to testing in the context of effects is *mocking:*
  - The external objects and APIs our tested code interfaces with is implemented as mock objects that behave just well enough to enable the test
  - Typically, mock objects should perform contained and reversible actions!

Scala
Collections Classes

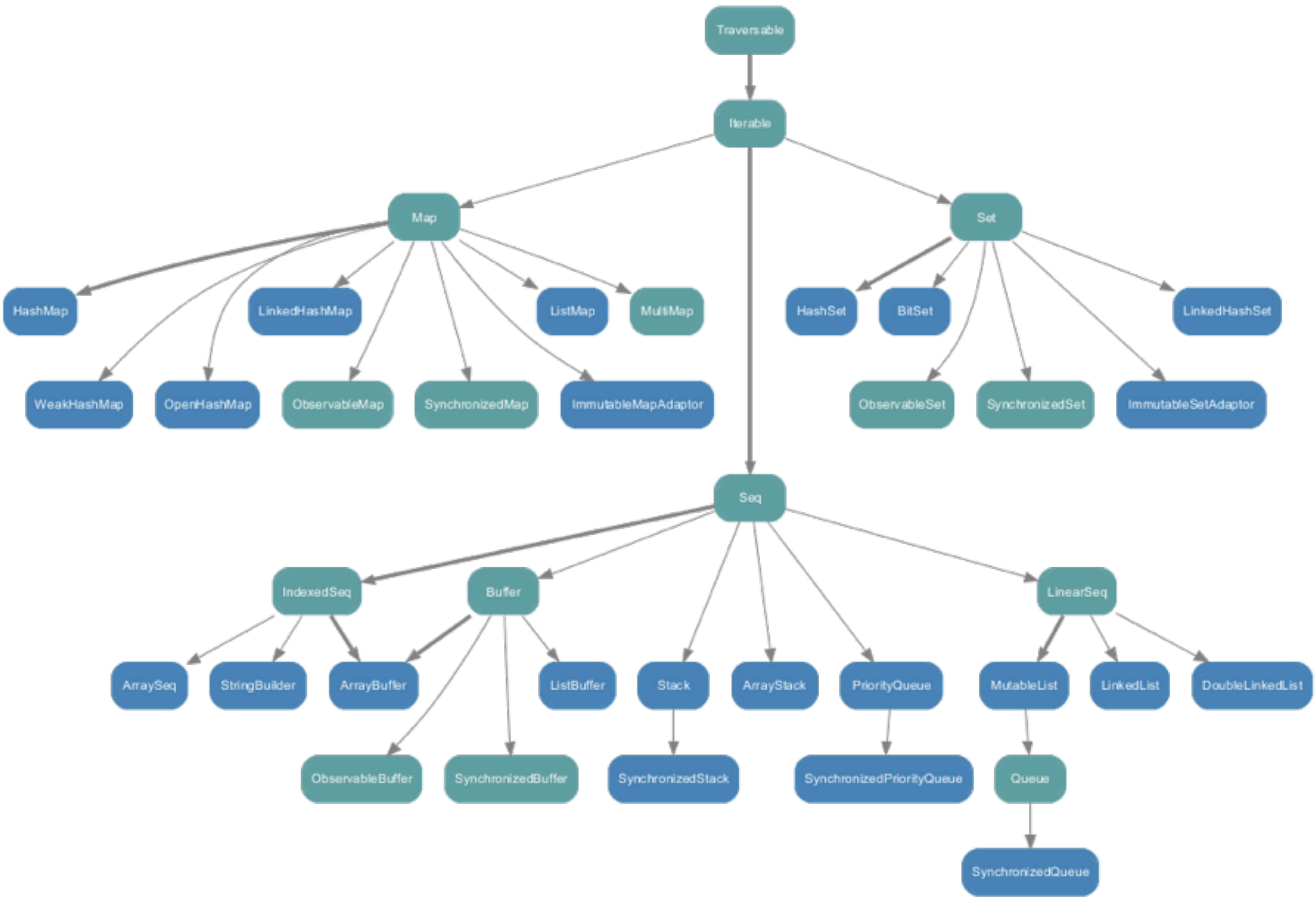
Collections in Scala



scala.collection.immutable



scala.collection.mutable



Trait Traversable

```
def foreach[U](f: Elem => U)
```

Indexed vs Linear Sequences

- Linear sequences are intended for recursive descent via head and tail (as with Lists)
- Indexed sequences are intended for random access to positions (as with Arrays)

Sorted Sets

- Sorted sets are non-repeating ordered collections of elements
- Canonical implementation is the `TreeSet` implementation (which uses red-black trees)

ListBuffers

- In the mutable package
- Constant time prepend and append operations
  - Append with `+=`
  - Prepend with `+=:`
  - Obtain a list by invoking `toList`

ArrayBuffers

- Like an array, but with prepend and append
- Prepending and appending on constant time on average but occasionally require linear time

Sets and Maps

- Mutable and immutable versions of these collections are available
- By default, you get the immutable versions
- Add and subtract elements using += and -=
- Add and subtract whole collections using ++= and —=