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>

↦
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