Chapter 18

Mutable Objects

In previous chapters, we put the spotlight on functional (immutable) objects. We did so because the idea of objects without any mutable state deserves to be better known. However, it is also perfectly possible to define objects with mutable state in Scala. Such mutable objects often come up naturally when you want to model objects in the real world that change over time.

This chapter explains what mutable objects are and what Scala provides in terms of syntax to express them. We will also introduce a larger case study on discrete event simulation, which involves mutable objects, as well as building an internal DSL for defining digital circuits to simulate.

18.1 WHAT MAKES AN OBJECT MUTABLE?

You can observe the principal difference between a purely functional object and a mutable one even without looking at the object's implementation. When you invoke a method or dereference a field on some purely functional object, you will always get the same result.

For instance, given a list of characters:

```
val cs = List('a', 'b', 'c')
```

an application of cs.head will always return 'a'. This is the case even if there is an arbitrary number of operations on the list cs between the point where it is defined and the point where the access cs.head is made.

For a mutable object, on the other hand, the result of a method call or field access may depend on what operations were previously performed on the object. A good example of a mutable object is a bank account. Listing 18.1 shows a simplified implementation of bank accounts:

```
class BankAccount {

  private var bal: Int = 0

  def balance: Int = bal

  def deposit(amount: Int) = {
    require(amount > 0)
    bal += amount
  }

  def withdraw(amount: Int): Boolean =
    if (amount > bal) false
    else {
      bal -= amount
      true
    }
}
```

Listing 18.1 - A mutable bank account class.

The BankAccount class defines a private variable, bal, and three public methods: balance returns the current balance; deposit adds a given amount to bal; and withdraw tries to subtract a givenamount from bal while assuring that the remaining balance won't be negative. The return value of withdraw is a Boolean indicating whether the requested funds were successfully withdrawn.

Even if you know nothing about the inner workings of the BankAccount class, you can still tell that BankAccounts are mutable objects:

```
scala> val account = new BankAccount
account: BankAccount = BankAccount@21cf775d

scala> account deposit 100

scala> account withdraw 80
res1: Boolean = true

scala>  account withdraw 80
res2: Boolean = false
```

Note that the two final withdrawals in the previous interaction returned different results. The first withdraw operation returned true because the bank account contained sufficient funds to allow the withdrawal. The second operation, although the same as the first one, returnedfalse because the balance of the account had been reduced so that it no longer covered the requested funds. So, clearly, bank accounts have mutable state, because the same operation can return different results at different times.

You might think that the mutability of BankAccount is immediately apparent because it contains a var definition. Mutation and vars usually go hand in hand, but things are not always so clear cut. For instance, a class might be mutable without defining or inheriting any vars because it forwards method calls to other objects that have mutable state. The reverse is also possible: A class might contain vars and still be purely functional. An example would be a class that caches the result of an expensive operation in a field for optimization purposes. To pick an example, assume the following unoptimized class Keyed with an expensive operationcomputeKey:

```
class Keyed {
  def computeKey: Int = ... // this will take some time
  ...
}
```

Provided that computeKey neither reads nor writes any vars, you can make Keyed more efficient by adding a cache:

```
class MemoKeyed extends Keyed {
  private var keyCache: Option[Int] = None
  override def computeKey: Int = {
    if (!keyCache.isDefined) keyCache = Some(super.computeKey)
    keyCache.get
  }
}
```

Using MemoKeyed instead of Keyed can speed things up because the second time the result of thecomputeKey operation is requested, the value stored in the keyCache field can be returned instead of running computeKey once again. But except for this speed gain, the behavior of class Keyed andMemoKeyed is exactly the same. Consequently, if Keyed is purely functional, then so is MemoKeyed, even though it contains a reassignable variable.

18.2 REASSIGNABLE VARIABLES AND PROPERTIES

You can perform two fundamental operations on a reassignable variable: get its value or set it to a new value. In libraries such as JavaBeans, these operations are often encapsulated in separate getter and setter methods, which need to be defined explicitly.

In Scala, every var that is a non-private member of some object implicitly defines a getter and a setter method with it. These getters and setters are named differently from the Java convention, however. The getter of a var x is just named "x", while its setter is named "x_=".

For example, if it appears in a class, the var definition:

```
var hour = 12
```

generates a getter, "hour", and setter, "hour_=", in addition to a reassignable field. The field is always marked private[this], which means it can be accessed only from the object that contains it. The getter and setter, on the other hand, get the same visibility as the original var. If the var definition is public, so are its getter and setter. If it is protected, they are alsoprotected, and so on.

For instance, consider the class Time shown in Listing 18.2, which defines two public vars named hour and minute:

```
class Time {
  var hour = 12
  var minute = 0
}
```

Listing 18.2 - A class with public vars.

This implementation is exactly equivalent to the class definition shown in Listing 18.3. In the definitions shown in Listing 18.3, the names of the local fields h and m are arbitrarily chosen so as not to clash with any names already in use.

```
class Time {

  private[this] var h = 12
  private[this] var m = 0

  def hour: Int = h
  def hour_=(x: Int) = { h = x }

  def minute: Int = m
  def minute_=(x: Int) = { m = x }
}
```

Listing 18.3 - How public vars are expanded into getter and setter methods.

An interesting aspect about this expansion of vars into getters and setters is that you can also choose to define a getter and a setter directly, instead of defining a var. By defining these access methods directly you can interpret the operations of variable access and variable assignment as you like. For instance, the variant of class Time shown in Listing 18.4 contains requirements that catch all assignments to hour and minute with illegal values.

```
class Time {

  private[this] var h = 12
  private[this] var m = 0

  def hour: Int = h
  def hour_= (x: Int) = {
    require(0 <= x && x < 24)
    h = x
  }

  def minute = m
  def minute_= (x: Int) = {
    require(0 <= x && x < 60)
    m = x
  }
}
```

Listing 18.4 - Defining getter and setter methods directly.

Some languages have a special syntactic construct for these variable-like quantities that are not plain variables in that their getter or setter can be redefined. For instance, C# has properties, which fulfill this role. In effect, Scala's convention of always interpreting a variable as a pair of setter and getter methods gives you the same capabilities as C# properties without requiring special syntax.

Properties can serve many different purposes. In the example shown in Listing 18.4, the setters enforced an invariant, thus protecting the variable from being assigned illegal values. You could also use a property to log all accesses to getters or setters of a variable. Or you could integrate variables with events, for instance by notifying some subscriber methods each time a variable is modified (you'll see examples of this in Chapter 35).

It's also possible, and sometimes useful, to define a getter and a setter without an associated field. For example, Listing 18.5 shows a Thermometer class, which encapsulates a temperature variable that can be read and updated. Temperatures can be expressed in Celsius or Fahrenheit degrees. This class allows you to get and set the temperature in either measure.

```
class Thermometer {

  var celsius: Float = _

  def fahrenheit = celsius * 9 / 5 + 32
  def fahrenheit_= (f: Float) = {
    celsius = (f - 32) * 5 / 9
  }
  override def toString = fahrenheit + "F/" + celsius + "C"
```

```
  }
```

Listing 18.5 - Defining a getter and setter without an associated field.

The first line in the body of this class defines a var, celsius, which will contain the temperature in degrees Celsius. The celsius variable is initially set to a default value by specifying `_' as the "initializing value" of the variable. More precisely, an initializer "= _" of a field assigns a zero value to that field. The zero value depends on the field's type. It is 0 for numeric types, falsefor booleans, and null for reference types. This is the same as if the same variable was defined in Java without an initializer.

Note that you cannot simply leave off the "= _" initializer in Scala. If you had written:

```
var celsius: Float
```

this would declare an abstract variable, not an uninitialized one.[1]

The celsius variable definition is followed by a getter, "fahrenheit", and a setter, "fahrenheit_=", which access the same temperature, but in degrees Fahrenheit. There is no separate field that contains the current temperature value in Fahrenheit. Instead the getter and setter methods for Fahrenheit values automatically convert from and to degrees Celsius, respectively. Here's an example of interacting with a Thermometer object:

```
scala> val t = new Thermometer
t: Thermometer = 32.0F/0.0C

scala> t.celsius = 100
t.celsius: Float = 100.0

scala> t
res3: Thermometer = 212.0F/100.0C

scala> t.fahrenheit = -40
t.fahrenheit: Float = -40.0

scala> t
res4: Thermometer = -40.0F/-40.0C
```

18.3 CASE STUDY: DISCRETE EVENT SIMULATION

The rest of this chapter shows by way of an extended example how mutable objects can be combined with first-class function values in interesting ways. You'll see the design and implementation of a simulator for digital circuits. This task is broken down into several subproblems, each of which is interesting individually.

First, you'll see a little language for digital circuits. The definition of this language will highlight a general method for embedding domain-specific languages (DSL) in a host language like Scala. Second, we'll present a simple but general framework for discrete event simulation. Its main task will be to keep track of actions that are performed in simulated time. Finally, we'll show how discrete simulation

programs can be structured and built. The idea of such simulations is to model physical objects by simulated objects, and use the simulation framework to model physical time.

The example is taken from the classic textbook *Structure and Interpretation of Computer Programs* by Abelson and Sussman [Abe96]. What's different here is that the implementation language is Scala instead of Scheme, and that the various aspects of the example are structured into four software layers: one for the simulation framework, another for the basic circuit simulation package, a third for a library of user-defined circuits, and the last layer for each simulated circuit itself. Each layer is expressed as a class, and more specific layers inherit from more general ones.

THE FAST TRACK

Understanding the discrete event simulation example presented in this chapter will take some time. If you feel you want to get on with learning more Scala instead, it's safe to skip ahead to the next chapter.

Figure 18.1 - Basic gates.

18.4 A LANGUAGE FOR DIGITAL CIRCUITS

We'll start with a "little language" to describe digital circuits. A digital circuit is built fromwires and function boxes. Wires carry signals, which are transformed by function boxes. Signals are represented by booleans: true for signal-on and false for signal-off.

Figure 18.1 shows three basic function boxes (or gates):

- An *inverter*, which negates its signal.
- An *and-gate*, which sets its output to the conjunction of its inputs.
- An *or-gate*, which sets its output to the disjunction of its inputs.

These gates are sufficient to build all other function boxes. Gates have delays, so an output of a gate will change only some time after its inputs change.

We'll describe the elements of a digital circuit by the following set of Scala classes and functions. First, there is a class Wire for wires. We can construct wires like this:

```
val a = new Wire
val b = new Wire
val c = new Wire
```

or, equivalent but shorter, like this:

```
val a, b, c = new Wire
```

Second, there are three procedures which "make" the basic gates we need:

```
def inverter(input: Wire, output: Wire)
def andGate(a1: Wire, a2: Wire, output: Wire)
def orGate(o1: Wire, o2: Wire, output: Wire)
```

What's unusual, given the functional emphasis of Scala, is that these procedures construct the gates as a side effect, instead of returning the constructed gates as a result. For instance, an invocation of inverter(a, b) places an inverter between the wires a and b. It turns out that this side-effecting construction makes it easier to construct complicated circuits gradually. Also, although methods most often have verb names, these have noun names that indicate which gate they are making. This reflects the declarative nature of the DSL: it should describe a circuit, not the actions of making one.

More complicated function boxes can be built from the basic gates. For instance, the method shown in Listing 18.6 constructs a half-adder. The halfAdder method takes two inputs, a and b, and produces a sum, s, defined by "s = (a + b) % 2" and a carry, c, defined by "c = (a + b) / 2". A diagram of the half-adder is shown in Figure 18.2.

```
def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire) = {
  val d, e = new Wire
  orGate(a, b, d)
  andGate(a, b, c)
  inverter(c, e)
  andGate(d, e, s)
}
```

Listing 18.6 - The halfAdder method.

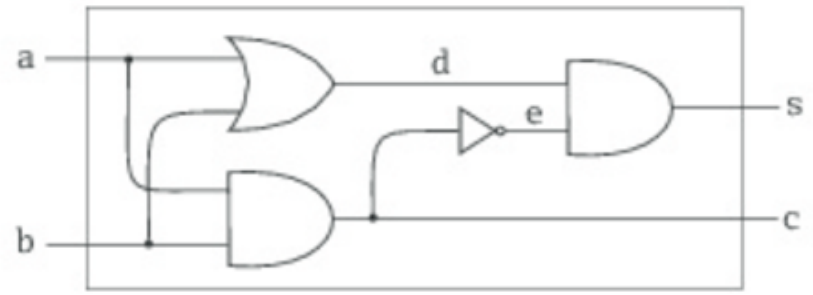

Figure 18.2 - A half-adder circuit.

Note that halfAdder is a parameterized function box just like the three methods that construct the primitive gates. You can use the halfAdder method to construct more complicated circuits. For instance, Listing 18.7 defines a full, one-bit adder, shown in Figure 18.3, which takes two inputs, a and b, as well as a carry-in, cin, and which produces a sum output defined by "sum = (a + b + cin) % 2" and a carry-out output defined by "cout = (a + b + cin) / 2".

```
def fullAdder(a: Wire, b: Wire, cin: Wire,
    sum: Wire, cout: Wire) = {

  val s, c1, c2 = new Wire
  halfAdder(a, cin, s, c1)
  halfAdder(b, s, sum, c2)
  orGate(c1, c2, cout)
```

```
}
```

Listing 18.7 - The fullAdder method.

Class Wire and functions inverter, andGate, and orGate represent a little language with which users can define digital circuits. It's a good example of an *internal* DSL, a domain-specific language defined as a library in a host language instead of being implemented on its own.

Figure 18.3 - A full-adder circuit.

The implementation of the circuit DSL still needs to be worked out. Since the purpose of defining a circuit in the DSL is simulating the circuit, it makes sense to base the DSL implementation on a general API for discrete event simulation. The next two sections will present first the simulation API and then the implementation of the circuit DSL on top of it.

18.5 THE SIMULATION API

The simulation API is shown in Listing 18.8. It consists of class Simulation in packageorg.stairwaybook.simulation. Concrete simulation libraries inherit this class and augment it with domain-specific functionality. The elements of the Simulation class are presented in this section.

```
abstract class Simulation {

  type Action = () => Unit

  case class WorkItem(time: Int, action: Action)

  private var curtime = 0
  def currentTime: Int = curtime

  private var agenda: List[WorkItem] = List()

  private def insert(ag: List[WorkItem],
      item: WorkItem): List[WorkItem] = {

    if (ag.isEmpty || item.time < ag.head.time) item :: ag
    else ag.head :: insert(ag.tail, item)
  }

  def afterDelay(delay: Int)(block: => Unit) = {
    val item = WorkItem(currentTime + delay, () => block)
    agenda = insert(agenda, item)
  }

  private def next() = {
    (agenda: @unchecked) match {
      case item :: rest =>
```

```
          agenda = rest
          curtime = item.time
          item.action()
        }
      }

      def run() = {
        afterDelay(0) {
          println("*** simulation started, time = " +
              currentTime + " ***")
        }
        while (!agenda.isEmpty) next()
      }
    }
```

Listing 18.8 - The Simulation class.

A discrete event simulation performs user-defined actions at specified *times*. The actions, which are defined by concrete simulation subclasses, all share a common type:

```
type Action = () => Unit
```

This statement defines Action to be an alias of the type of procedure that takes an empty parameter list and returns Unit. Action is a *type member* of class Simulation. You can think of it as a more readable name for type () => Unit. Type members will be described in detail inSection 20.6.

The time at which an action is performed is simulated time; it has nothing to do with the actual "wall clock" time. Simulated times are represented simply as integers. The current simulated time is kept in a private variable:

```
private var curtime: Int = 0
```

The variable has a public accessor method, which retrieves the current time:

```
def currentTime: Int = curtime
```

This combination of private variable with public accessor is used to make sure that the current time cannot be modified outside the Simulation class. After all, you don't usually want your simulation objects to manipulate the current time, except possibly if your simulation models time travel.

An action that needs to be executed at a specified time is called a work item. Work items are implemented by the following class:

```
case class WorkItem(time: Int, action: Action)
```

We made the WorkItem class a case class because of the syntactic conveniences this entails: You can use the factory method, WorkItem, to create instances of the class, and you get accessors for the constructor parameters time and action for free. Note also that class WorkItem is nested inside class Simulation. Nested classes in Scala are treated similarly to Java. Section 20.7 will give more details.

The Simulation class keeps an agenda of all remaining work items that have not yet been executed. The work items are sorted by the simulated time at which they have to be run:

```
private var agenda: List[WorkItem] = List()
```

The agenda list will be kept in the proper sorted order by the insert method, which updates it. You can see insert being called from afterDelay, which is the only way to add a work item to the agenda:

```
def afterDelay(delay: Int)(block: => Unit) = {
  val item = WorkItem(currentTime + delay, () => block)
  agenda = insert(agenda, item)
}
```

As the name implies, this method inserts an action (given by block) into the agenda so that it is scheduled for execution delay time units after the current simulation time. For instance, the following invocation would create a new work item to be executed at the simulated time,currentTime + delay:

```
afterDelay(delay) { count += 1 }
```

The code to be executed is contained in the method's second argument. The formal parameter for this argument has type "=> Unit" (*i.e.*, it is a computation of type Unit which is passed by name). Recall that by-name parameters are not evaluated when passed to a method. So in the call above, count would be incremented only when the simulation framework calls the action stored in the work item. Note that afterDelay is a curried function. It's a good example of the principle set forward in Section 9.5 that currying can be used to make method calls look more like built-in syntax. The created work item still needs to be inserted into the agenda. This is done by the insert method, which maintains the invariant that the agenda is time-sorted:

```
private def insert(ag: List[WorkItem],
    item: WorkItem): List[WorkItem] = {

  if (ag.isEmpty || item.time < ag.head.time) item :: ag
  else ag.head :: insert(ag.tail, item)
}
```

The core of the Simulation class is defined by the run method:

```
def run() = {
  afterDelay(0) {
    println("*** simulation started, time = " +
        currentTime + " ***")
  }
  while (!agenda.isEmpty) next()
}
```

This method repeatedly takes the first item in the agenda, removes it from the agenda and executes it. It does this until there are no more items left in the agenda to execute. Each step is performed by calling the next method, which is defined as follows:

```
private def next() = {
  (agenda: @unchecked) match {
    case item :: rest =>
```

```
        agenda = rest
        curtime = item.time
        item.action()
    }
  }
```

The next method decomposes the current agenda with a pattern match into a front item, item, and a remaining list of work items, rest. It removes the front item from the current agenda, sets the simulated time curtime to the work item's time, and executes the work item's action.

Note that next can be called only if the agenda is non-empty. There's no case for an empty list, so you would get a MatchError exception if you tried to run next on an empty agenda.

In fact, the Scala compiler would normally warn you that you missed one of the possible patterns for a list:

```
Simulator.scala:19: warning: match is not exhaustive!
missing combination            Nil

    agenda match {
    ^
one warning found
```

In this case, the missing case is not a problem because you know that next is called only on a non-empty agenda. Therefore, you might want to disable the warning. You saw in Section 15.5 that this can be done by adding an @unchecked annotation to the selector expression of the pattern match. That's why the Simulation code uses "(agenda: @unchecked) match", not "agenda match". That's it. This might look like surprisingly little code for a simulation framework. You might wonder how this framework could possibly support interesting simulations, if all it does is execute a list of work items? In fact the power of the simulation framework comes from the fact that actions stored in work items can themselves install further work items into the agenda when they are executed. That makes it possible to have long-running simulations evolve from simple beginnings.

18.6 CIRCUIT SIMULATION

The next step is to use the simulation framework to implement the domain-specific language for circuits shown in Section 18.4. Recall that the circuit DSL consists of a class for wires and methods that create and-gates, or-gates, and inverters. These are all contained in aBasicCircuitSimulation class, which extends the simulation framework. This class is shown inListings 18.9 and 18.10.

```
package org.stairwaybook.simulation

abstract class BasicCircuitSimulation extends Simulation {

  def InverterDelay: Int
  def AndGateDelay: Int
  def OrGateDelay: Int

  class Wire {

    private var sigVal = false
```

```
  private var actions: List[Action] = List()

  def getSignal = sigVal

  def setSignal(s: Boolean) =
    if (s != sigVal) {
      sigVal = s
      actions foreach (_ ())
    }

  def addAction(a: Action) = {
    actions = a :: actions
    a()
  }
}

def inverter(input: Wire, output: Wire) = {
  def invertAction() = {
    val inputSig = input.getSignal
    afterDelay(InverterDelay) {
      output setSignal !inputSig
    }
  }
  input addAction invertAction
}

// continued in Listing 18.10...
```

Listing 18.9 - The first half of the BasicCircuitSimulation class.

```
// ...continued from Listing 18.9
def andGate(a1: Wire, a2: Wire, output: Wire) = {
  def andAction() = {
    val a1Sig = a1.getSignal
    val a2Sig = a2.getSignal
    afterDelay(AndGateDelay) {
      output setSignal (a1Sig & a2Sig)
    }
  }
  a1 addAction andAction
  a2 addAction andAction
}

def orGate(o1: Wire, o2: Wire, output: Wire) = {
  def orAction() = {
    val o1Sig = o1.getSignal
    val o2Sig = o2.getSignal
    afterDelay(OrGateDelay) {
      output setSignal (o1Sig | o2Sig)
    }
  }
  o1 addAction orAction
  o2 addAction orAction
}

def probe(name: String, wire: Wire) = {
  def probeAction() = {
    println(name + " " + currentTime +
        " new-value = " + wire.getSignal)
```

```
      }
      wire addAction probeAction
    }
  }
```

Listing 18.10 - The second half of the BasicCircuitSimulation class.

Class BasicCircuitSimulation declares three abstract methods that represent the delays of the basic gates: InverterDelay, AndGateDelay, and OrGateDelay. The actual delays are not known at the level of this class because they depend on the technology of circuits that are simulated. That's why the delays are left abstract in class BasicCircuitSimulation, so that their concrete definition is delegated to a subclass.[2] The implementation of class BasicCircuitSimulation's other members is described next.

The Wire class

A wire needs to support three basic actions:

- getSignal: Boolean: returns the current signal on the wire.
- setSignal(sig: Boolean): sets the wire's signal to sig.
- addAction(p: Action): attaches the specified procedure p to the actions of the wire. The idea is that all action procedures attached to some wire will be executed every time the signal of the wire changes. Typically actions are added to a wire by components connected to the wire. An attached action is executed once at the time it is added to a wire, and after that, every time the signal of the wire changes.

Here is the implementation of the Wire class:

```
class Wire {

  private var sigVal = false
  private var actions: List[Action] = List()

  def getSignal = sigVal

  def setSignal(s: Boolean) =
    if (s != sigVal) {
      sigVal = s
      actions foreach (_ ())
    }

  def addAction(a: Action) = {
    actions = a :: actions
    a()
  }
}
```

Two private variables make up the state of a wire. The variable sigVal represents the current signal, and the variable actions represents the action procedures currently attached to the wire. The only interesting method implementation is the one for setSignal: When the signal of a wire changes, the new value is stored in the variable sigVal. Furthermore, all actions attached to a wire are executed. Note the shorthand syntax for doing this: "actions foreach (_ ())" applies the function, "_ ()", to each element in

the actions list. As described in Section 8.5, the function "_ ()" is a shorthand for "f => f ()"—*i.e.*, it takes a function (we'll call it f) and applies it to the empty parameter list.

The inverter method

The only effect of creating an inverter is that an action is installed on its input wire. This action is invoked once at the time the action is installed, and thereafter every time the signal on the input changes. The effect of the action is that the value of the inverter's output value is set (via setSignal) to the inverse of its input value. Since inverter gates have delays, this change should take effect only InverterDelay units of simulated time after the input value has changed and the action was executed. This suggests the following implementation:

```
def inverter(input: Wire, output: Wire) = {
  def invertAction() = {
    val inputSig = input.getSignal
    afterDelay(InverterDelay) {
      output setSignal !inputSig
    }
  }
  input addAction invertAction
}
```

The effect of the inverter method is to add invertAction to the input wire. This action, when invoked, gets the input signal and installs another action that inverts the output signal into the simulation agenda. This other action is to be executed after InverterDelay units of simulated time. Note how the method uses the afterDelay method of the simulation framework to create a new work item that's going to be executed in the future.

The andGate and orGate methods

The implementation of and-gates is analogous to the implementation of inverters. The purpose of an and-gate is to output the conjunction of its input signals. This should happen atAndGateDelay simulated time units after any one of its two inputs changes. Hence, the following implementation:

```
def andGate(a1: Wire, a2: Wire, output: Wire) = {
  def andAction() = {
    val a1Sig = a1.getSignal
    val a2Sig = a2.getSignal
    afterDelay(AndGateDelay) {
      output setSignal (a1Sig & a2Sig)
    }
  }
  a1 addAction andAction
  a2 addAction andAction
}
```

The effect of the andGate method is to add andAction to both of its input wires a1 and a2. This action, when invoked, gets both input signals and installs another action that sets the outputsignal to the conjunction of both input signals. This other action is to be executed afterAndGateDelay units of

simulated time. Note that the output has to be recomputed if either of the input wires changes. That's why the same andAction is installed on each of the two input wiresa1 and a2. The orGate method is implemented similarly, except it performs a logical-or instead of a logical-and.

Simulation output

To run the simulator, you need a way to inspect changes of signals on wires. To accomplish this, you can simulate the action of putting a probe on a wire:

```
def probe(name: String, wire: Wire) = {
  def probeAction() = {
    println(name + " " + currentTime +
        " new-value = " + wire.getSignal)
  }
  wire addAction probeAction
}
```

The effect of the probe procedure is to install a probeAction on a given wire. As usual, the installed action is executed every time the wire's signal changes. In this case it simply prints the name of the wire (which is passed as first parameter to probe), as well as the current simulated time and the wire's new value.

Running the simulator

After all these preparations, it's time to see the simulator in action. To define a concrete simulation, you need to inherit from a simulation framework class. To see something interesting, we'll create an abstract simulation class that extends BasicCircuitSimulation and contains method definitions for half-adders and full-adders as they were presented earlier in this chapter in Listings 18.6 and 18.7, respectively. This class, which we'll callCircuitSimulation, is shown in Listing 18.11.

```
package org.stairwaybook.simulation

abstract class CircuitSimulation
  extends BasicCircuitSimulation {

  def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire) = {
    val d, e = new Wire
    orGate(a, b, d)
    andGate(a, b, c)
    inverter(c, e)
    andGate(d, e, s)
  }

  def fullAdder(a: Wire, b: Wire, cin: Wire,
      sum: Wire, cout: Wire) = {

    val s, c1, c2 = new Wire
    halfAdder(a, cin, s, c1)
    halfAdder(b, s, sum, c2)
    orGate(c1, c2, cout)
  }
}
```

Listing 18.11 - The CircuitSimulation class.

A concrete circuit simulation will be an object that inherits from class CircuitSimulation. The object still needs to fix the gate delays according to the circuit implementation technology that's simulated. Finally, you will also need to define the concrete circuit that's going to be simulated.

You can do these steps interactively in the Scala interpreter:

```
scala> import org.stairwaybook.simulation._
import org.stairwaybook.simulation._
```

First, the gate delays. Define an object (call it MySimulation) that provides some numbers:

```
scala> object MySimulation extends CircuitSimulation {
         def InverterDelay = 1
         def AndGateDelay = 3
         def OrGateDelay = 5
       }
defined module MySimulation
```

Because you are going to access the members of the MySimulation object repeatedly, an import of the object keeps the subsequent code shorter:

```
scala> import MySimulation._
import MySimulation._
```

Next, the circuit. Define four wires, and place probes on two of them:

```
scala> val input1, input2, sum, carry = new Wire
input1: MySimulation.Wire =
    BasicCircuitSimulation$Wire@111089b
input2: MySimulation.Wire =
    BasicCircuitSimulation$Wire@14c352e
sum: MySimulation.Wire =
    BasicCircuitSimulation$Wire@37a04c
carry: MySimulation.Wire =
    BasicCircuitSimulation$Wire@1fd10fa

scala> probe("sum", sum)
sum 0 new-value = false

scala> probe("carry", carry)
carry 0 new-value = false
```

Note that the probes immediately print an output. This is because every action installed on a wire is executed a first time when the action is installed.

Now define a half-adder connecting the wires:

```
scala> halfAdder(input1, input2, sum, carry)
```

Finally, set the signals, one after another, on the two input wires to true and run the simulation:

```
scala> input1 setSignal true

scala> run()
```

```
*** simulation started, time = 0 ***
sum 8 new-value = true

scala> input2 setSignal true

scala> run()
*** simulation started, time = 8 ***
carry 11 new-value = true
sum 15 new-value = false
```

18.7 CONCLUSION

This chapter brought together two techniques that seem disparate at first: mutable state and higher-order functions. Mutable state was used to simulate physical entities whose state changes over time. Higher-order functions were used in the simulation framework to execute actions at specified points in simulated time. They were also used in the circuit simulations astriggers that associate actions with state changes. Along the way, you saw a simple way to define a domain-specific language as a library. That's probably enough for one chapter!

If you feel like staying a bit longer, you might want to try more simulation examples. You can combine half-adders and full-adders to create larger circuits, or design new circuits from the basic gates defined so far and simulate them. In the next chapter, you'll learn about type parameterization in Scala, and see another example in which a combination of functional and imperative approaches yields a good solution.

Footnotes for Chapter 18:

[1] Abstract variables will be explained in Chapter 20.

[2] The names of these "delay" methods start with a capital letter because they represent constants. They are methods so they can be overridden in subclasses. You'll find out how to do the same thing with vals in Section 20.3.