Extended Example: Discrete Event Simulation

1 Introduction

This chapter shows by way of an extended example how objects and higher-order functions can be combined in interesting ways. The task is to design and implement a digital circuit simulator. This task can be decomposed into several subproblems, each of which is interesting individually: First, you'll be presented a simple but general framework for discrete simulation. The main task of this framework is to keep track of simulated time. Second, you'll learn how to discrete simulation programs are are structured and built. The idea of such simulations is to model physical objects by simulated objects, and to use the simulation framework to model physical time. Finally, you'll see a little domain specific language for digital circuits. The definition of this language highlights a general method how domain-specific languages can be defined in a host language like Scala.

The basic example is taken from the classic textbook "Structure and Interpretation of Computer Programs" by Abelson and Sussman [**?**]. What's different here is that the implementation language is Scala instead of Scheme, and that the different aspects of the example are structured into four software layers: One for the simulation framework, another for the basic circuit simulation package, a third layer 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.

2 A language for digital circuits

Let's start with a little language to describe digital circuits. A digital circuit is built from *wires* and *function boxes*. Wires carry *signals* which are transformed by function boxes. Signals will be represented by booleans: `true` for signal-on and `false` for signal-off.

There are three basic function boxes (or: *gates*):

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

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 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 as follows.

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

or, equivalent but shorter:

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

Second, there are procedures

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

which "make" the basic gates we need. 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.

More complicated function boxes can now be built from these. For instance, the following method constructs a half-adder, which 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`.

```
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)
}
```

A picture of this half-adder is shown in Figure **??**.

Note that `halfAdder` is a parameterized function box just like the three methods which construct the primitive gates. You can use the `halfAdder` method to construct in turn more complicated circuits. For instance, the following defines a full one bit adder, (shown in Figure **??**) 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)
}
```

Class `Wire` and functions `inverter`, `andGate`, and `orGate` represent a little language in which users can define digital circuits. The implementations of this class and these functions are based on a simple and general API for discrete event simulation. This API will be studied next.

3 The Simulation API

The simulation API is given by a class `Simulation` in package `simulator`. Concrete simulation libraries will inherit this class and augment it with domain-specific functionality. The different elements of the `Simulation` class are presented in the following.

```
package simulator
class Simulation {
```

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

```
type Action = () ⇒ unit
```

The definition above defines `Action` to be an alias of the type of procedures that take an empty parameter list and that return `unit`.

The time at which an action is performed is simulated time; it is has nothing to do 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
```

As usual, the combination of private variable with public accessor is used to make sure that the current time cannot be modified outside the `Simulation` class.

An action which is to be executed at a specified time is called a *work-item*; it is an instance of the following class:

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

Note the difference between the type definition of `Action` and the class definition of `WorkItem`. A class definition defines a new type with some specified parents and components. By contrast, a type definition does not define a new type; it simply intrORUces a new name for an existing type.

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

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

The only way to add a work item to the agenda is with the following method:

```
def afterDelay(delay: int)(block: ⇒ unit) {
  agenda = insert(agenda, WorkItem(currentTime + delay, () ⇒ block))
}
```

A call like `afterDelay(delay} { count += 1 }` creates a new work-item which will executed at simulated time `currentTime + delay`. 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 call-by-name parameters are not evaluated when passed to a method. So in the call above `count` would be incremented only once the simulation framework calls the action which it stored in a work-item.

The created work-item is then 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 heart of the `Simulation` class is defined by the `run` method.

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

The method repeatedly takes the front item in the agenda and executes it. until there are no more items to execute. This step is performed by calling the `next` method, which is defined as follows.

```
private def next() {
  agenda 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.

That's it. This seems surprisingly little code for a simulation framework. One concern you might have is how interesting simulations can be supported by this framework, 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.

Missing cases and the @unchecked annotation

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 will 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 intentional, so you want to disable the warning. You can do this by adding an `@unchecked` annotation to the selector expression of the pattern match:

```
private def next() {
  (agenda: @unchecked) match {
    case item :: rest ⇒
      agenda = rest
      curtime = item.time
      item.action()
  }
```

Annotations are written as in Java; they start will an `@`-sign, which is followed by an identifier and possibly some arguments. What's new is that Scala lets you annotate not just definitions, but also types and expressions.

4 Circuit Simulation

The next step is to use the simulation framework to implement the domain-specific language for circuits. Recall that this DSL consists of a class for wires and methods that create and-gates, or-gates, and inverters. These are all contained in a class `BasicCircuitSimulation` which extends the simulation framework. Their implementations are presented in the following.

```
abstract class BasicCircuitSimulation extends Simulation {
```

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 the gates connected it. An action is also executed once at the time it is added to a wire.

Here is an 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 closure `(_ ())` to each element in the `actions` list. The closure itself is a shorthand for `(f ⇒ f ())`, i.e. it takes a function (let's name it `f`) and applies it to the empty parameter list.

Gate delays

Gate delays are specified by three abstract value definitions.

```
val InverterDelay: int
val andgatedelay: int
val orgatedelay: int
```

Particular simulations have to give concrete definitions of these delays (see below).

The Inverter Class

The only effect of creating an inverter is that an action is installed on its input wire. This action is invoked everytime the signal on the input changes. It's effect is that the value of the inverter's output value is set (via `setSignal`) to the inverse of its input value. However, 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
```

Note that the implementation uses the `afterDelay` method of the simulation framework to create a new work-item that's going to be executed in the future.

The And-Gate Class

And-gates are implemented analogously to inverters. The action of an `andGate` is to output the conjunction of its input signals. This should happen at `AndGateDelay` 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
  }
```

Note that the output has to be recomputed if either one of the input wires changes. That's why the same `andAction` is installed on each of the two input wires `a1` and `a2`.

Exercise: Write the implementation of `orGate`.

Exercise: Another way is to define an or-gate by a combination of inverters and and gates. Define a function `orGate` in terms of `andGate` and `inverter`. What is the delay time of this function?

Simulation output

To run the simulator, you still need a way to inspect changes of signals on wires. To this purpose, it's useful to add a `probe` method, which simulates 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
  }
} // end BasicCircuitSimulation
```

The effect of the `probe` procedure is to install a `probeAction` on a given wire. As usual, the installed action is executed everytime 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 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, let's assume there is a class `CircuitSimulation` which extends `BasicCircuitSimulation` and contains definitions of half adders and full adders as they were presented earlier in the chapter:

```
abstract class CircuitSimulation extends BasicCircuitSimulation {
  def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire) { ... }
  def fullAdder(a: Wire, b: Wire, cin: Wire, sum: Wire, cout: Wire) { ... }
}
```

A concrete circuit simulation will be an object that inherits that class. The object still needs to fix the gate delays according to the circuit implementation technology that's simulated. Finally, one also needs to define the concrete circuit that's going to be simulated. You can do these steps interactively using the Scala interpreter:

```
scala> import simulator._
import simulator._
```

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

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

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

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

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

scala> probe("sum", sum)
```

```
sum 0 new−value = false
unnamed0: Unit = ()

scala> probe("carry", carry)
carry 0 new−value = false
unnamed1: Unit = ()
```

Note that the probe's immediately print an output. This is a consequence of the fact that 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)
unnamed2: Unit = ()
```

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

```
scala> input1 setSignal true; run
error: That kind of statement combination is not supported by the interpreter.

scala> input1 setSignal true
unnamed3: Unit = ()

scala> run()
*** simulation started, time = 0 ***
sum 8 new−value = true
unnamed4: Unit = ()

scala> input2 setSignal true
unnamed5: Unit = ()

scala> run()
*** simulation started, time = 8 ***
carry 11 new−value = true
sum 15 new−value = false
unnamed6: Unit = ()
```

Summary

This chapter has brought together two techniques that seem at first disparate: mutable state and higher-order functions. Mutable state was used to simulate phsical 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 as *triggers* that associate actions with state changes. On the side, you have seen 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, maybe you want to try more simulation examples. You can combine half-adders and full-adders to larger circuits, or design new circuits from the basic gates defined so far and simulate them.