Discrete Event Simulation

Principles of Functional Programming

Advanced Example: Discrete Event Simulation

We now consider an example of how assignments and higher-order functions can be combined in interesting ways.

We will construct a digital circuit simulator.

This example also shows how to build programs that do discrete event simulation.

Digital Circuits

Let's start with a small description language for digital circuits.

A digital circuit is composed of *wires* and of functional components.

Wires transport signals that are transformed by components.

We represent signals using booleans `true` and `false`.

The base components (gates) are:

- The *Inverter*, whose output is the inverse of its input.
- The *AND Gate*, whose output is the conjunction of its inputs.
- The *OR Gate*, whose output is the disjunction of its inputs.

Other components can be constructed by combining these base components.

The components have a reaction time (or *delay*), i.e. their outputs don't change immediately after a change to their inputs.

Digital Circuit Diagrams

A Language for Digital Circuits

We describe the elements of a digital circuit using the following Scala classes and functions.

To start with, the class `Wire` models wires.

Wires can be constructed as follows:

```
val a = Wire(); val b = Wire(); val c = Wire()
```

or, equivalently:

```
val a, b, c = Wire()
```

Then, there exist the following functions, which create base components, as a side effect.

```
def inverter(input: Wire, output: Wire): Unit
def andGate(in1: Wire, in2: Wire, output: Wire): Unit
def orGate(in1: Wire, in2: Wire, output: Wire): Unit
```

Constructing Components

More complex components can be constructed from these.

For example, a half-adder can be defined as follows:

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

More Components

This half-adder can in turn be used to define a full adder:

```
def fullAdder(a: Wire, b: Wire, cin: Wire, sum: Wire, cout: Wire): Unit =
  val s = Wire()
  val c1 = Wire()
  val c2 = Wire()
  halfAdder(a, cin, s, c1)
  halfAdder(b, s, sum, c2)
  orGate(c1, c2, cout)
```

Exercise

What logical function does this program describe?

```
def f(a: Wire, b: Wire, c: Wire): Unit =
  val d, e, f, g = Wire()
  inverter(a, d)
  inverter(b, e)
  andGate(a, e, f)
  andGate(b, d, g)
  orGate(f, g, c)
```

0	`a & ~b`	0	`a & ~(b & a)`	0	`b & ~a`
0	`a == b`	0	`a != b`	0	`a * b`

Exercise

What logical function does this program describe?

```
def f(a: Wire, b: Wire, c: Wire): Unit =
  val d, e, f, g = Wire()
  inverter(a, d)
  inverter(b, e)
  andGate(a, e, f)
  andGate(b, d, g)
  orGate(f, g, c)
```

0	`a & ~b`	0	`a & ~(b & a)`	0	`b & ~a`
0	`a == b`	X	`a != b`	0	`a * b`

Implementation

The class `Wire` and the functions `inverter`, `andGate`, and `orGate` represent a small description language of digital circuits.

We now give the implementation of this class and its functions which allow us to simulate circuits.

These implementations are based on a simple API for discrete event simulation.

Actions

A discrete event simulator performs *actions*, specified by the user at a given *moment*.

An *action* is a function that doesn't take any parameters and which returns `Unit`:

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

The *time* is simulated; it has nothing to with the actual time.

Simulation Trait

A concrete simulation happens inside an object that inherits from the abstract class `Simulation`, which has the following signature:

```
trait Simulation:
  type Action = () => Unit
  def currentTime: Int = ???
  def afterDelay(delay: Int)(block: => Unit): Unit = ???
  def run(): Unit = ???
end Simulation
```

Here,

`currentTime` returns the current simulated time in the form of an integer.

`afterDelay` registers an action to perform after a certain delay (relative to the current time, `currentTime`).

`run` performs the simulation until there are no more actions waiting.

Class Diagram

The `Wire` Class

A wire must support three basic operations:

`getSignal(): Boolean`

Returns the current value of the signal transported by the wire.

`setSignal(sig: Boolean): Unit`

Modifies the value of the signal transported by the wire.

`addAction(a: Action): Unit`

Attaches the specified procedure to the *actions* of the wire. All of the attached actions are executed at each change of the transported signal.

Implemening Wires

Here is an implementation of the class `Wire`:

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

  def getSignal(): Boolean = sigVal

  def setSignal(s: Boolean): Unit =
    if s != sigVal then
      sigVal = s
      actions.foreach(_())

  def addAction(a: Action): Unit =
    actions = a :: actions
    a()
```

State of a Wire

The state of a wire is modeled by two private variables:

`sigVal` represents the current value of the signal.

`actions` represents the actions currently attached to the wire.

The Inverter

We implement the inverter by installing an action on its input wire.

This action produces the inverse of the input signal on the output wire.

The change must be effective after a delay of `InverterDelay` units of simulated time.

We thus obtain the following implementation:

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

The AND Gate

The AND gate is implemented in a similar way.

The action of an AND gate produces the conjunction of input signals on the output wire.

This happens after a delay of `AndGateDelay` units of simulated time.

We thus obtain the following implementation:

```
def andGate(in1: Wire, in2: Wire, output: Wire): Unit =
  def andAction(): Unit =
    val in1Sig = in1.getSignal()
    val in2Sig = in2.getSignal()
    afterDelay(AndGateDelay) { output.setSignal(in1Sig & in2Sig) }
  in1.addAction(andAction)
  in2.addAction(andAction)
```

The OR Gate

The OR gate is implemented analogously to the AND gate.

```
def orGate(in1: Wire, in2: Wire, output: Wire): Unit =
  def orAction(): Unit =
    val in1Sig = in1.getSignal()
    val in2Sig = in2.getSignal()
    afterDelay(OrGateDelay) { output.setSignal(in1Sig | in2Sig) }
  in1.addAction(orAction)
  in2.addAction(orAction)
```

Exercise

What happens if we compute `in1Sig` and `in2Sig` inline inside `afterDelay` instead of computing them as values?

```
def orGate2(in1: Wire, in2: Wire, output: Wire): Unit =
  def orAction(): Unit =
    afterDelay(OrGateDelay) {
      output.setSignal(in1.getSignal | in2.getSignal) }
    }
  in1.addAction(orAction)
  in2.addAction(orAction)
```

- ‘orGate‘ and ‘orGate2‘ have the same behavior.
- ‘orGate2‘ does not model OR gates faithfully.

Exercise

What happens if we compute `in1Sig` and `in2Sig` inline inside `afterDelay` instead of computing them as values?

```
def orGate2(in1: Wire, in2: Wire, output: Wire): Unit =
  def orAction(): Unit =
    afterDelay(OrGateDelay) {
      output.setSignal(in1.getSignal | in2.getSignal) }
    }
  in1.addAction(orAction)
  in2.addAction(orAction)
```

O ‘orGate‘ and ‘orGate2‘ have the same behavior.
X ‘orGate2‘ does not model OR gates faithfully.

The Simulation Trait

All we have left to do now is to implement the `Simulation` trait.

The idea is to keep in every instance of the `Simulation` trait an *agenda* of actions to perform.

The agenda is a list of `Events`. Each event is composed of an action and the time when it must be produced.

The agenda list is sorted in such a way that the actions to be performed first are in the beginning.

```
trait Simulation:
  ...
  private case class Event(time: Int, action: Action)
  private type Agenda = List[Event]
  private var agenda: Agenda = List()
```

Handling Time

There is also a private variable, `curtime`, that contains the current simulation time:

```
private var curtime = 0
```

An application of the `afterDelay(delay)(block)` method inserts the task

```
Event(curtime + delay, () => block)
```

into the `agenda` list at the right position.

Implementing AfterDelay

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

Implementing AfterDelay

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

The `insert` function is straightforward:

```
private def insert(ag: List[Event], item: Event): List[Event] = ag match
  case first :: rest if first.time <= item.time =>
    first :: insert(rest, item)
  case _ =>
    item :: ag
```

The Event Handling Loop

The event handling loop removes successive elements from the agenda, and performs the associated actions.

```
private def loop(): Unit = agenda match
  case first :: rest =>
    agenda = rest
    curtime = first.time
    first.action()
    loop()
  case Nil =>
```

Implementing Run

An application of the `run` method removes successive elements from the agenda, and performs the associated actions.

This process continues until the agenda is empty:

```
def run(): Unit =
  afterDelay(0) {
    println(s"*** simulation started, time = $currentTime ***")
  }
  loop()
```

Probes

Before launching the simulation, we still need a way to examine the changes of the signals on the wires.

To this end, we define the function `probe`.

```
def probe(name: String, wire: Wire): Unit =
  def probeAction(): Unit =
    println(s"$name $currentTime value = ${wire.getSignal()}")
  wire.addAction(probeAction)
```

Defining Technology-Dependent Parameters

It's convenient to pack all delay constants into their own trait which can be mixed into a simulation. For instance:

```
trait Delays:
  def InverterDelay = 2
  def AndGateDelay = 3
  def OrGateDelay = 5

object sim extends Circuits, Delays
```

Setting Up a Simulation

Here's a sample simulation that you can do in the worksheet.

Define four wires and place some probes.

```
import sim._
val input1, input2, sum, carry = Wire()
probe("sum", sum)
probe("carry", carry)
```

Next, define a half-adder using these wires:

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

Launching the Simulation

Now give the value `true` to `input1` and launch the simulation:

```
input1.setSignal(true)
run()
```

To continue:

```
input2.setSignal(true)
run()
```

A Variant

An alternative version of the OR-gate can be defined in terms of AND and INV.

```
def orGateAlt(in1: Wire, in2: Wire, output: Wire): Unit =
  val notIn1, notIn2, notOut = Wire()
  inverter(in1, notIn1); inverter(in2, notIn2)
  andGate(notIn1, notIn2, notOut)
  inverter(notOut, output)
```

Exercise

Question: What would change in the circuit simulation if the implementation of `orGateAlt` was used for OR?

0 Nothing. The two simulations behave the same.

0 The simulations produce the same events, but the indicated times are different.

0 The times are different, and `orGateAlt` may also produce additional events.

0 The two simulations produce different events altogether.

Summary

State and assignments make our mental model of computation more complicated.

In particular, we lose referential transparency.

On the other hand, the assignment allows us to formulate certain programs in an elegant way.

Example: discrete event simulation.

- Here, a system is represented by a mutable list of *actions*.
- The effect of actions, when they're called, change the state of objects and can also install other actions to be executed in the future.

As always, the choice between functional and imperative programming must be made depending on the situation.