Functions and State

Functions and State

Until now, our programs have been side-effect free.

Therefore, the concept of *time* wasn't important.

For all programs that terminate, any sequence of actions would have given the same results.

This was also reflected in the substitution model of computation.

Rewriting can be done anywhere in a term, and all rewritings which terminate lead to the same solution.

This is an important result of the λ-calculus, the theory behind functional programming.

Stateful Objects

One normally develops the world like a set of objects, some of which have a state that *changes* over the course of time.

An object *has a state* if its behavior is influenced by its history.

Example: a bank account has a state, because the answer to the question

> *"can I withdraw 100 CHF ?"*

may vary over the course of the lifetime of the account.

Implementation of State

Every form of mutable state is constructed from variables.

A variable definition is written like a value definition, but with the keyword `var` in place of `val`:

```
var x: String = "abc"
var count = 111
```

Just like a value definition, a variable definition associates a value with a name.

However, in the case of variable definitions, this association can be changed later through an *assignment*, like in Java:

```
x = "hi"
count = count + 1
```

State in Objects

Objects in the "real world" with state are represented by objects that have some variable members.

Example: Here is a class modeling a bank account.

```
class BankAccount {
  private var balance = 0
  def deposit(amount: Int): Unit = {
    if (amount > 0) balance = balance + amount
  }
  def withdraw(amount: Int): Int =
    if (0 < amount && amount <= balance) {
      balance = balance - amount
      balance
    } else throw new Error("insufficient funds")
}
```

State in Objects (2)

The class `BankAccount` defines a variable `balance` that contains the current balance of the account.

The methods `deposit` and `withdraw` change the value of the `balance` through assignments.

Note that `balance` is `private` in the `BankAccount` class, it therefore cannot be accessed from outside the class.

To create bank accounts, we use the usual notation for object creation:

```
val account = new BankAccount
```

Working with Mutable Objects

Here is a worksheet that manipulates bank accounts.

```
val account = new BankAccount        // account: BankAccount = BankAccount‘1797795
account deposit 50                   //
account withdraw 20                  // res1: Int = 30
account withdraw 20                  // res2: Int = 10
account withdraw 15                  // java.lang.Error: insufficient funds
```

Applying the same operation to an account twice in a row produces different results. Clearly, accounts are stateful objects.

Identity and Change

Assignment poses the new problem of deciding whether two expressions are "the same"

When one excludes assignments and one writes:

```
val x = E; val y = E
```

where `E` is an arbitrary expression, then it is reasonable to assume that `x` and `y` are the same. That is to say that we could have also written:

```
val x = E; val y = x
```

(This property is usually called *referential transparency*)

Identity and Change (2)

But once we allow the assignment, the two formulations are different. For example:

```
val x = new BankAccount
val y = new BankAccount
```

Q: Are `x` and `y` the same?

Operational Equivalence

To respond to the last question, we must specify what is meant by "the same".

The precise meaning of "being the same" is defined by the property of *operational equivalence*.

In a somewhat informal way, this property is stated as follows.

Suppose we have two definitions `x` and `y`.

`x` and `y` are operationally equivalent if *no possible test* can distinguish between them.

Testing for Operational Equivalence

To test if `x` and `y` are the same, we must

- Execute the definitions followed by an arbitrary sequence of operations that involves `x` and `y`, observing the possible outcomes.
- Then, execute the definitions with another sequence `S'` obtained by renaming all occurrences of `y` by `x` in `S`
- If the results obtained by executing `S'` are different, then the expressions `x` and `y` are certainly different.
- On the other hand, if all possible pairs of sequences `(S, S')` produce the same result, then `x` and `y` are the same.

Counterexample for Operational Equivalence

Based on this definition, let's see if the expressions

```
val x = new BankAccount
val y = new BankAccount
```

define the values `x` and `y` such that they are the same.

Let's follow the definitions by a test sequence:

```
val x = new BankAccount
val y = new BankAccount
x deposit 30                    // val res1: Int = 30
y withdraw 20                   // java.lang.Error: insufficient funds
```

Counterexample for Operational Equivalence (2)

Now rename all occurrences of `y` with `x` in this sequence. We obtain:

```
val x = new BankAccount
val x = new BankAccount
x deposit 30                    // val res1: Int = 30
x withdraw 20                   // val res2: Int = 10
```

The final results are different. We conclude that `x` and `y` are not the same.

Establishing Operational Equivalence

On the other hand, if we define

```
val x = new BankAccount
val y = x
```

then no sequence of operations can distinguish between `x` and `y`, so `x` and `y` are the same in this case.

Assignment and Substitution Model

The preceding examples show that our model of computation by substitution cannot be used.

Indeed, according to this model, one can always replace the name of a value by the expression that defines it. For example, in

```
val x = new BankAccount
val y = x
```

the `x` in the definition of `y` could be replaced by `new BankAccount`

But we have seen that this change leads to a different program.

The substitution model ceases to be valid when we add the assignment.

It is possible to adapt the model by introducing a *store*, but it becomes considerably more complicated.

Loops

Proposition: Variables make it possible to model all imperative programs.

But what about control statements like loops?

We can model them using functions.

Example: Here is a Scala program that uses a `while` loop:

```
def power (x: Double, exp: Int): Double = {
  var r = 1.0
  var i = exp
  while (i > 0) { r = r * x; i = i - 1 }
  r
}
```

In Scala, `while` is a keyword.

But how could we define `while` by using a function?

Definition of `while`

The instruction `while` can be defined as a function that takes two arguments:

- a condition of type boolean, and
- a command, of type `Unit`

The condition and the command must be passed by name so that they're reevaluated in each iteration.

This brings us to the following definition of `while`.

```
def while(condition: => Boolean)(command: => Unit): Unit =
  if (condition) {
    command; while(condition)(command)
  } else ()
```

Note that `while` is tail recursive, so it can operate with a constant stack size.

Exercise

Write a function implementing `repeat` loop that is used as follows:

```
repeat {
  command
} ( condition )
```

Is it also possible to obtain the following syntax?

```
repeat {
  command
} until ( condition )
```

For-Loops

The classical `for` loop in Java cannot be modeled simply by a higher-order function.

The reason is that in a Java program like

```
for (int i = 1; i < 3; i = i + 1) { System.out.print(i + " "); }
```

the arguments of `for` contain the *declaration* of the variable `i`, which is visible in other arguments and in the body.

However, in Scala there is a kind of `for` loops similar to Java's extended for loop:

```
for (i <- 1 until 3) { System.out.print(i + " ") }
```

This displays `1 2`.

Translation of For-Loops

For-loops translate similarly to for-expressions, but using the `foreach` combinator instead of `map` and `flatMap`.

`foreach` is defined on collections with elements of type `T` as follows:

```
def foreach(f: T => Unit): Unit =
  // apply ‘f‘ to each element of the collection
```

Example

```
for (i <- 1 until 3; j <- ”abc”) println(i+” ”+j)
```

translates to:

```
(1 until 3) foreach (i => ”abc” foreach (j => println(i+” ”+j)))
```

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

(see blackboard)

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 = new Wire; val b = new Wire; val c = new Wire
```

or, equivalently:

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

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

```
def inverter(input: Wire, output: Wire): Unit
def andGate(a1: Wire, a2: Wire, output: Wire): Unit
def orGate(o1: Wire, o2: 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 = new Wire
  val e = new 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 = new Wire
  val c1 = new Wire
  val c2 = new Wire
  halfAdder(a, cin, s, c1)
  halfAdder(b, s, sum, c2)
  orGate(c1, c2, cout)
}
```

What's Left To Do?

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 {
  def currentTime: Int = ???
  def afterDelay(delay: Int)(block: => Unit): Unit = ???
  def run(): Unit = ???
}
```

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.

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

Exercise

1. Write the implementation of the OR gate.
2. The OR gate can be defined in the same way by combining inverters and AND gates. Define a function `orGate` in terms of `andGate` and `inverter`. What is the delay of this component?

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 pairs. Each pair 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 {
  case class WorkItem(time: Int, action: Action)
  private type Agenda = List[WorkItem]
  private var agenda: Agenda = List()
}
```

Implementation of AfterDelay

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

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

into the `agenda` list at the right position.

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("*** simulation started, time = "+currentTime+" ***")
  }
  while (!agenda.isEmpty) next()
}
```

Exercise

The `run` method uses the `next` function, which removes the first action in the agenda, executes it, and updates the current time.

1. Provide an implementation for `next`.
2. Provide an implementation for `afterDelay`.

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(name + " " + currentTime + " value = " + wire.getSignal)
  }
  wire addAction probeAction
}
```

Setting Up a Simulation

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

Define four wires and place some probes.

```
val input1, input2, sum, carry = new 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
```

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.