Week 8: Functions and State

Up to now, all our programs did not have side-effects.

Therefore, the notion of *time* did not matter.

For a program that terminates, any sequence of actions would have led to the same result!

This is also reflected by the substitution model of computation.

A rewrite-set can be applied anywhere in a term, and all rewritings that terminate lead to the same solution.

This is a deep result in λ-calculus, the theory underlying functional programming.

More about that in the course "Concurrency: Languages, Programming and Theory".

Stateful Objects

We normally view the world as a set of objects, some of which have state that *changes* over time.

An object has state (or: is stateful) if its behavior is influenced by its history.

Example: a bank account object has state, because the question

"can I withdraw 100 CHF?"

might have different answers during the lifetime of the account.

Implementing State

All mutable state is ultimately built from variables.

A variable definition is written like a value definition, but starts with ***var*** instead of ***val***.

Example:

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

Like a value definition, a variable definition associates a name with a value.

But in the case of a variable definition, this association may be changed later by an assignment.

Assignments are written like in Java.

Example:

```
x = "hello";
count = count + 1;
```


State in Objects

Real-world objects with state are represented by objects that have variables as members.

Example: Here is a class that represents bank accounts.

```
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 error("insufficient funds");
}
```

The class defines a variable *balance* which contains the current balance of an account.

Methods *deposit* and *withdraw* can change the value of *balance* through assignments.

Note that *balance* is ***private*** in class *BankAccount* – hence it can not be accessed directly outside the class.

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

```
val myAccount = new BankAccount
```


Example: Here is a *siris* session that deals with bank accounts.

```
> :l bankaccount.scala
loading file 'bankaccount.scala'
> val account = new BankAccount
val account : BankAccount = BankAccount$class@1797795
> account deposit 50
(): scala.Unit
> account withdraw 20
30: scala.Int
> account withdraw 20
10: scala.Int
> account withdraw 15
java.lang.RuntimeException: insufficient funds
        at error(Predef.scala:3)
        at BankAccount$class.withdraw(bankaccount.scala:13)
        at <top-level>(console:1)
>
```

Applying the same operation twice to an account yields different results.

So, clearly, accounts are stateful objects.

Sameness and Change

Assignments pose new problems in deciding when two expressions are "the same".

If assignments are excluded, and one writes

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

where E is an arbitrary expression, then x and y can reasonably be assumed to be the same. I.e. one could have equivalently written

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

(This property is usually called referential transparency.)

But once we admit assignments, the two definition sequences are different. Consider:

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

Q: Are x and y the same?

Operational Equivalence

To answer the last question, we need to be more precise what "sameness" means.

The precise meaning of "being the same" is captured in the property of *operational equivalence.*

Somewhat informally, this property is stated as follows.

Suppose we have two definitions of x and y.

To test whether x and y are the same,

- Execute the definitions followed by an arbitrary sequence S of operations that involve x and y. Observe the results (if any).
- Then, execute the definitions with another sequence S' which results from S by renaming all occurrences of y in S to x.
- If the results of running S' are different, then surely x and y are different.

- On the other hand, if all possible pairs of sequences (S, S') yield the same results, then x and y are the same.

Given this definition, let's test whether

> **val** *x* = **new** *BankAccount*
> **val** *y* = **new** *BankAccount*

defines values x and y which are the same.

Here are the definitions again, followed by a test sequence:

```
> val x = new BankAccount
> val y = new BankAccount
> x deposit 30
30
> y withdraw 20
java.lang.RuntimeException: insufficient funds
```

Now, rename all occurrences of y in that sequence to x. We get:


```
> val x = new BankAccount
> val y = new BankAccount
> x deposit 30
30
> x withdraw 20
10
```

Since the final results are different, we have established that x and y are not the same.

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 the Substitution Model

These examples show that our previous substitution model of computation cannot be used anymore.

After all, under this substitution model, we could always replace a value name by its defining expression.

For instance 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.

So the substitution model must be invalid, once we add assignments.

We will see next week how to modify the substitution model so that it can deal with the additions.

Loops

Claim: Variables are enough to model all of imperative programming.

But what about control structures such as loops?

These can be modelled 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
}
```

The example shows that *while* is not a reserved word.

How is the name defined?

Definition of *while*

while is a function that takes two parameters:

- a condition, of type *boolean*, and
- a command, of type *unit*.

Both condition and command need to be passed by-name, so that they are evaluated repeatedly.

This leads to the following definition of *while*.

```
def while (def condition: boolean) (def command: unit): unit =
  if (condition) {
    command; while (condition) (command)
  } else {
  }
```

Note that *while* is tail recursive, so it should be able to operate in constant stack space.

Exercise: Write a function implementing *repeat* loop, which should be applied as follows:

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

Is there also a way to obtain a loop syntax like the following?

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

In the Scala interpreter, functions for *while* and *repeat* are pre-loaded in file *Predef.scala*.

So they are available without further definitions to user programs.

For-Loops

Java's for-loop is an exception; it cannot be simply modelled as a higher-order function.

The reason is that in code like

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

the argument to **for** contains a *declaration* of the variable *i*, which is visible in the other arguments and in the body.

However, there exists a for-loop syntax in Scala that does something similar.

```
for (val i ← List.range(1, 3)) do { System.out.print(i + " ") }
```

This will print *1 2*.

Contrast this with

```
> for (val i ← List.range(1, 3)) yield i
List(1, 2)
```


Extended Example: Discrete Event Simulation

We now discuss an example, which demonstrates how assignments and higher-order functions can be combined in interesting ways.

We will build a simulator for digital circuits.

The example also shows how discrete event simulation programs in general are structured and built.

Digital Circuits

We 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.

We will represent signals by the booleans ***true*** and ***false***.

Basic function boxes (or: *gates*) are:

- 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.

Other function boxes can be built by combining basic ones.

Gates have *delays*, so an output of a gate will change only some time after its inputs change.

A Language for Digital Circuits

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;
```

Second, there are functions

```
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
```

which "make" the basic gates we need (as side-effects).

More complicated function boxes can now be built from these.

For instance, to construct a half-adder, we can define:

```
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);
}
```


This abstraction can itself be used, for instance in defining a full adder:

```
def fullAdder(a: Wire, b: Wire, cin: Wire, sum: Wire, cout: Wire) = {
  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 Else Needs to Be Done?

To summarize, class *Wire* and functions *inverter*, *andGate*, and *orGate* represent a little language in which users can define digital circuits.

We now give implementations of this class and these functions, which allow one to simulate circuits.

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

The Simulation API

Discrete event simulation performs user-defined actions at specified times.

An action is a function which takes no parameters and returns a *unit* result:

```
type Action = () => unit;
```

The time is simulated; it is not the actual "wall-clock" time.

A concrete simulation will be done inside an object which inherits from the abstract *Simulation* class, which has the following signature:

```
abstract class Simulation {
  def currentTime: int;
  def afterDelay(delay: int)(action: Action): unit;
  def run: unit;
}
```


Here,

currentTime returns the current simulated time as an integer number.

afterDelay schedules an action to be performed at a specified delay after *currentTime*.

run runs the simulation until there are no further actions to be performed.

The Wire Class

A wire needs to support three basic actions.

- *getSignal*: *boolean* returns the current signal on the wire.
- *setSignal*(*sig*: *boolean*): *unit* sets the wire's signal to *sig*.
- *addAction*(*p*: *Action*): *unit* attaches the specified procedure *p* to the *actions* of the wire. All attached action procedures will be executed every time the signal of a wire changes.

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(action => action());
    }
  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.
- The variable *actions* represents the action procedures currently attached to the wire.

Inverters

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

The action of an *inverter* is to output the negated input signal.

The action needs to take effect at *InverterDelay* simulated time units after the input changes.

Hence, the following implementation:

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


And-Gates

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;
}
```


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?

The Simulation Class

Now, we just need to implement class *Simulation*, and we are done.

The idea is that we maintain inside a *Simulation* object an agenda of actions to perform.

The agenda is represented as a list of pairs of actions and the times they need to be run.

The agenda list is sorted, so that earlier actions come before later ones.

```
class Simulation {
  private type Agenda = List[Pair[int, Action]];
  private var agenda: Agenda = List();
```

There is also a private variable *curtime* to keep track of the current simulated time.

```
  private var curtime = 0;
```


An application of the method *afterDelay(delay)(action)* inserts the pair *(curtime + delay, action)* into the *agenda* list at the appropriate place.

An application of the method *run* removes successive elements from the *agenda* and performs their actions.

It continues until the agenda is empty:

```
def run = {
  afterDelay(0){ () => System.out.println("*** simulation started ***"); }
  while (!agenda.isEmpty) { next }
}
```

It makes use of a function *next*, which removes the first element of the agenda and performs its action.

The implementations of *next* and *afterDelay* are left as an exercise.

Running the Simulator

To run the simulator, we still need a way to inspect changes of signals on wires.

To this purpose, we write a function *probe*.

```
def probe(name: String, wire: Wire): unit = {
  wire addAction { () =>
    System.out.println(
      name + " " + currentTime + " new_value = " + wire.getSignal);
  }
}
```

Now, define four wires, and place probes on two of them:


```
> val input1 = new Wire
> val input2 = new Wire
> val sum = new Wire
> val carry = new Wire

> probe("sum", sum)
sum 0 new_value = false
> probe("carry", carry)
carry 0 new_value = false
```

Define a half-adder connecting the wires:

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

set *input1* to **true** and run the simulation.

```
> input1 setSignal true; run
*** simulation started ***
sum 8 new_value = true
> input2 setSignal true; run
carry 11 new_value = true
sum 15 new_value = false
```

etc.

Summary

- State and Assignment complicate our mental model of computation.
- In particular, referential transparency is lost.
- On the other hand, assignment gives us new ways to formulate programs elegantly.
- Example: Discrete event simulation.
- Here, a system is represented by a mutable list of *action* procedures.
- Action procedures, when called, change the state of objects and can also install further action procedures.
- As always, it depends on the situation whether purely functional programming or programming with assignments works best.