Functions and evaluations

Objectives

- To learn about the basic principles of functional programming
- To learn how to construct functional programs
- To be able to reason about functional programs

Programming paradigms

Paradigm, in science, this refers distinct concepts of thought patterns in some scientific discipline. Main programming paradigms are:

- Imperative programming
- Functional programming
- Logic programming

Orthogonal to them, is object-oriented programming

Imperative programming, is about the modification of mutable variables using assignments and control structures such as selection, loops, and interruptions. The most common informal way to understand it is using the instructions sequences for a Von-neumann computer (CPU + RAM + IO bus). That model, mantains a strong correspondence between its structures and computer hardware concepts (mutable variables with memory cells, variable dereferences with load instructions, variable assignments with store instructions, and control instructions with jumps). There's a problem with that, the conceptualization of programs written in that style, is attached with the meaning of the program word by word (John Backus, *Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs*). In the end, imperative programming is limited by the "Von Neumann" bottleneck. We need other techniques for defining high-level abstractions such as collections, polynomials, string, documents. And ideally, to develop a **theory** of those objects to enable higher-order reasoning.

A **theory**, consists of one or more data types, operations on these types, and laws to describe the relationships between values and operations (a theory, does not describes mutations, that is, to change an object keeping its identity the same). For instance, the theory of polynomials, defines the sum of two polynomials by laws such as:

```
(a*x + b) + (c*x + d) = (a + c)*x + (b + d)
```

But it does not define an operator to change the coefficient while keeping the polynomial the same. But one could write an imperative program such as:

```
struct Polynomial {
    char* coefficients;
};

struct Polynomial p;
p.coeffiecients = (char*) malloc(sizeof(char) * 50);
*(p.coefficients + 0) = 42
```

In this piece of code, one coefficient of the polynomial is altered, while the polynomial, remains the same. That's an inconsistency with the theory. If we want to implement high-level concepts following their mathematical theories, there's not place for mutation. Since the theories does not admits it, since it can destroy useful laws in the theories. That leads to a new style of programming, where we concentrate on the definition of theories for operators expressed as functions, avoid mutations, and provide powerful means to abstract and compose functions.

Functional programming

In a restricted sense, functional programming means programming without mutable variables, assignments, loops, and other imperative programming control structures. In a wider sense, FP, means focusing on the functions. In particular, functons can be values that are produced, consumed and composed. A functional programming languages, in the restricted sense, does not have mutable variables, assignments of imperative control structures. In a wider sense, a FPL enables the construction of elegant programs that focus on the functions. In particular, functions in a FPL are first-class citizens. This means:

- They can be defined anywhere, including inside functions
- The can be passed as parameters to functionas, and be returned as values
- There exists a set of operators to compose functions

Some functional programming languages in the not restricted view:

- Lisp, Scheme, Clojure
- SML, OCaml, F#
- Haskell
- Scala

- Smalltalk, Ruby

Functional programming offers simpler reasoning principles. better modularity, and good exploitation of parallelism for multicore and cloud computing.

Elements of programming

Every non-trivial language, has primitive expressions that represents the simplest elements in the language (atoms), ways to combine expressions, and ways to abstract expressions, which introduce a name for an expression by which it can then be referred to. A **REPL**, provides an interactive shell in which one can type expressions, ang get the rcomputation of those expressions.

```
scala
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_45).
Type in expressions to have them evaluated.
Type :help for more information.

scala> 4 + 5
res0: Int = 9

scala> def radius = 10
radius: Int

scala> def pi = 3.14159
pi: Double

scala> raidus * pi
res1: Double = 31.4159
```

Evaluation

Every non-primitive expression, is evaluated as follows:

1. Take the leftmost operator
2. Evaluate its operands (left before right)
3. Apply the operator to the operands

A name is evaluated by replacing it with the right hand side of its definition (no mutability accepted). The evaluation halts once it result in a value (an atom).

Example:

```
EXP: (2 * pi) * radius
eval(EXP)
    LEFTMOST = (2 * pi)
    eval(LEFTMOST)
        LEFTMOST = 2
        eval(LEFTMOST) = 2
        RIGHTMOST = pi
        eval(RIGHTMOST) = 3.14159
    = 6.28318
    RIGHTMOST = radius
    eval(radius) = 10
= 62.8318
```

Evaluation of function applications

Definitions can also have parameters

```
scala> def square(x: Double) = x * x
square: (Double)Double

scala> def sumOfSquares(x: Double, y: Double) = square(x) + square(y)
sumOfSquares: (Double,Double)Double

// Optinal type
scala> def power(x: Double, y: Int): Double = ...
```

```
Int         32-bit integers
Double      64-bit floating point numbers
Boolean     true, false
```

Applications of parametrized functions are evaluated in a similar way as operators:

1. Evaluate all function arguments, from left to right
2. Replace the function application by the function's right-hand side, and at the same time
3. Replace the formal parameters of the functions, by the actual parameters

Example:

```
EXP: sumOfSquares(3, 2 + 2)
eval(EXP)
    sumOfSquare(3, 2 + 2)
    sumOfSquare(eval(3), 2 + 2)
    sumOfSquare(3, eval(2 + 2))
    sumOfSquare(3, 4)
    square(3) + square(4)
    square(eval(3)) + square(eval(4))
    3 * 3 + square(4)
    9 + 4 * 4
    9 + 16
= 25
```

This scheme is called the substitution model. The general idea, is that evaluation does reduce an expression to a value (simple rewriting steps until there's a final value). It can be applied to every expression, as long as they have no side effects. It is formalized in the lambda calculus, which also gives a foundation for functional programming, and is equivalent to Turing machines. Not every expression reduce to a value. For instance:

```
def loop: Int = loop
```

One could alternatively, apply the function to unreduced arguments while evaluating a function application.

```
EXP: sumOfSquares(3, 2 + 2)
eval(EXP)
    sumOfSquare(3, 2 + 2)
    square(3) + square(2 + 2)
    3 * 3 + square(2 + 2)
    9 + (2 + 2) * (2 +2)
    9 + 4 * (2 + 2)
    9 + 4 * 4
= 25
```

The first evaluation strategy, is called **call-by-value**, the second one, is known as **call-by-name**. Both strategies reduce to the same final values as long as:

- The reduced expression consists of pure functions
- Both evaluations terminate

call-by-value, has the adventage that it evaluates every function argument just once. **call-by-name**, has the adventage that the function argument is not evaluated if the corresponding parameter is unused in the evaluation of the function's body.

Example:

```
def test(x: int, y: Int): Int = x * x
```

call-by-value vs **call-by-name** (eval(atom) does not count)

```
EXP: test(2, 3) -> Same complexity

eval(EXP, cbv)                              eval(EXP, cbn)
    2 * 2                                       2 * 2
    4                                           4

EXP: test(3 + 4, 8)  -> call-by-value wins

eval(EXP, cbv)                              eval(EXP, cbn)
    test(7, 8)                                  (3 + 4) * (3 + 4)
    7 * 7                                       7 * (3 + 4)
    49                                          7 * 7
                                                49

EXP: test(7, 2 * 4) -> call-by-name wins

eval(EXP, cbv)                              eval(EXP, cbn)
    test(7, 8)                                  7 * 7
    7 * 7                                       49
    49

EXP: test(3 + 4, 2 * 4) -> Same complexity

eval(EXP, cbv)                              eval(EXP, cbn)
    test(7, 2 * 4)                              (3 + 4) * (3 + 4)
    test(7, 8)                                  7 * (3 + 4)
    7 * 7                                       7 * 7
    49                                          49
```

Evaluation models and termination

Here we explore the means of evaluation and termination.

Theorem *call-by-value* and *call-by-name* evaluation strategies reduce an expression to the same value, as long as both evaluations terminate.

Theorem If *call-by-value* evaluation of EXP terminates, then *call-by-name* evaluation terminates, too.

Example:

```
def fisrt(x: Int, y: Int) = x

EXP: fisrt(1, loop)

eval(EXP, cbv)                                  eval(EXP, cbn)
    LOOP                                            1
```

Scala uses normally call-by-value, sinc in practice, cbv, avoids recomputation of expressions, leading to an exponential factor incresing its performance, and plays much nicer with side effects and imperative programming. Scala let's one force cbn, using `paramater: => Type`.

Example

```
def one(x: Int, y: => Int) = 1

EXP: one(1 + 2, loop)

eval(EXP)
    one(3, loop)    // Since first argument is cbv
    1               // Since second argument is cbn

EXP: one(loop, 1 + 2)

eval(EXP)
    one(loop, 1 + 2)
    one(loop, 1 + 2)
    one(loop, 1 + 2)
    ...
```

Conditionals and value definitions

Conditional expressions

To express choosing betweene two alternatives, Scala has the conditional expression `if-else`. It is used for expressions, not statements.

```
def abs(x: Int): Int = if (x >= 0) x else -x
```

`x >= 0` is a predicate, of type Boolean. Boolean expressions can be composed of

```
true false       // Constants
!b               // Negation
b && b           // Conjunction
b || b           // Disjunction

// Usual comparison operations:

e <= e, e >= e. e < e, e > e, e == e, e != e
```

The rewrite rules fro Boolean expressions are (semantics is defined using the substitution model):

```
!true           ->      false
!false          ->      true
true && e       ->      e
false && e      ->      false       // Short-circuit evaluation
true || e       ->      true        // Short-circuit evaluation
false || e      ->      e
```

Example:

```
if (b) e1 else e2

eval(b) = true THEN
    e1
eval(b) = false THEN
    e2
```

Value definitions

We've seen the function parameters, can be passed by value, or be passed by name. The same applies to definitions. The `def` form, is **by-name** (its right hand side is evaluated on each use). The `val` form, id **by-**

value.

Example:

```
val x = 2
val y = square(x)
```

The right-hand side of a `val` definition, is evaluated at the point of the definition itself. Afterwards, the name refers to the value. For instance, `y` above refers to 4, not `square(2)`.

Example:

```
def and(x: Boolean, y: => Boolean) = if (x) y else false
def or(x: Boolean, y: Boolean) = if (x) true else y
```

An example of a program: Newton's method

To calculate square roots with Newton's method. We will define a function with the following signature:

```
/** Calculate the square root of parameter x */
def sqrt(x: Duble): Double = ...
```

The classical way to achieve that, is by successive approximations using Newton's method.

Example: `x = 2`

- Start with an initial positive number as guess (let's pick `y=1`)
- Repeatedly improve the estimate by taking the mean of `y` and `x/y`

Estimation	Quotient	Mean
1	2 / 1 = 2	1.5
1.5	2 / 1.5 = 1.333	1.4167
1.4167	2 / 1.4167 = 1.4118	1.4142
1.4142	...	...

```scala
val EPSILON = 0.000001

def abs(x: Double) = if (x < 0) -x else x

// Recursive definitons need an explicit return type in Scala
def newton(guess: Double, x: Double): Double =
    if (isGoodEnough(guess, x)) guess
    else newton(improve(guess, x), x)

def isGoodEnough(guess: Double, x: Double) =
    abs(guess * guess - x) < EPSILON

def improve(guess: Double, x: Double) =
    (guess + x / guess) / 2

def sqrt(x: Double) = newton(1.0, x)

println(sqrt(2))
println(sqrt(9))
println(sqrt(25))
```

Blocks and lexical scoping

To develop a mechanism based on blocks and lexical scoping to organize programs. It's good for functional programming style to split up a task into many small functions. For example, the functions `abs, newton, isGoodEnough, improve`, matter only for the implementation of `sqrt`, not for its usage. Normally, we would not like users to access these functions directly. We can achieve the same functionality by putting those auxiliary functions inside `sqrt`.

Blocks affects the visibility of variables in a program.

Example:

```scala
val x = 0
def f(y: Int) = y + 1
val result = {
    val x = f(3)
    x * x
}
```

- The definitions inside a block are only visible from within the block
- The definitions inside a block shadow definitions of the same names outside the block

Example:

```scala
val x = 0
def f(y: Int) = y + 1
val result = {
    val x = f(3)
    x * x
} + x
```

The result value of the program, is 16.

In Scala, semicolons at the end of the line are in most cases optional. If there are more than one statement on a line they need to be separated by semicolons.

```scala
val y = x + 1; y * y
```

Finally, a modified version of the Newton's method using blocks.

```scala
def abs(x: Double) = if (x < 0) -x else x

/* A block is delimited by brces, it may contain a sequence of definitions or
 * expressions. The last element (expression) of a block defines the value of
 * the block. This return expression, can be preceded by auxiliary definitions
 * Blocks are themselves expressions; a block may appear everywhere an
 * expression can.
*/
def sqrt(x: Double) = {

    val EPSILON = 0.000001

    // Recursive definitons need an explicit return type in Scala
    def newton(guess: Double): Double =
        if (isGoodEnough(guess)) guess
        else newton(improve(guess))

    def isGoodEnough(guess: Double) =
        abs(guess * guess - x) / x < EPSILON

    def improve(guess: Double) =
        (guess + x / guess) / 2

    newton(1.0)
}

println(sqrt(2))
println(sqrt(1e-6))
println(sqrt(1e60))
```

Tail recursion

Review of function application: one evaluates a function application `f(e1, ..., en)`.

- By evaluating the exressions `e1, ..., en` resulting in the values `v1, ..., vn` then
- By replacing the application with the body of `f`, in which
- The actual parameters `v1, ..., vn` replace the formal parameters of `f`

This rewriting rule, can be formalized:

```
def f(x1, ..., xn) = B; ... f(v1, ..., vn)
->
def f(x1, ..., xn) = B; ... [v1 / x1, ..., vn / xn] B

where [v1 / x1, ..., vn / xn] means
The expression B in which all ocurrences of xi have been replaced by vi.

[v1 / x1, ..., vn / xn] is called a substitution
```

Example:

```
def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

EXP: gcd(14, 21)
eval(EXP)
    if (21 == 0) 14 else gcd(21, 14 % 21)
    gcd(21, 14 % 21)
    gcd(21, 14)
    if (14 == 0) 21 else gcd(14, 21 % 14)
    gcd(14, 21 % 14)
    gcd(14, 7)
    if (7 == 0) 14 else gcd(7, 14 % 7)
    gcd(7, 14 % 7)
    gcd(7, 0)
    if (0 == 0) 7 else gcd(0, 7 % 0)
    7
```

Example:

```
def factorial(n: Int): Int =
    if (n == 0) 1 else n * factorial(n - 1)

EXP: factorial(4)
eval(EXP)
    if (4 == 0) 1 else 4 * factorial(4 - 1)
    4 * factorial(4 - 1)
    4 * factorial(3)
    ...
    4 * (3 * factorial(2))
    ...
    4 * (3 * (2 * factorial(1)))
    ...
    4 * (3 * (2 * (1 * factorial(0))))
    ...
    4 * (3 * (2 * (1 * 1)))
    4 * (3 * (2 * 1))
    4 * (3 * 2)
    4 * 6
    24
```

The evaluation of `gcd`, oscillates. The evaluation of `factorial` adds some value to the final expression, so it become bigger and bigger in every step of the recursion. That difference in the rewriting rules, translates directly on the execution on the computer. If you have a recursive function that calls itself as its last action, then you can reuse the stack frame of that function (**tail-recursion**). So it's really another fomulation of an iterative procedure. In the `factorial` function, the recursive call is not the last action. After the call to `factorial(n - 1)` we still need to multiply it by `n`. So that call, is not tail-recursive, we need to keep something until we reach the final value. Tail recursion avoid very deep recursive chains, avoiding stack overflow related errors. But a non-tail recursive, could be clearer than its tail recursive version.

(*Premature optimization is the root of all evil*) Donald E. Knuth

A tail recursive version of factorial

```
def factorial(n: Int) = {
    def iter(acc: Int, n: Int): Int =
        if (n == 0) acc
        else iter(acc * n, n - 1)
    iter(1, n)
}
```