Functional Programming Principles in Scala

Martin Odersky
Programming Paradigms

Paradigm: In science, a *paradigm* describes distinct concepts or thought patterns in some scientific discipline.

Main programming paradigms:

- imperative programming
- functional programming
- logic programming

Orthogonal to it:

- object-oriented programming
Imperative programming is about

- modifying mutable variables,
- using assignments
- and control structures such as if-then-else, loops, break, continue, return.

The most common informal way to understand imperative programs is as instruction sequences for a Von Neumann computer.
There’s a strong correspondence between

Mutable variables  \approx  memory cells
Variable dereferences  \approx  load instructions
Variable assignments  \approx  store instructions
Control structures  \approx  jumps

_Problem:_ Scaling up. How can we avoid conceptualizing programs word by word?

In the end, pure imperative programming is limited by the “Von Neumann” bottleneck:

*One tends to conceptualize data structures word-by-word.*

We need other techniques for defining high-level abstractions such as collections, polynomials, geometric shapes, strings, documents.

Ideally: Develop *theories* of collections, shapes, strings, ...
What is a Theory?

A theory consists of

- one or more data types
- operations on these types
- laws that describe the relationships between values and operations

Normally, a theory does not describe mutations!
Theories without Mutation

For instance the theory of polynomials defines the sum of two polynomials by laws such as:

\[(a\times x + b) + (c\times x + d) = (a+c)\times x + (b+d)\]

But it does not define an operator to change a coefficient while keeping the polynomial the same!
Theories without Mutation

For instance the theory of polynomials defines the sum of two polynomials by laws such as:

\[(a \times x + b) + (c \times x + d) = (a+c)\times x + (b+d)\]

But it does not define an operator to change a coefficient while keeping the polynomial the same!

Whereas in an imperative program one can write:

```java
class Polynomial { double[] coefficient; }
Polynomial p = ...;
p.coefficient[0] = 42;
```
Other example:

The theory of strings defines a concatenation operator $++$ which is associative:

$$(a ++ b) ++ c = a ++ (b ++ c)$$

But it does not define an operator to change a sequence element while keeping the sequence the same!

(This one, some languages do get right; e.g. Java’s strings are immutable)
Consequences for Programming

If we want to implement high-level concepts following their mathematical theories, there's no place for mutation.

- The theories do not admit it.
- Mutation can destroy useful laws in the theories.

Therefore, let's

- concentrate on defining theories for operators expressed as functions,
- avoid mutations,
- have powerful ways to abstract and compose functions.
Functional Programming

- In a *restricted* sense, functional programming (FP) means programming without mutable variables, assignments, loops, and other imperative control structures.
- In a *wider* sense, functional programming means focusing on the functions.
- In particular, functions can be values that are produced, consumed, and composed.
- All this becomes easier in a functional language.
In a *restricted* sense, a functional programming language is one which does not have mutable variables, assignments, or imperative control structures.

In a *wider* sense, a functional programming language enables the construction of elegant programs that focus on functions.

In particular, functions in a FP language are first-class citizens. This means:
- they can be defined anywhere, including inside other functions
- like any other value, they can be passed as parameters to functions and returned as results
- as for other values, there exists a set of operators to compose functions
Some functional programming languages

In the restricted sense:

- Pure Lisp, XSLT, XPath, XQuery, FP
- Haskell (without I/O Monad or UnsafePerformIO)

In the wider sense:

- Lisp, Scheme, Racket, Clojure
- SML, Ocaml, F#
- Haskell (full language)
- Scala
- Smalltalk, Ruby (!)
## History of FP languages

<table>
<thead>
<tr>
<th>Year</th>
<th>Language(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1959</td>
<td>Lisp</td>
</tr>
<tr>
<td>1975-77</td>
<td>ML, FP, Scheme</td>
</tr>
<tr>
<td>1978</td>
<td>Smalltalk</td>
</tr>
<tr>
<td>1986</td>
<td>Standard ML</td>
</tr>
<tr>
<td>1990</td>
<td>Haskell, Erlang</td>
</tr>
<tr>
<td>1999</td>
<td>XSLT</td>
</tr>
<tr>
<td>2000</td>
<td>OCaml</td>
</tr>
<tr>
<td>2003</td>
<td>Scala, XQuery</td>
</tr>
<tr>
<td>2005</td>
<td>F#</td>
</tr>
<tr>
<td>2007</td>
<td>Clojure</td>
</tr>
</tbody>
</table>
Recommended Book (1)


A classic. Many parts of the course and quizzes are based on it, but we change the language from Scheme to Scala.

The full text can be downloaded here.

A comprehensive step-by-step guide

Programming in

Scala

Second Edition

The standard language introduction and reference.
Other Recommended Books

The first part of “Scala for the Impatient” is available for free download.
Why Functional Programming?

Functional Programming is becoming increasingly popular, because it offers the following benefits.

▶ simpler reasoning principles
▶ better modularity
▶ good for exploiting parallelism for multicore and cloud computing.

To find out more, see the video of my 2011 Oscon Java keynote Working Hard to Keep it Simple (16.30 minutes).

The slides for the video are available separately.