Functional Programming Principles in Scala

Principles of Functional Programming

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

Review: Imperative 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.

Imperative Programs and Computers

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?

Reference: John Backus, Can Programming Be Liberated from the von. Neumann Style?, Turing Award Lecture 1978.

Scaling Up

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

```
class Polynomial { double[] coefficient; }
Polynomial p = ...;
p.coefficient[0] = 42;
```

Theories without Mutation

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 and immutable data.
- In particular, functions can be values that are produced, consumed, and composed.
- All this becomes easier in a functional language.

Functional Programming Languages

- 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 and immutable data structures.
- 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

- Lisp, Scheme, Racket, Clojure
- SML, Ocaml, F#
- Haskell
- Scala

By now, concepts and constructs from functional languages are also found in many traditional languages.

History of FP languages

1959	(Lisp)	2003	Scala
1975-77	ML, FP, Scheme	2005	F#
1978	(Smalltalk)	2007	Clojure
1986	Standard ML	2017	Idris
1990	Haskell, Erlang	2020	Scala 3
2000	OCaml		

Scala 3 is the language we will use in this course.

Origins of FP

1930s: Lambda Calculus (Alonzo Church)

- Shown to be equivalent to Turning Machines
- Stays relevant today as one of the theoretical foundations of FP

1959: Lisp

- Functions and recursive data tools for artifical intelligence research

1980/90s: ML, Haskell, ...

- New type systems with a strong connection to mathematical logic

Why Functional Programming?

- Reduce errors
- Improve modularity
- Higher-level abstractions
- Shorter code
- Increased developer productivity

Why Functional Programming Now?

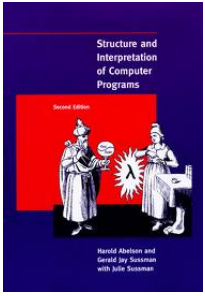
1. It's an effective tool to handle concurrency and parallelism, on every scale.
2. Our computers are not Van-Neuman machines anymore. They have
   - parallel cores
   - clusters of servers
   - distribution in the cloud

This causes new programming challenges such as

- cache coherency
- non-determinism

Recommended Book (1)

Structure and Interpretation of Computer Programs. Harold Abelson and Gerald J. Sussman. 2nd edition. MIT Press 1996.

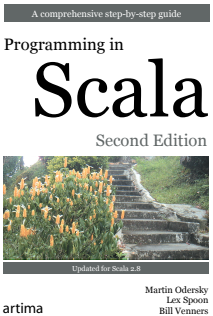


A classic. Many parts of the course and quizzes are based on it, but we change the language from Scheme to Scala.

The full text can be downloaded here.

Recommended Book (2)

Programming in Scala. Martin Odersky, Lex Spoon, and Bill Venners. 4th edition. Artima 2019.



The standard language introduction and reference.

Other Recommended Books

There areb many other good introductions to Scala. Among them: