Functional and Reactive Programming

Part 1: Functional Programming

- Dr. Herbert Prähofer
  Institute for System Software
  JKU Linz

JOHANNES KEPLER
UNIVERSITY LINZ

Lecturer

Computer Science
SYSTEM SOFTWARE

a.Univ.-Prof. Dipl.-Ing. Dr. Herbert Prähofer

Address: Altenberger Straße 69, 4040 Linz, Austria
Building: Computer Science Building (Science Park 3) Room: 205
Phone: + 43-732-2468-4352
Fax: + 43-732-2468-4345
Email: **herbert.praehofer@jku.at**

- Research and teaching background
  - software development
  - programming languages
  - functional programming
  - static and dynamic software analysis
  - software development and engineering methods in the automation domain

Literature

Funktionale Programmierung in Java

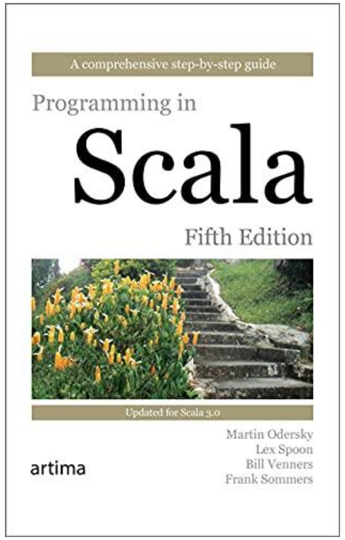
- published by dpunkt.verlag
- in German
- July 2020
- this course is based on parts of this book

currently I am working on an extended revised edition

Funktionale Programmierung
in Java und Kotlin

LITERATURE

- M. Odersky et al.: Programming in Scala, 5th Edition, 2021
  - from the developer of Scala



- P. Chuisano, R. Bjarnason: Functional Programming in Scala, 2015
  - purely functional programming in Scala

Schedule (Part 1 – FP)

Lecture 1

- Introduction
- Introduction to Scala
- Foundations of functional programming
- Functional data structures

Lab 1

Lecture 2

- Functional exception handling
- Reduction
- Function chaining

Lab 2

Lecture 3

- Function composition
- Lazy evaluation and streams
- Parallel streams

Lab 3

Abhaltung und Beurteilung

Übungen

- Während des Semesters werden Übungsangaben ausgeteilt, die zum Teil in den Übungsstunden gemacht werden und zum Teil als Hausübung auszuarbeiten sind.
- Die Teilnahme und Mitarbeit an den Übungsstunden ist verpflichtend
- Die Abgabe der Hausübungen ist verpflichtend

Klausur

- schriftliche Klausur am Semesterende
- theoretischer und praktischer Teil
  - praktischer Teil Stoff der Übungen
  - theoretischer Teil Stoff der Vorlesung

Benotung

- Aus der Beurteilung des theoretischen und praktischen Teils der Klausur wird die Gesamtnote ermittelt.
- Für die Vorlesung und die Übung wird dieselbe Note vergeben

Functional Programming

■ 1 Introduction

FUNCTIONAL PROGRAMMING (FP)

- **Functional programming (FP)** is programming with **mathematical functions**

$$x = y \quad \Rightarrow \quad f(x) = f(y)$$

Result only depends on values of arguments ➔ No side effects!!

- Pure functional programs exclusively consist of
  - **values** and **types**
  - **function definitions**
    - including recursive functions
  - **function application expressions**
  - **higher-order functions**, i.e., **functions as parameters** and **return values**
  - **function composition**, i.e., **creating functions from functions**

Functional Programming (FP)

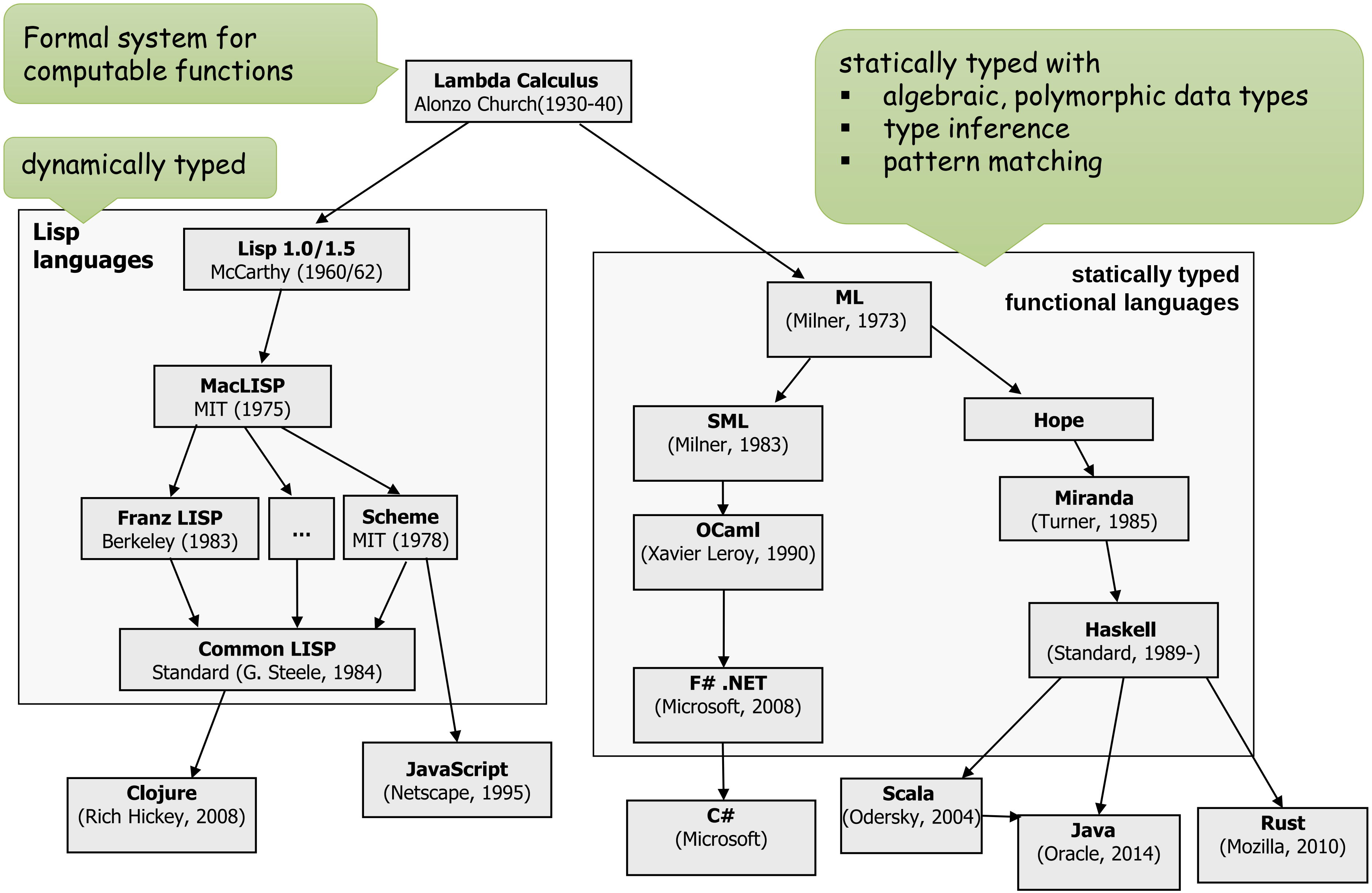
- In distinction to imperative and object-oriented programming
  - **no notion** of **value store**
  - **no pointers** or **references**
  - **no assignments** and **changes to memory**
  - **no side effects**
  - **no statements, no statement sequences**
  - **but only expressions**

Research in FP has shown, that programing only with functions
- **theoretically possible**
- **practically useful and relevant**

Lambda Kalkül

Pure functional languages (e.g. Haskell)

Development of Functional Languages



Imperative vs. Functional Style

Example: Inner product of two vectors a and b

- imperative solution

```
int c = 0;
for (int i = 0; i < n; i++) {
  c = c + a.get(i) * b.get(i);
}
```

- **mutable state**
- processing **single elements** one after the other

"word-at-a-time style of programming"

- functional solution

```
int c = map2((x, y) -> x * y).andThen(reduce((r, x) -> r + x))(a, b)
```

- **aggregate operations** which operate on vectors as a whole and not on single elements
- functions take **functions as parameters**
- **functions compose** to **more complex functions**

based on:
John Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, Aug. 1978. https://dl.acm.org/doi/10.1145/359576.359579.

John Backus: Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *CACM*, 1978

"Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive **word-at-a-time style of programming** inherited from their common ancestor–the von Neumann computer, their **close coupling of semantics to state transitions**, their **division of programming into a world of expressions and a world of statements**, their inability to effectively use **powerful combining forms for building new programs from existing ones**, and their lack of useful mathematical properties for reasoning about programs."

[...]

An alternative functional style of programming is founded on the **use of combining forms for creating programs**. […] Associated with the functional style of programming is an **algebra of programs whose variables range over programs and whose operations are combining forms**."

Benefits of Functional Programming

- Functional programming at a **higher, declarative level of abstraction**
- Functional programs are **less error-prone** and **more robust**
- Functional programs can be **verified more easily**
- Functional programs lend themselves for **parallel execution**
  - execution order of expressions is irrelevant for result
- Functional programming principles are **promising for concurrent and distributed** applications, e.g.,
  - Twitter message server
  - WhatsApp kernel

functional programming without side effects !

Concepts of Functional Programming

	supported in Java	*supported in Scala*
Programming with functions	***Yes***	***Yes***
Function objects and higher-order functions	***Yes***	***Yes***
Type inference	***Partly***	***Yes, not complete***
Algebraic data types	***Yes***	***Yes***
Polymorphic data types (generics)	***Yes, for reference types***	***Yes***
Pattern matching expressions	***Yes***	***Yes***
Non-strict evaluation (lazy evaluation)	***for streams***	***Yes***
Monads for function composition	***Yes***	***Yes***
Monad comprehension	***No***	***Yes***

Pure vs. Impure Functions

- Pure functions = only return values
  - no side effects
  - same argument values ➔ same results
  - not dependent on any (mutable) state
  - all what is relevant must be passed in arguments

- Impure functions = functions with side effects
  - two calls with same parameter may give different results
  - may depend on some (external) state

Example: Random Number Generators

Random number generation with java.util.Random

- same function call, different results ➔ **nextInt** is not a function (in the sense of FP)

it is not "pure" !

- function **nextInt** is **stateful**

```
val random = java.util.Random(977)

val rn1 = random.nextInt()
val rn2 = random.nextInt()
```

```
rn1 = -1223585132
rn2 = 1933351633
```

different results!

A pure function only uses parameter and return values and has no state and no side effect

- function **rand** takes a seed value and returns a random value plus a new seed

```
def rand(seed: Long) : (Int, Long) = {
  val i = (seed >>> 16).toInt
  val newSeed = (seed * 0x5DEECE66DL + 0xBL) & 0xFFFFFFFFFFFFL
  (i, newSeed)
}
```

```
val r1 = rand(12312341977L)
val r2 = rand(12312341977L)
```

```
(187871,5530422524784)
(187871,5530422524784)
```

```
val s0 = 12312341977L
val (i1, s1) = rand(s0)
val (i2, s2) = rand(s1)
```

```
(187871,5530422524784)
(84387550,186084096765627)
```

Advantages of Pure Functions

- **Parallel execution**
  - Expressions can be executed in **parallel**

- **Lazy evaluation (on-demand execution)**
  - Expressions can be evaluated when result **is needed**

- **Testability**
  - Pure functions are independent of others and can be **tested independently**

- **Compositibilty**
  - Functions are **composable**

- **Memoization**
  - Once computed for an specific argument value, the value **can be cached** for later use

Loops vs. Recursive Functions

Functions with loops mutate variables and thus has side effects

- Example faculty: Imperative solution with mutable variables f and i

```
def facIter(n: Int) : Int = {
  var f = 1
  for (i <- 2 to n) {
    f = f * i
  }
  f
}
```

Pure functions use recursion instead of loops

- recursive solution has no mutable variable

```
def facRec(n: Int) : Int = {
  if (n == 1) then 1
  else n * facRec(n - 1)
}
```

<u>However:</u>
from outside are both pure functions
because mutable variables f and i are local
and do not have a side effect visible outside
➔ local mutable variables are often used in Scala
for performance reasons

Referential Transparency

An important property of an expression is

Referential Transparency

*An expression is **referential transparent** if the value of an expression which contains **sub-expressions** is **ONLY DEPENDENT** on the **values** of the sub-expressions!*

Any other property such as

- its internal structure,
- the number and nature of its components,
- the order in which they are evaluated

are irrelevant

!

From left to right,
or right to left
or in parallel

```
val r = facIter(5) - facIter(7)
```

referential transparent

```
val r2 = random.nextInt() - random.nextInt();
```

not referential transparent

Remarks on FP in Scala

- Scala is a pure object-oriented programming language
- Scala runs on the Java VM
  - which is not a functional execution engine
- not purely functional
  - allows side-effects

But

- Scala has full implementation of functional principles (from Haskell)
- Scala allows pure functional programming
- Scala libraries are (mainly) built on functional principles
  - often with functional external and imperative internally (for performance reasons)