Functional programming

theory and practice (Scala)

Martin Konvicka

May 13, 2024

Programming paradigms

Code structure	Procedural	Object-oriented
State handling	Imperative	Functional

Table: Programming paradigms – overview

- Procedural – Separation of procedures/function from data.
- Object-oriented – Close coupling of functions with data.
- Imperative – Functions may depend on and modify external state.
- Functional – Functions don't modify or depend on external state.*

Turing machine

- Alan Turing (1912 – 1954)
- State-based model of computation
- Many equivalent variations

$$TM = \{Q, \Sigma, \Gamma, q_0, \delta$$
$$q_{accept}, q_{reject}\}, \text{ where}$$

Q – finite set of states

Σ – input alphabet

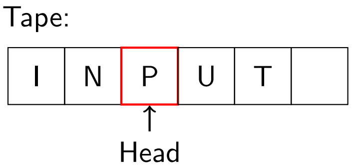
Γ – tape alphabet

$q_0 \in Q$ – start state

$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$

q_{accept} – accept state

q_{reject} – reject state



Lambda calculus

- Alonzo Church (1903 – 1995)
- Functional model of computation

1. Variables
2. Way of building functions
3. Way of applying functions to arguments

$$\lambda x.\ \lambda y.\ x + y$$

$$(\lambda x.\ \lambda y.\ x + y)\ 2\ 3$$

$$TRUE = \lambda x.\ \lambda y.\ x$$

$$FALSE = \lambda x.\ \lambda y.\ y$$

$$NOT = \lambda b.\ b\ FALSE\ TRUE$$

$$Y = \lambda f.\ (\lambda x.\ f(x\ x))\ (\lambda x.\ f(x\ x))$$

Functional programming – the essence

- Pure functions – fully dependent on it's arguments
- First class functions and higher-order functions
- Immutable data structrues
- Referential transparency (Substitution model)

Functional programming – benefits

- Easier testing
- Memoization
- HOF – abstractions and code reuse
- Immutable data structures – concurrency, state navigation
- Modularity and expressiveness
- Intellectual enlightenment

Substitution model

```
def f(x1, x2, ..., xn) = B
f(e1, e2, ..., en) =>
f(v1, v2, ..., vn) =>
[v1/x1, v2/x2, ..., vn/xn]B

def sqr(x: Int) = x * x
def sqr2(x: => Int) = x * x
```

Call-by-value

```
sqr(2 * 3 + 6) =>
sqr(6 + 6) =>
sqr(12) =>
12 * 12 =>
144
```

Call-by-name

```
sqr2(2 * 3 + 6) =>
(2 * 3 + 6) * (2 * 3 + 6) =>
(6 + 6) * (2 * 3 + 6) =>
12 * (2 * 3 + 6) =>
12 * (6 + 6) =>
12 * 12 =>
144
```

Referential transparency

Expressions yielding the same value are interchangeable.

```
def sqr3(x: Int) =
  print(x * x) // side-effect
  x * x

val s = sqr3(2) // prints 4
s + s == 8

sqr3(2) + sqr3(2) == 8 // prints 44
```

Stack recursion

Defining functions in terms of themselves.

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

/*
    factorial(4) =>
    if 4 == 0 then 1 else 4 * factorial(4 - 1) =>
    4 * factorial(3) =>
    4 * if 3 == 0 then 1 else 3 * factorial(3 - 1) =>
    4 * 3 * factorial(2) =>
    4 * 3 * if 2 == 0 then 1 else 2 * factorial(2 - 1) =>
    4 * 3 * 2 * factorial(1) =>
    4 * 3 * 2 * if 1 == 0 then 1 else 1 * factorial(1 - 1) =>
    4 * 3 * 2 * 1 * factorial(0) =>
    4 * 3 * 2 * 1 * if 0 == 0 then 1 else 0 * factorial(0 - 1) =>
    4 * 3 * 2 * 1 * 1 =>
    24
*/
```

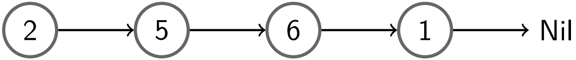
Tail recursion

```
def factorialTail(n: Int): Int =
  @tailrec
  def factorialIter(n: Int, acc: Int): Int =
    if n == 1 then acc
    else factorialIter(n - 1, n * acc)

  if n == 0 then 1
  else factorialIter(n, 1)

/*
    factorialTail(4) =>
    if 4 == 0 then 1 else factorialIter(4, 1) =>
    factorialIter(4, 1) =>
    if 4 == 1 then acc else factorialIter(4 - 1, 4 * 1) =>
    factorialIter(3, 4) =>
    if 3 == 1 then acc else factorialIter(3 - 1, 3 * 4) =>
    factorialIter(2, 12) =>
    if 2 == 1 then acc else factorialIter(2 - 1, 2 * 12) =>
    factorialIter(1, 24) =>
    if 1 == 1 then acc else factorialIter(1 - 1, 1 * 24) =>
    24
*/
```

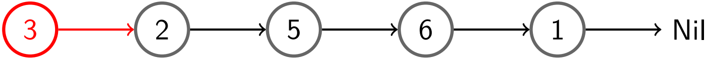
Immutable DS – List

```scala
sealed abstract class List[+A]
final case class :: [+A](head: A, next: List[A])
    extends List[A]
case object Nil extends List[Nothing]
```

```scala
val aList = 2 :: 5 :: 6 :: 1 :: Nil
```

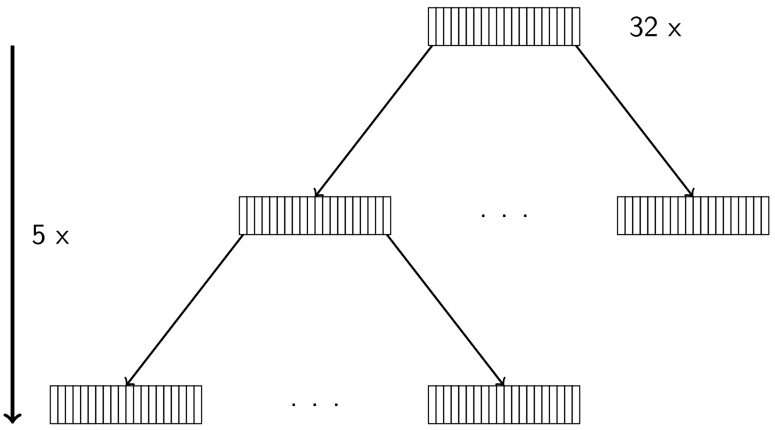



```scala
3 :: aList
```

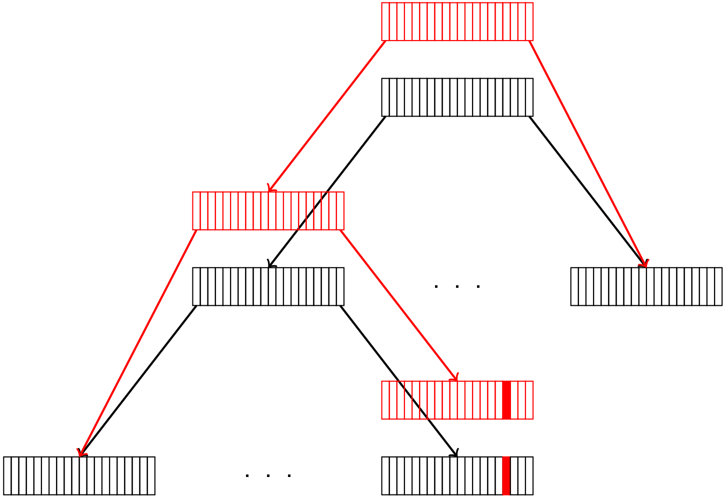



```scala
aList :+ 3
```

Immutable DS – Vector



Immutable DS – Vector (Modification)



Monads

- parametric type M[T] with two operations

```
trait M[T]:
  def unit[T](x: T): M[T]
  def flatMap[U](f: T => M[U]): M[U]
```

- has to satisfy 3 algebraic laws
  1. Associativity:
     `m.flatMap(f).flatMap(g) == m.flatMap(f(_).flatMap(g))`
  2. Left unit:
     `unit(x).flatMap(f) == f(x)`
  3. Right unit:
     `m.flatMap(unit) == m`

- List, Set, Option, Future, Try*

Algebraic data types (ADT)

- DT – set of values
- Definition of composite types
  - Sum

```
sealed abstract class Role

case class Student(id: Option[Long], user: User, course: Course) extends Role
case class Instructor(id: Option[Long], user: User, course: Course) extends Role
```

  - Product

```
case class User(
    id: Option[Long],
    username: String,
    givenName: String,
    familyName: String,
    password: String,
    email: Email,
)
```

Subtype polymorphism

```
abstract class Animal:
    def speak(): String

class Cat extends Animal:
    override def speak(): String = "MEOW !!!"
class Dog extends Animal:
    override def speak(): String = "HOOF !!!"


def main(args: Array[String]): Unit =
    val animal: Animal = Dog()
    println(animal.speak())                 // HOOF

    val aDifferentAnimal: Animal = Cat()
    println(aDifferentAnimal.speak())       // MEOW
```

Parametric polymorphism

```scala
def map[T, R](elements: List[T], f: (T => R)): List[R] = elements match
case Nil => Nil
case head :: tail => f(head) :: map(tail, f)

def main(args: Array[String]): Unit =
    val numbers = List(1, 2, 3, 4, 5)
    val increment = (a: Int) => a + 1
    val numbersIncremented = map(numbers, increment)  // List(2, 3, 4, 5, 6)

    val letters = List('a', 'b', 'c', 'd', 'e')
    val upperCase = (a: Char) => a.toUpper
    val lettersUpperCased = map(letters, upperCase)   // List(A, B, C, D, E)
```

Ad-hoc polymorphism – Type classes

1. Define a generic abstract trait (new functionality)
2. Implement the trait for the appropriate data types
3. Create an API utilizing the type classes (generic, implicit)
4. Optionally, define extension methods for the data types

Contextual abstractions

- Term inference
- A way of accessing context in functions
- Implicit values

Contextual abstractions – example

```scala
final case class Person(name: String, age: Int)

object Ordering {
    trait Ordering[A]:
        def lessThan(first: A, second: A): Boolean

    given intOrdering: Ordering[Int] = new Ordering[Int]:
        def lessThan(first: Int, second: Int): Boolean = first < second

    given Ordering[Person] with
        def lessThan(first: Person, second: Person): Boolean =
            intOrdering.lessThan(first.age, second.age)

    def sort[T](elements: List[T])(using Ordering[T]): List[T] =
    //       ^ type inference       ^^^^^^^^^^^^^^^^^^ term inference
        def merge(left: List[T], right: List[T]): List[T] = ???
            // ............ summon[Ordering[T]].lessThan ............

        if elements.size <= 1 then elements
        else
            val (left, right) = elements.splitAt(elements.length / 2)
            val leftSorted = sort(left)
            val rightSorted = sort(right)
            merge(leftSorted, rightSorted)
}
```

Functional stacks

Typelevel

- Collection of libraries
- Cats, Cats Effect, ...
- IO class
- Tagless final

ZIO

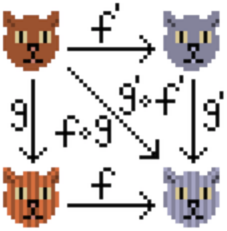
- Single library
- ZIO class
- ???

Typelevel libraries

Name	Description
Cats	Abstractions for functional programming
Cats Effect	Pure asynchronous runtime
FS2	Concurrent streams
Doobie	Pure functional JDBC layer
Http4s	Functional HTTP library
Circe	JSON serialization
Tsec	Cryptographic algorithms

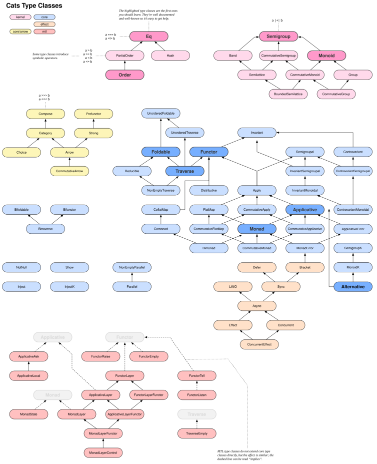
Cats

- Category theory 😅
- Abstractions for functional programming
  - Type classes
  - Data types
- Algebraic rules (associativity, commutativity, identity, ...)
  - Property-based testing (ScalaCheck)
- Backbone of the entire Typelevel ecosystem



Cats – Type classes

Semigroup, Monoid, Functor,
Monad, Semigroupal,
Applicative, Foldable,
Traverse, ...



cats-infographic 1.2 for cats 1.3.0, cats-effect 1.0.0, and cats-mtl 0.2.3 • Source available at github.com/tpolecat/cats-infographic • Shareable under CC-BY-SA 4.0

```scala
trait Semigroup[A]:
  def combine(x: A, y: A): A

trait Functor[F[_]]:
  def map[A, B](fa: F[A])(f: A => B): F[B]
  def lift[A, B](f: A => B): F[A] => F[B] =
    fa => map(fa)(f)

trait Applicative[F[_]] extends Functor[F]:
  def ap[A, B](ff: F[A => B])(fa: F[A]): F[B]
  def pure[A](a: A): F[A]
  def map[A, B](fa: F[A])(f: A => B): F[B] =
    ap(pure(f))(fa)

trait Foldable[F[_]]:
  def foldLeft[A, B](fa: F[A], b: B)(f: (B, A) => B): B
  def foldRight[A, B](fa: F[A], lb: Eval[B])(f: (A, Eval[B]) => Eval[B]): Eval[B]
```

Cats – Data types

DT	Use case
Kleisli[F[_], -A, B]	A => F[B]
Reader[A, B] (Kleisli[Id, -A, B])	dependency injection
Writer[L, A]	logging
Eval[A+]	stack-safe computation
State[S, A]	application state S => (S, A)
Validated[E, R]	data validation

```scala
case request@GET -> Root / "courses" asAuthed user =>
  for
    coursesStudent <- getCoursesWithUserAs(user.id, Role.Student).transact(xa)
    coursesTeacher <- getCoursesWithUserAs(user.id, Role.Instructor).transact(xa)
    courses = WebCourses(
      coursesStudent.map(WebCourse(_)).collect { case Some(c) => c },
      coursesTeacher.map(WebCourse(_)).collect { case Some(c) => c }
    )
    response <- Ok(courses, `Content-Type`(MediaType.application.json))
  yield response
```

Validated – example

```scala
private def validateMaxSubmissions(attempt: SubmissionAttempt): Validated[List[String], SubmissionAttempt] =
  attempt.assignment.maxSubmissions match
    case Some(max) if attempt.assignment.submitted >= max =>
      invalid(List("Max submissions reached for this assignment"))
    case _ => valid(attempt)

private def validateDueDate(attempt: SubmissionAttempt): Validated[List[String], SubmissionAttempt] =
  attempt.assignment.due match
    case Some(due) if ZonedDateTime.now.isAfter(due) =>
      invalid(List(f"Due date [$due] passed for this assignment"))
    case _ => valid(attempt)

private def validateLoadSubmission(attempt: SubmissionAttempt, to: Path)
: IO[Validated[List[String], SubmissionAttempt]] =
  for
    result <- getLoader(attempt.location)(to)
    a <- result match
      case LoaderResult(_, true, _) => IO(Validated.valid(attempt))
      case _ => IO(invalid(List(f"Could not load submission from ${attempt.location}")))
  yield a

private def canSubmit(attempt: SubmissionAttempt, to: Path): IO[Validated[List[String], SubmissionAttempt]] =
  val shouldTryLoad = (validateMaxSubmissions(attempt), validateDueDate(attempt)).mapN((_, _) => attempt)
  shouldTryLoad match
    case Valid(a) => validateLoadSubmission(a, to).map(vls => (vls, shouldTryLoad).mapN((_, _) => attempt))
    case Invalid(errors) => IO(invalid(errors))
```

IO (Cats Effect)

- Wraps an arbitrary expression yielding a value of type T
- Separation of a definition from its execution
- The final expression is evaluated at the end

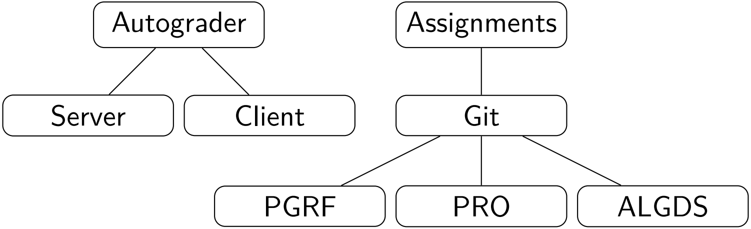
```
case class IO[T](unsafeRun: => T) extends M[T]
```

```
def sourceFile(name: String): Resource[IO, BufferedSource] = ???
def readLines(source: BufferedSource): IO[List[String]] = ???
def printLines(lines: List[String]): IO[Unit] = ???
def parseLines(lines: List[String]): List[Try[(User, String)]] = ???

val userFile = sourceFile(path)
userFile.use(readLiner)
  .flatMap(lines => printLines(lines)
    .flatMap(_ => parseLines(lines)))

for
  userFile = sourceFile(path)
  lines <- fileWithUsers.use(readLines)
  _ <- printLines(lines)
  users = parseLines(lines)
  ...
yield ()
```

Project Autograder

- Automatic generation (from templates)
- Automatic evaluation (ScalaCheck – referential implementations)
- Web interface
- Data persistence (PostgreSQL)

System Architecture



Scala

- JVM, JS, Native
- Functional, Object-oriented
- Strong, static typing
- Expressiveness (implicits, macros)
- Slow autocomplete and phantom errors in IntelliJ (Metals seems to offer some improvements)

Sources

- Functional Programming in Scala Specialization (Coursera)
- RockTheJVM
- Scala, Typelevel documentation