

What did functional programming ever do for us (software engineers)?

An extreme pragmatic and un-academic approach

Sergei Winitzki

Academy by the Bay

2020-08-22

Overview: Advantages of functional programming

Features of functional programming are being added to many languages

- What are these features?
- What advantages do they give programmers?

Programmer-facing features listed in the increasing order of complexity
(and decreasing order of advantage-to-cost ratio)

- 1 Write iterative code without loops
- 2 Use functions parameterized by types, checked by compiler
- 3 Use disjunctive types to model special cases, errors, etc.
- 4 Use special syntax for chaining effectful computations

Feature 1: Loop-free iterative programs. Transformation

An easy interview question:

Given an array of integers, find all pairs that sum to a given number

- Solutions using loops: 10-20 lines of code
- FP solution with $O(n^2)$ complexity:

```
# Python
[ (x, y) for x in array for y in array if x + y == n ]
// Scala
for { x <- array; y <- array; if x + y == n } yield (x, y)
-- Haskell
do; x <- array; y <- array; guard (x + y == n); return (x, y)
```

- FP solution with $O(n \log n)$ complexity:

```
// Scala
val hash = array.toSet
for { x <- hash; if hash contains (n - x) } yield (x, n - x)
```

Feature 1: Loop-free iterative programs. Aggregation

- Given an array of integers, compute the sum of square roots of all elements except negative ones

Python

```
sum( [ sqrt(x) for x in givenArray if x >= 0 ] )
```

// Scala

```
givenArray.filter(_ >= 0).map(math.sqrt).sum
```

- Given a set of integers, compute the sum of those integers which are non-negative and whose square root is also present in the set

- ▶ Solution using loops: 15-20 lines of code

- ▶ FP solution:

givenSet // Scala

```
.filter { x => x >= 0 && givenSet.contains(math.sqrt(x)) }  
.sum
```

- Compute a positive integer from a given array of its decimal digits

digits.reverse.zipWithIndex // Scala

```
.map { case (d, i) => d * math.pow(10, i).toInt } .sum
```

Feature 1: Loop-free iterative programs. Induction

- Compute the mean value of a given sequence in single pass

```
scala> Seq(4, 5, 6, 7).foldLeft((0.0, 0.0)) {  
  | case ((sum, count), x) => (sum + x, count + 1)  
  | }.pipe { case (sum, count) => sum / count }  
res1: Double = 5.5
```

- Any inductive definition can be converted to a “fold”
 - ▶ Base case is the initial value
 - ▶ Inductive step is the updater function

Feature 1: Loop-free iterative programs. Other applications

- The implementation of `map`, `filter`, `fold`, `flatMap`, `groupBy`, `sum`, etc., may be asynchronous (Akka Streams), parallel, and/or distributed (Spark, Flink)
 - ▶ The programmer writes loop-free code in the map/filter/reduce style
 - ▶ The runtime engine implements parallelism, fault tolerance, etc.
 - ▶ Many types of programmer errors are avoided automatically
- What we need to support programming in the map/filter/reduce style:
 - ▶ Collections with user-definable methods
 - ▶ Functions (“lambdas”) passed as parameters to other functions
 - ▶ Easy work with tuple types
- Lambdas were added to most programming languages by 2015

Feature 2: Type parameters. Usage

- Collections can have types parameterized by element type
 - ▶ Array of integers: `Array[Int]`
 - ▶ Array of strings: `Array[String]`
 - ▶ Array of arrays of pairs of integers and strings: `Array[Array[(Int, String)]]`
- Methods such as `map`, `zip`, `flatMap`, `groupBy` change the type parameters

```
Seq(4, 5, 6, 7)                // Seq[Int]
  .zip(Seq("a", "b", "c", "d")) // Seq[(Int, String)]
  .map { case (i, s) => s"$s : $i" } // Seq[String]
```
- The compiler prevents using incorrect type parameters anywhere

Feature 2: Type parameters. Language features

- Code can be written once, then used with different type parameters
- Example (Scala):

```
def map[A, B](xs: Seq[A])(f: A => B): Seq[B] = ...
```
- Collections (`Array`, `Set`, etc.) with type parameters support such methods
- Many programming languages have type parameters
 - ▶ Functions with type parameters were added to Java in 2006
 - ▶ C++ can imitate this functionality with templates
 - ▶ Go-lang might get type parameters in the future

Feature 3: Disjunctive types

- Enumeration type (`enum`) describes a set of disjoint possibilities:

```
enum Color { RED, GREEN, BLUE; } // Java
```

- A value of type `Color` can be only one of the three possibilities
- Disjunctive types are “enriched” `enum` types, carrying extra values:

```
// Scala 3
```

```
enum RootOfEq { case NoRoots(); case OneRoot(x: Float); }
```

- The `switch` is “enriched” to extract data from disjunctive values

```
// Scala
```

```
roots match {  
  case OneRoot(x) => // May use 'x' in this expression.  
  case NoRoots() => // May not use 'x' here by mistake.  
}
```

- Disjunctive types describe values from “tagged union” sets
 - ▶ `OneRoot(x)` \cong the set of all `Float` values
 - ▶ `NoRoots()` \cong the set consisting of a single value
 - ▶ `RootOfEq` \cong either some `Float` value or the special value `NoRoots()`

Feature 3: Disjunctive types. Adoption in languages

- Disjunctive types and pattern matching are required for FP
- Introduced in Standard ML (1973)
- Supported in all FP languages (OCaml, Haskell, F#, Scala, Swift, ...)
- The support of disjunctive types only comes in FP-designed languages
 - ▶ Not supported in C, C++, Java, JavaScript, Python, Go, ...
 - ▶ Not supported in relational languages (Prolog, SQL, Datalog, ...)
 - ▶ Not supported in configuration data formats (XML, JSON, YAML, ...)
- Logical completeness of the type system:

Scala type	Logic operation	Logic notation	Type notation
(A, B)	conjunction	$A \wedge B$	$A \times B$
<code>Either[A, B]</code>	disjunction	$A \vee B$	$A + B$
<code>A => B</code>	implication	$A \Rightarrow B$	$A \rightarrow B$
<code>Unit</code>	true	\top	1
<code>Nothing</code>	false	\perp	0

- Programming is easier in languages having a *complete* logic of types
 - ▶ “Hindley-Milner type system”

Feature 4: Chaining of effects. Motivation

How to compose computations that may fail with an error?

- A “result or error” disjunctive type: `Try[A] \cong Either[Throwable, A]`
- In Scala, `Try[A]` is a disjunction of `Failure[Throwable]` and `Success[A]`

Working with `Try[A]` requires two often-used code patterns:

- Use `Try[A]` in a computation that cannot fail, `f: A => B`
 - ▶ `Success(a)` goes to `Success(f(a))` but `Failure(t)` remains unchanged
- Use `Try[A]` in a computation that can fail, `g: A => Try[B]`
 - ▶ `Success(a)` goes to `g(a)` while `Failure(t)` remains unchanged

Feature 4: Chaining of effects. Implementation

Implementing the two code patterns using pattern matching:

- Pattern 1: use `Try[A]` in a computation that cannot fail

```
tryGetFileStats() match {  
  case Success(stats) => Success(stats.getTimestamp)  
  case Failure(exception) => Failure(exception)  
}
```

- Pattern 2: use `Try[A]` in a computation that can fail

```
tryOpenFile() match {  
  case Success(file) => tryRead(file) // Returns Try[Array[Byte]]  
  case Failure(exception) => Failure(exception)  
}
```

Feature 4: Chaining of effects. Implementation

- The two patterns may be combined at will

```
// Read a file, decode UTF-8, return the number of chars.  
def utfChars(name: String): Try[Int] = tryOpenFile(name) match {  
  case Success(file) => tryRead(file) match {  
    case Success(bytes) => tryDecodeUTF8(bytes) match {  
      case Success(decoded) => Success(decoded.length)  
      case Failure(exception) => Failure(exception)  
    }  
    case Failure(exception) => Failure(exception)  
  }  
  case Failure(exception) => Failure(exception)  
}
```

- The code is awkwardly nested and repetitive

- ▶ This sort of code is common in go-lang programs:

```
err1, res1 := tryOpenFile();  
if (res1 != nil) {  
  err2, res2 := tryRead(res1);  
  if (res2 != nil) { ... } // Continue with no errors.  
  else ... // Handle second error.  
else ... // Handle first error.
```

Feature 4: Chaining of effects. Using map and flatMap

Implement the two code patterns using `map` and `flatMap`:

- `Try(a).map(f)` – use with `f: A => B` that cannot fail
- `Try(a).flatMap(g)` – use with `g: A => Try[B]` that can fail

```
def fmap[A, B](f: A => B): Try[A] => Try[B]
def flm[A, B](g: A => Try[B]): Try[A] => Try[B]
```

- Pattern 1: use `Try[A]` in a computation that cannot fail

```
tryGetFileStats() match {
  case Success(stats) => Success(stats.getTimestamp)
  case Failure(exception) => Failure(exception)
}
```

- Pattern 2: use `Try[A]` in a computation that can fail

```
tryOpenFile() match {
  case Success(file) => read(file) // Returns Try[InputStream]
  case Failure(exception) => Failure(exception)
}
```

Feature 4: Chaining of effects. Special syntax

Using `map` and `flatMap`, the code from the previous slide simplifies:

```
tryGetFileStats().map(stats => stats.getTimestamp)  
tryOpenFile().flatMap(file => read(file))
```

- We can now write code like this (avoiding nesting and repetition):

```
Try(one()).map { x => f(x) }.flatMap { y => Try(another(y)) }
```

- Instead of a chain of `map` and `flatMap` methods, use a special syntax:

```
for {  
  x <- Try(one())  
  y = f(x)  
  z <- Try(another(y))  
} yield z
```

- Resembles the syntax for nested loops (“for-comprehension”)

```
for { x <- list1; y <- list2 } yield p(x, y) // Scala  
[ z(x, y) for y in list2 for x in list1 ] // Python
```

- In Haskell: “do notation”, in F#: “computation expressions”

Feature 4: Chaining of effects. Special syntax

Using the special syntax, a chain of computations looks like this:

```
// Read a file, decode UTF-8, return the number of chars.  
def utfChars(name: String): Try[Int] = for {  
  file    <- tryOpenFile(name)  
  bytes   <- tryRead(file)  
  decoded <- tryDecodeUTF8(bytes)  
  length  = decoded.length  
} yield length
```


Features 1-4 may be combined at will

The main advantages in practical coding come from combining features 1-4 of functional programming

- Use functional methods on collections with type parameters
- Use disjunctive types to represent program requirements
- Avoid explicit loops and make sure all types match
- Compose programs at high level by chaining effects (`Try`, `Future`, ...)

- FP proposes 4 essential features that make programming easier
 - ▶ loop-free iteration, type parameters, disjunctive types, chaining syntax
 - ▶ other, more advanced features have lower cost/advantage ratios
- All “FP languages” have these features and give the same advantages
 - ▶ OCaml, Haskell, Scala, F#, Swift, Rust, Elm, ...
- Most “non-FP languages” lack at least 2 of these features
 - ▶ Lack of features may be compensated but raises the cost of their use