Programowanie funkcyjne

Patryk Gronkiewicz

KN Machine Learning

2022-12-06

Co to jest to cale FP?

Definicja

Programowanie funkcyjne jest paradygmatem programowania, gdzie programy są tworzone przez stosowanie i składanie funkcji. Jest deklaratywnym paradygmatem programowania w którym definicje funkcji są drzewem wyrażeń, które mapuje wartości na inne wartości — w przeciwieństwie do sekwencji imperatywnych wyrażeń aktualizujących stan programu¹

Pojawia się tu dużo ciężkich określeń, jednak zacznijmy od początku.

¹https://en.wikipedia.org/wiki/Functional_programming (tłum. własne)

Co to jest to cale FP?

Definicja

Programowanie funkcyjne jest paradygmatem programowania, gdzie programy są tworzone przez stosowanie i składanie funkcji. Jest deklaratywnym paradygmatem programowania w którym definicje funkcji są drzewem wyrażeń, które mapuje wartości na inne wartości — w przeciwieństwie do sekwencji imperatywnych wyrażeń aktualizujących stan programu¹

Pojawia się tu dużo ciężkich określeń, jednak zacznijmy od początku.

¹https://en.wikipedia.org/wiki/Functional_programming (tłum. własne)

Paradygmat

Jest to "styl" pisania kodu, czasem wymuszony przez język programowania. Angielska Wikipedia wyróżnia ich 74 (sic!). Nas w tym momencie interesują dwa (i pół): funkcyjny i imperatywny (oraz pochodna tego drugiego — obiektowy). Definicję paradygmatu funkcyjnego już widzieliśmy, natomiast bez takich formalności — paradygmat imperatywny to "typowe" pisanie kodu znane z Pythona, C++ czy Javy.

```
Programming paradigms
```

Czyste funkcje (pure functions)

Czyste funkcje to takie, które:

Dla dokładnie takich samych argumentów zwracają dokładnie takie same wartości;

Nie mają skutków ubocznych.

W Scali wszystkie funkcje, które interfejsują się z czymkolwiek poza innymi funkcjami nie są czyste — np. println ma skutek uboczny (wypisanie na konsoli).

```
// pure function
val pureDouble = (x: Int) => x * 2
// impure function
val impureDouble = (x: Int) => {
  val newX = x * 2
  println(newX)
  newX
}
fn(4); gn(4)
```

Pętle



Rekurencja

Żeby zrozumieć rekurencję

Musisz zrozumieć rekurencję

Żeby zrozumieć rekurencję

Musisz zrozumieć rekurencję

Żeby zrozumieć rekurencję

Musisz zrozumieć rekurencję

Rekurencja



NO INFINITE RECURSION





NOUNFINITE REGURSION





Rekurencja ogonowa (tail recursion)

Pozwala ona na zastosowanie optymalizacji — kompilator zamiast zagnieżdżać kolejne wywołania "spłaszcza" tę strukturę i pozbywa się rekurencji.

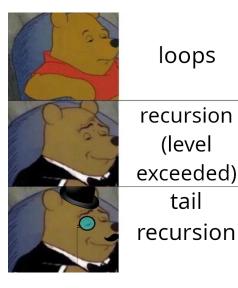
Zwykła rekurencja

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

Rekurencja ogonowa

```
@tailrec
def factorial_tr(n: Int, acc: Int = 1): Int =
   if n == 0 then acc
   else factorial_tr(n - 1, acc * n)
```

Rekurencja ogonowa (tail recursion)



Funkcje anonimowe (tzw. lambdy)

Funkcje, które żyją tak krótko, że nie nadajemy nawet im nazwy — są to tzw. lambdy. W scali mają składnię (arg1, arg2, ...) => res. Jeśli jest tylko jeden argument — nawias można pominąć. Czasem występuje jeszcze prostszy zapis z podkreśleniami (Funkcje niżej są jednoznaczne).

```
(arg1: Int, arg2: Int) = arg1 * arg2
- * -
```

Warto zauważyć, że w drugim przypadku w obu sytuacjach argumentem jest podkreślenie — po każdym jej wystąpieniu brany jest następny argument.

Funkcje wyższego rzędu (Higher Order Functions, HOF)

Funkcje przyjmujące inne funkcje jako argumenty.

```
val greater_than_1: Int => Boolean = (x: Int) => x > 1
val double: Int => Int = (x: Int) => x * 2
val sum: (Int, Int) => Int = (x: Int, y: Int) => x+y
List(1,2,3).filter(_>1).map(_*2).reduce(_+_)
List(1,2,3).filter(greater_than_1).map(double).reduce(sum)
```

Teoria teorią, ale do czego to się przydaje?

- W data science! Cały Spark opiera się przede wszystkim na HOF. To samo Pandas/Pola.rs/inne biblioteki do ramek danych. No i cały język R.
- W strumieniowym przetwarzaniu danych.
- Przy odtwarzaniu rzeczy dużo łatwiej doprowadzić program do tego samego stanu mając ściśle zdefiniowane transformacje.