Functional programming

TRAPS & FREAKS

~~TIPS & TRICKS~~

2018

by Roman Stranchewsky

About myself

- Roman Stranchewsky

Job:

- Android developer @Citadele

Education:

- Computer Science @TSI (Past)
- Bioinformatics @LU (Present)

Agenda

- What is FP once again
- Recursive trap
- Lazy trap
- Slow trap
- Lambdas

with examples and demo

Functional programming

~~TRAPS & FREAKS~~

~~TIPS & TRICKS~~

~~2016~~

by Roman Stranchewsky

The paradigm

- Computation is evaluation of mathematical functions
- No changing state
- No mutable data
- Programming is done with expressions or declarations

No state = no loops

```
fun loopPrint(audience: List<String>) {
    for (listener in audience) {
        println(listener)
    }
}
```


iterator is mutable

Loops vs. recursion

- Every loop can be replaced with recursion

```
fun recursivePrint(audience: List<String>) {
    if (audience.isNotEmpty()) {
        println(head(audience))
        recursivePrint(tail(audience))
    }
}
```


In pure functional PLs, such as Haskell, lists are abstracted as head and tail

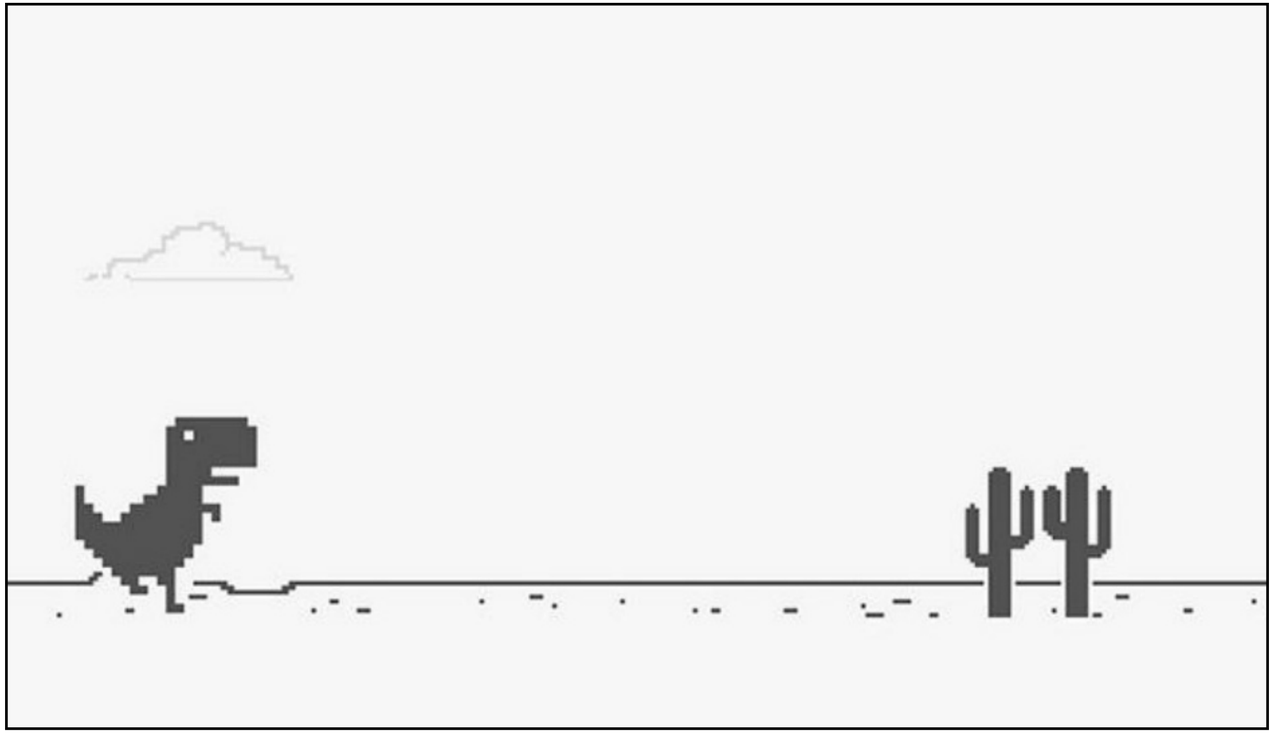
```
greet [] = []
greet (person:persons) = ("Hello, " ++ person ++ "!") :
  greet(persons)
```

Beaty of recursion

```
fun fibonacci(n: Int): Int {
    return when (n) {
        0, 1 -> n
        else -> fibonacci( n: n - 1) + fibonacci( n: n - 2)
    }
}
```

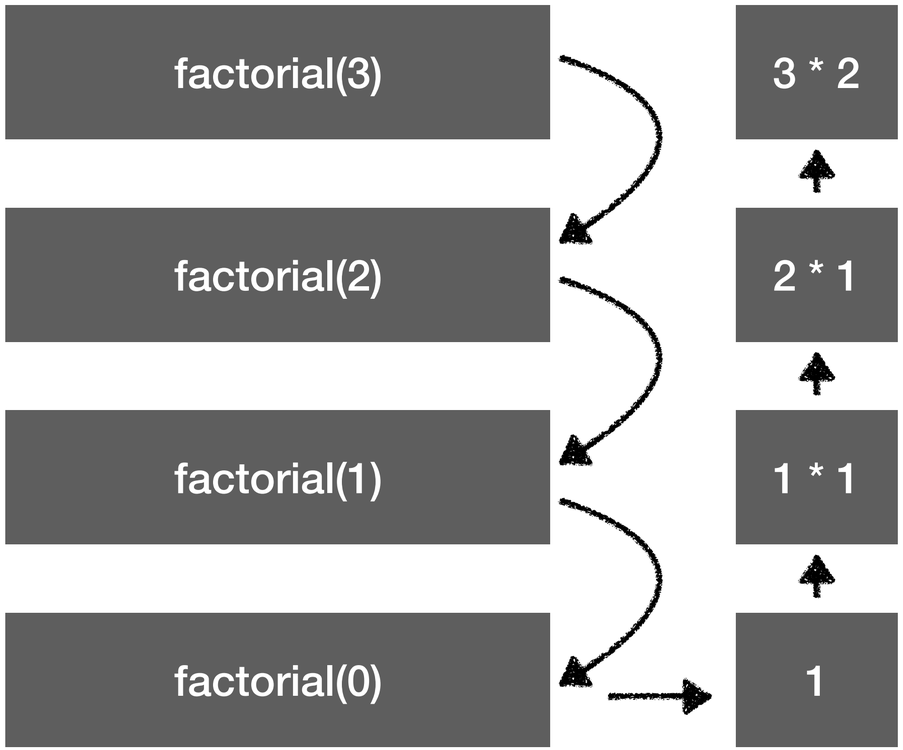
```
fun factorial(n: Int): Int {
    return when (n) {
        1 -> 1
        else -> n * factorial( n: n - 1)
    }
}
```

Recursive trap



https://static.digit.in/default/898e58a0a699ae79913ba1a754239b3c95365d2d.jpeg

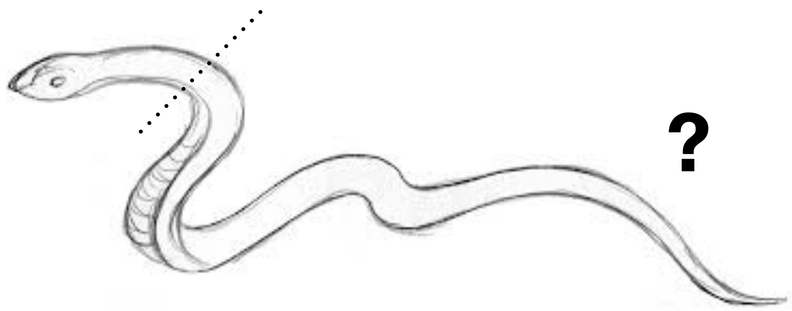
What’s the cost?



https://png.icons8.com/metro/1600/stackoverflow.png

Answer’s in the tail

- Function calls itself at the end of the function
- No computation is done after the return of recursive call



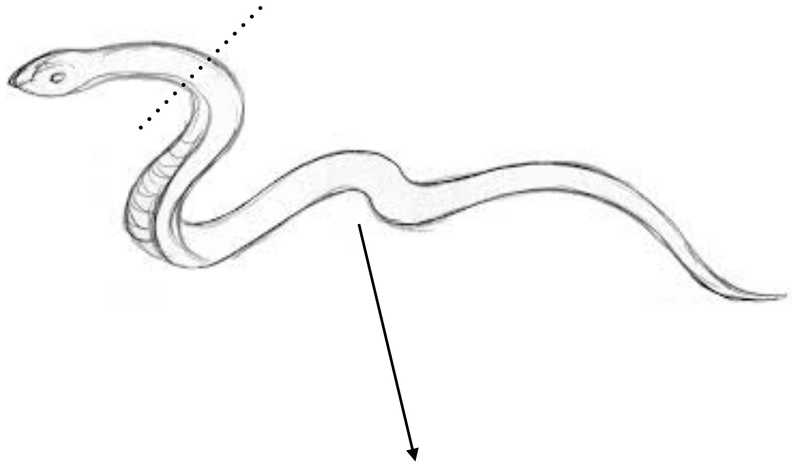

```
fun fact1(n: Int): Int {
    return when (n) {
        1 -> 1
        else -> n * fact1( n: n - 1)
    }
}
```

```
fun fact2(n: Int, acc: Int = 1): Int {
    return when (n) {
        1 -> acc
        else -> fact2( n: n - 1,  acc: acc * n)
    }
}
```

https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcSsCfQgfLVEedLLgPAZXskWzeV9u5tM7BKMw3Vf_FXGw50mR3C7

Tail recursion

```
fun fact1(n: Int): Int {
    return when (n) {
        1 -> 1
        else -> n * fact1( n: n - 1)
    }
}
```



```
tailrec fun fact2(n: Int, acc: Int = 1): Int {
    return when (n) {
        1 -> acc
        else -> fact2( n: n - 1,  acc: acc * n)
    }
}
```

https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcSsCfQgfLVEedLLgPAZXskWzeV9u5tM7BKMw3Vf_FXGw50mR3C7

Good news

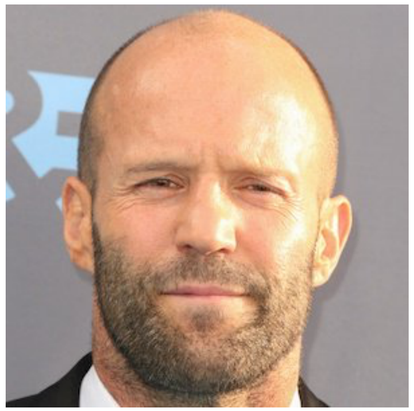
- Tail recursion can be converted into loop
- Tail recursion optimization is common in functional languages

```
public static final int fact2(int n, int acc) {
    while(true) {
        switch(n) {
        case 1:
            return acc;
        default:
            int var10000 = n - 1;
            acc *= n;
            n = var10000;
        }
    }
}
```

https://cfblog-58e9.kxcdn.com/wp-content/uploads/2015/10/content1-1024x626.jpg

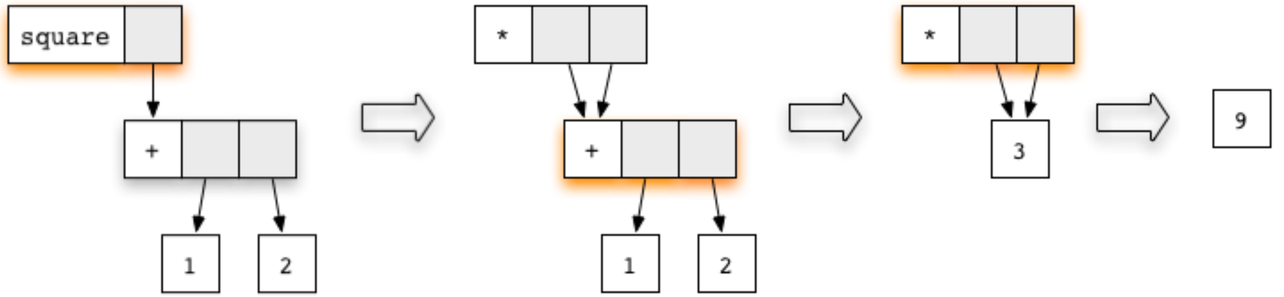
Lazy evaluations

- Ability to define potentially infinite data structures
- Avoid needless calculations
- Graph reduction



“Lazy evaluation never performs more evaluation steps than eager evaluation.”
J. Statham

```
λ> square x = x * x
λ>
λ> square (1 + 2)
λ>
=> 9
```



https://www.famousbirthdays.com/faces/statham-jason-image.jpg

https://hackhands.com/data/blogs/ClosedSource/lazy-evaluation-works-haskell/assets/blocks-square-eval.png

Lazy evaluations

```
fibs = 1 : 1 : zipWith(+) fibs (tail fibs)
```

```
> take 10 fibs
>
=> [1,1,2,3,5,8,13,21,34,55]
```

```
primes = sieve (2 : [3, 5..])
  where
    sieve (p:xs) = p : sieve [x|x <- xs, x `mod` p > 0]
```

```
> take 10 primes
>
=> [2,3,5,7,11,13,17,19,23,29]
```

Lazy trap

https://ul-a.akamaihd.net/opt/ul-a.akamaihd.net/images/products/134363/slide/666x363/Felicity_Sofa_Cum_Bed_Yellow_2.jpg?1482402802

Lazy evaluations

- Time win
- Space fail (sometimes)

```haskell
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f a []     = a
foldl f a (x:xs) = foldl f (f a x) xs
```

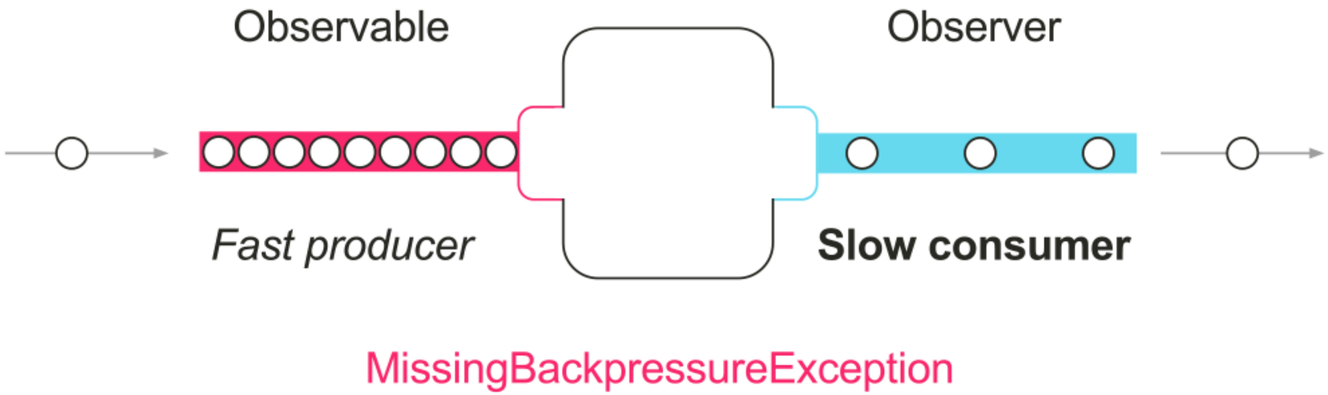
```haskell
foldl (+) 0 [1..100]
=>  foldl (+) 0 (1:[2..100])
=>  foldl (+) (0 + 1) [2..100]
=>  foldl (+) (0 + 1) (2:[3..100])
=>  foldl (+) ((0 + 1) + 2) [3..100]
=>  foldl (+) ((0 + 1) + 2) (3:[4..100])
=>  foldl (+) (((0 + 1) + 2) + 3) [4..100]
=>  …
```

https://cfblog-58e9.kxcdn.com/wp-content/uploads/2015/10/content1-1024x626.jpg

Slow trap

https://images.spot.im/v1/production/c5yynw7sjn0al8pd1nh2

Rx & Backpressure

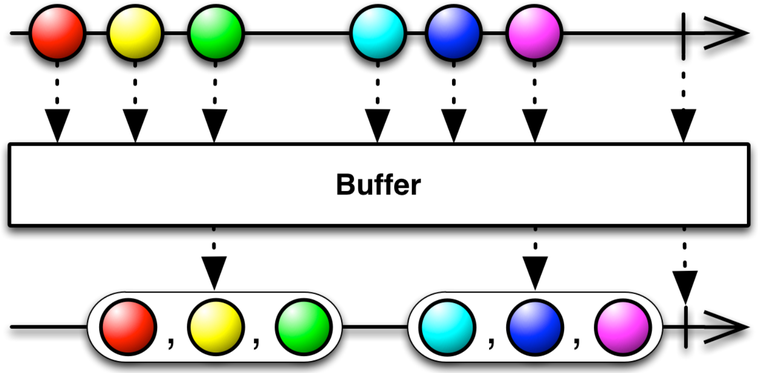


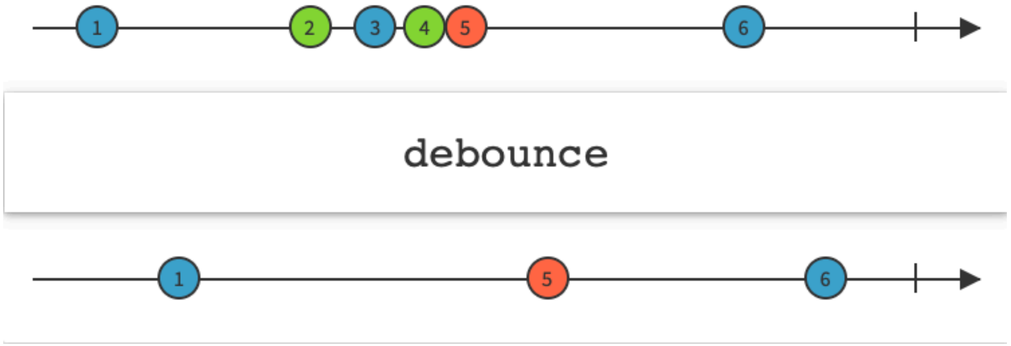
- Observable is emitting items more rapidly than an operator or observer can consume them
- What to do with such a growing backlog of unconsumed items?

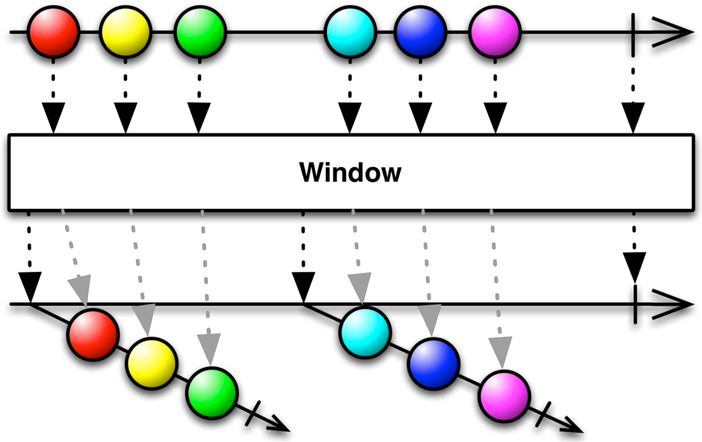
https://cdn-images-1.medium.com/max/1600/1*G-yJQ_ururyvMGkGRA3eAw.png

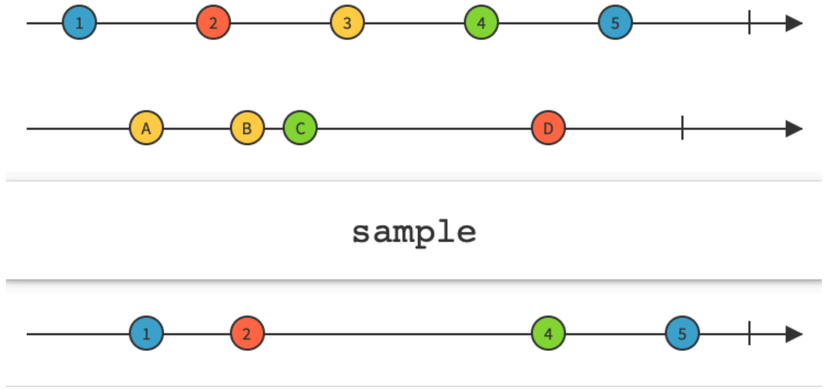
Rx & Backpressure

Flow control operators









http://reactivex.io

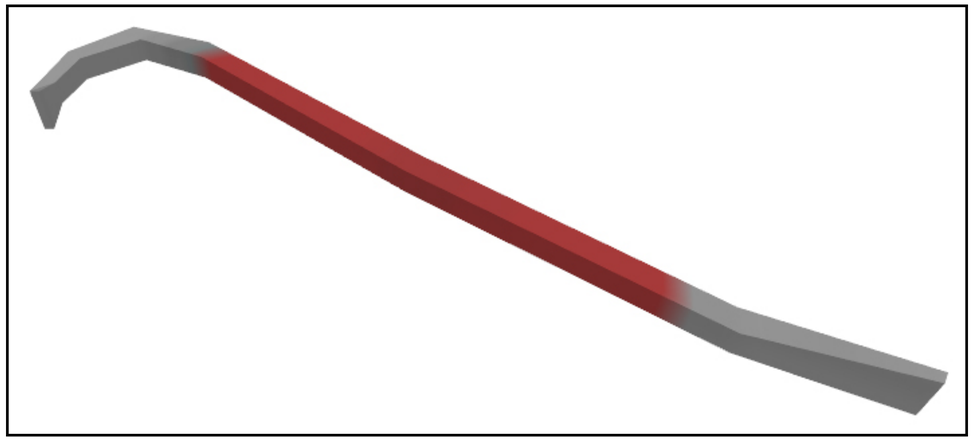
Rx & Backpressure

Resolution strategies (Flowable)

LATEST	**Keeps only the latest onNext value, overwriting any previous value**
BUFFER	**Buffers *all* onNext values until the downstream consumes it**
DROP	**Drops the most recent onNext value if the downstream can't keep up**
ERROR	**Signals a MissingBackpressureException in case the downstream can't keep up**
MISSING	**OnNext events are written without any buffering or dropping**

https://cfblog-58e9.kxcdn.com/wp-content/uploads/2015/10/content1-1024x626.jpg

Lambdas

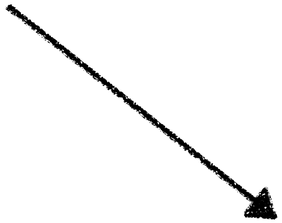


https://vignette.wikia.nocookie.net/half-life/images/9/9b/Crowbar_worldmodel.jpg/revision/latest?cb=20090627131558&path-prefix=en

Lambdas for better life

```
stateOwner.addStateListener(new StateChangeListener() {

    public void onStateChange(State oldState, State newState) {
        // do something with the old and new state.
    }
});
```



```
stateOwner.addStateListener(
    (oldState, newState) -> System.out.println("State changed")
);
```

Naive approach

Retrolambda: Use Lambdas on Java 7

- Lambda expressions are backported by converting them to anonymous inner classes
- Optimization of using a singleton instance for stateless lambda expressions to avoid repeated object allocation

https://cdn-images-1.medium.com/max/1200/1*qCNzc4zhl3yfKFeNErBq1A.png

Kotlin tricks

- Lambdas and anonymous functions are compiled as Function objects
- Each lambda expression compiled as a Function will actually add 3 or 4 methods to the total methods count
- Kotlin has **inline** keyword that tells compiler to inline! lambdas :D

Java 8+

- uses **invokedynamic** instruction (introduced in Java7)
- non-capturing lambdas are compiled to static methods
- capturing lambdas are compiled to instance methods
- **invokedynamic** allows to resolve lambdas at runtime

https://upload.wikimedia.org/wikipedia/en/thumb/3/30/Java_programming_language_logo.svg/1200px-Java_programming_language_logo.svg.png

https://cfblog-58e9.kxcdn.com/wp-content/uploads/2015/10/content1-1024x626.jpg

Conclusion

- Functional Programming is great
- Tail recursion is preferable
- Thunks can eat your memory
- Don’t forget about backpressure
- Use lambdas with care

https://images.unsplash.com/photo-1520895653685-c739b6db8fce?ixlib=rb-0.3.5&ixid=eyJhcHBfaWQiOjEyMDd9&s=2477b2e4516ec05d13863a8cb6d2feab&w=1000&q=80