А нам-то зачем функциональное программирование?

Вагиф Абилов

Miles Norway

Vagif Abilov

Functional programming What's in it for us?

About myself

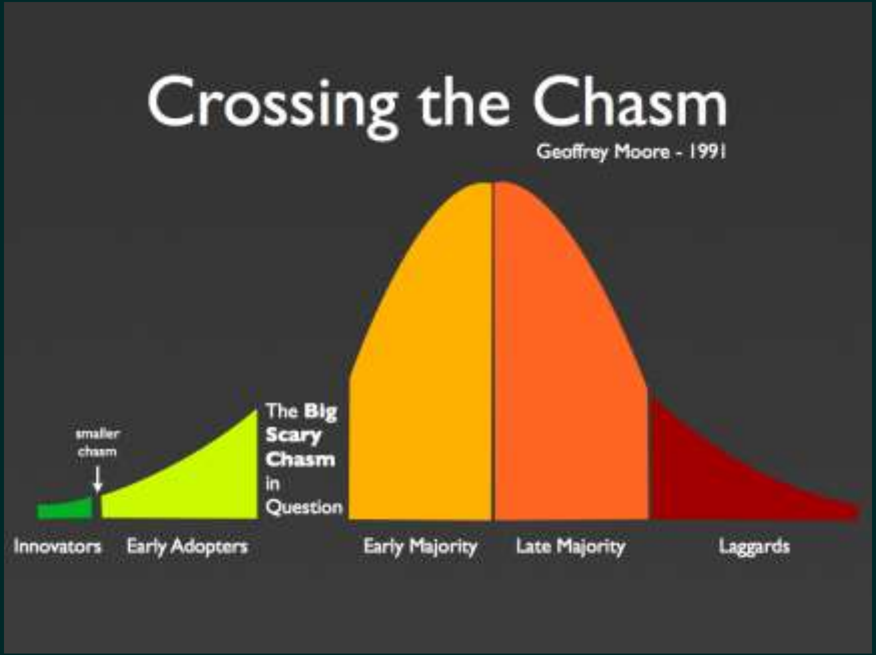
- Consultant in Miles (several “Great Place to Work” awards in Norway in Europe)
- Mail: vagif.abilov@gmail.com
- Twitter: @ooobject
- GitHub: object
- BitBucket: object
- Blog: http://vagifabilov.wordpress.com/
- Articles: http://www.codeproject.com

What this talk is *not* about

- Proving that certain language paradigm better fits agile development practices
- Convincing you that the language of your choice is *sooo* last week
- Starting the Vietnam war
- Going into deep level language details

So what is it about then?

- The talk is aimed at "pragmatists in pain"
- Term coined by Erik Sink in his blog post about F# adoption
- Refers to Geoffrey Moore's technology adoption life cycle



Agenda

1. Making it with just transformations
2. Designing generic code with type inference
3. Painless concurrency
4. Discriminated unions and pattern matching
5. Railway oriented error handling
6. Specifications and tests

The code examples are in F#, Scala has similar syntax

Let's play job interview: designing Conway’s game of life

Conway's game of life

- Invented in 1970 by the British mathematician John Conway
- Zero-player game, its evolution is fully determined by its initial state
- The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, alive or dead

Rules of Conway's game of life

1. Any live cell with fewer than two live neighbours dies, as if caused by under-population
2. Any live cell with two or three live neighbours lives on to the next generation
3. Any live cell with more than three live neighbours dies, as if by overcrowding
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction

Implementing Conway's game of life

How do we start?

- Classes
- Class properties
- Class methods
- Structs
- Enums

What people say on Twitter

"How is writing Java like writing classic Russian literature? You have to introduce 100 names before anything can happen."

@jamesiry

Speaking about Russian literature

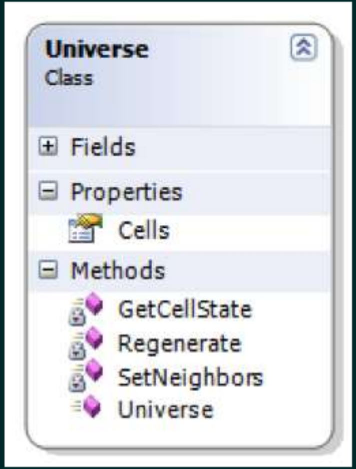
Шепот, робкое дыханье.
Трели соловья,
Серебро и колыханье
Сонного ручья.

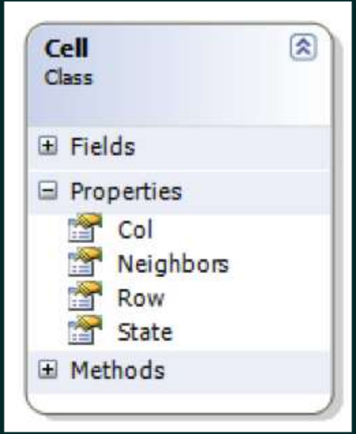
Свет ночной, ночные тени,
Тени без конца,
Ряд волшебных изменений
Милого лица,

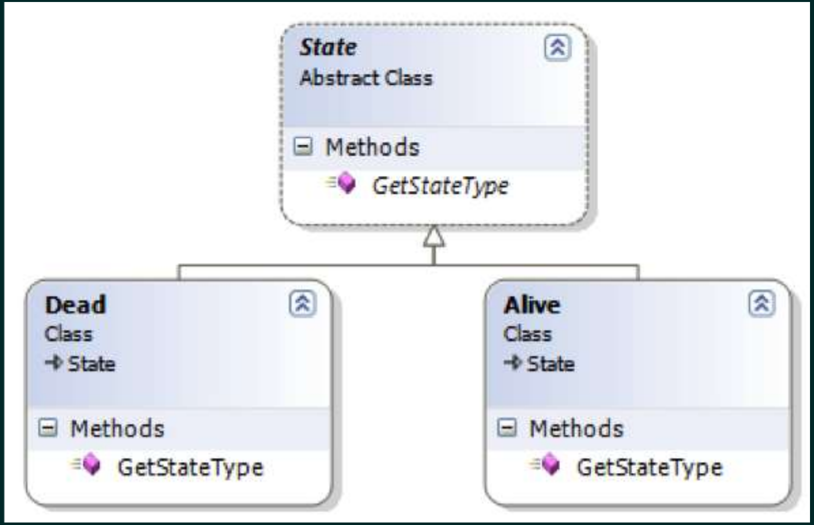
В дымных тучках пурпур розы,
Отблеск янтаря,
И лобзания, и слезы,
И заря, заря!..

A short poem by Afanasy Fet (1850) without a single verb

Checking reference implementation







CodeProject.com

Solving Conway's Game of Life using State Pattern

Implementation code metrics

- 5 classes
- 5 properties
- 5 methods
- 316 lines of code
- 64 effective lines of code (calculated using VS code metrics)

Cell state definition propositions

- Class (base class with 2 subclasses implementing State pattern)
- Enum (Dead, Alive)
- Boolean

Cell state choice consequence

- No matter what type you choose to represent cell state, you will need a cell property to hold it
- Having cells with different property values (Dead and Alive) encourages design where both states of cells are stored
- Storing cells with different states has negative impact on the scalability
- Moreover, it limits the solution to boards of fixed size
- Adding names add constraints!

Design patterns rant

Design patterns are often introduced to patch up shortcomings in the language

Solving Conway's game in a functional way

```
let neighbours (x, y) =
    [ for i in x-1..x+1 do
      for j in y-1..y+1 do
           if not (i = x && j = y) then yield (i,j) ]

let isAlive population cell =
    population
    |> List.exists ((=) cell)

let aliveNeighbours population cell =
    neighbours cell
    |> List.filter (isAlive population)
```

…survival and reproduction criteria

```
let survives population cell =
    aliveNeighbours population cell
    |> List.length
    |> fun x -> x >= 2 && x <= 3

let reproducible population cell =
    aliveNeighbours population cell
    |> List.length = 3
```

… the final part of the solution

```
let allDeadNeighbours population =
    population
    |> List.collect neighbours
    |> Set.ofList |> Set.toList
    |> List.filter (not << isAlive population)

let nextGeneration population =
    List.append
        (population
        |> List.filter (survives population))
        (allDeadNeighbours population
        |> List.filter (reproducible population))
```

Note use of colors: the only word in white is ”population". No variables!

Preliminary observations

- We haven't defined a single class
- We haven’t explicitly used types
- Having defined necessary functions, we used them without defining a single variable
- The solution doesn't only scale well, it works on an infinite board
- The algorithm is generic (will be demonstrated in a minute)
- The algorithm can be easily parallelized (will be demonstrated in a few minutes)

Type inference makes code generic

```fsharp
let isAlive population cell =
    population
    |> List.exists ((=) cell)

let aliveNeighbours population cell =
    neighbours cell
    |> List.filter (isAlive population)

let survives population cell =
    aliveNeighbours population cell
    |> List.length
    |> fun x -> x >= 2 && x <= 3

let reproducible population cell =
    aliveNeighbours population cell
    |> List.length = 3
```

```fsharp
let allDeadNeighbours population =
    population
    |> List.collect neighbours
    |> Set.ofList |> Set.toList
    |> List.filter (not << isAlive population)

let nextGeneration population =
    List.append
        (population
        |> List.filter
            (survives population))
        (allDeadNeighbours population
        |> List.filter
            (reproducible population))
```

Inferring board dimension

```
let neighbours (x, y) =
    [ for i in x-1..x+1 do
      for j in y-1..y+1 do
           if not (i = x && j = y) then yield (i,j) ]

let neighbours (x, y, z) =
    [ for i in x-1..x+1 do
      for j in y-1..y+1 do
      for k in z-1..z+1 do
          if not (i = x && j = y && k = z) then yield (i,j,k) ]
```

Conway's game of colors using type inference

```
type Color = Red | Green | Blue | White | Gray
           | Black | Orange | Yellow | Brown

let neighbours color =
    match color with
    | Red -> [Red; Orange; Brown]
    | Green -> [Green; Blue; Yellow]
    | Blue -> [Blue; Green]
    | White -> [White; Gray]
    | Black -> [Black; Gray]
    | Gray -> [Gray; Black; White]
    | Orange -> [Orange; Red; Yellow]
    | Yellow -> [Yellow; Orange; Green]
    | Brown -> [Brown; Red]
```

… and the main algorithm hasn't changed a bit

```fsharp
let isAlive population cell =
    population
    |> List.exists ((=) cell)

let aliveNeighbours population cell =
    neighbours cell
    |> List.filter (isAlive population)

let survives population cell =
    aliveNeighbours population cell
    |> List.length
    |> fun x -> x >= 2 && x <= 3

let reproducible population cell =
    aliveNeighbours population cell
    |> List.length = 3
```

```fsharp
let allDeadNeighbours population =
    population
    |> List.collect neighbours
    |> Set.ofList |> Set.toList
    |> List.filter (not << isAlive population)

let nextGeneration population =
    List.append
        (population
        |> List.filter
            (survives population))
        (allDeadNeighbours population
        |> List.filter
            (reproducible population))
```

Language immutability as a remedy for concurrency hell

"There’s no such thing as a convention of immutability, as anyone who has tried to enforce one can attest. If a data structure offers only an immutable API, that is what’s most important. If it offers a mixed API, it’s simply not immutable."

Rich Hickey, creator of Clojure and Datomic

Parallelizing Conway's game solution

```
// Sequential solution
let nextGeneration population =
    List.append
        (population
        |> List.filter (survives population))
        (allDeadNeighbours population
        |> List.filter (reproducible population))

// Parallel solution
let nextGeneration population =
    seq {
         yield (population
                |> PList.filter (survives population))
         yield (allDeadNeighbours population
                |> PList.filter (reproducible population))
    }
    |> PSeq.toList
```

Are you with me so far? If so you are awarded

Functional languages as DSLs

- Functional languages often don't have the same level of ceremony as traditional object-oriented languages
- Terse syntax may sound like a threat to readability but in fact domain specific definitions in languages like Scala and F# are clear and readable even for non-programmers

Recommended reading

- Scott Wlaschin

Domain Driven Design with the F# type System

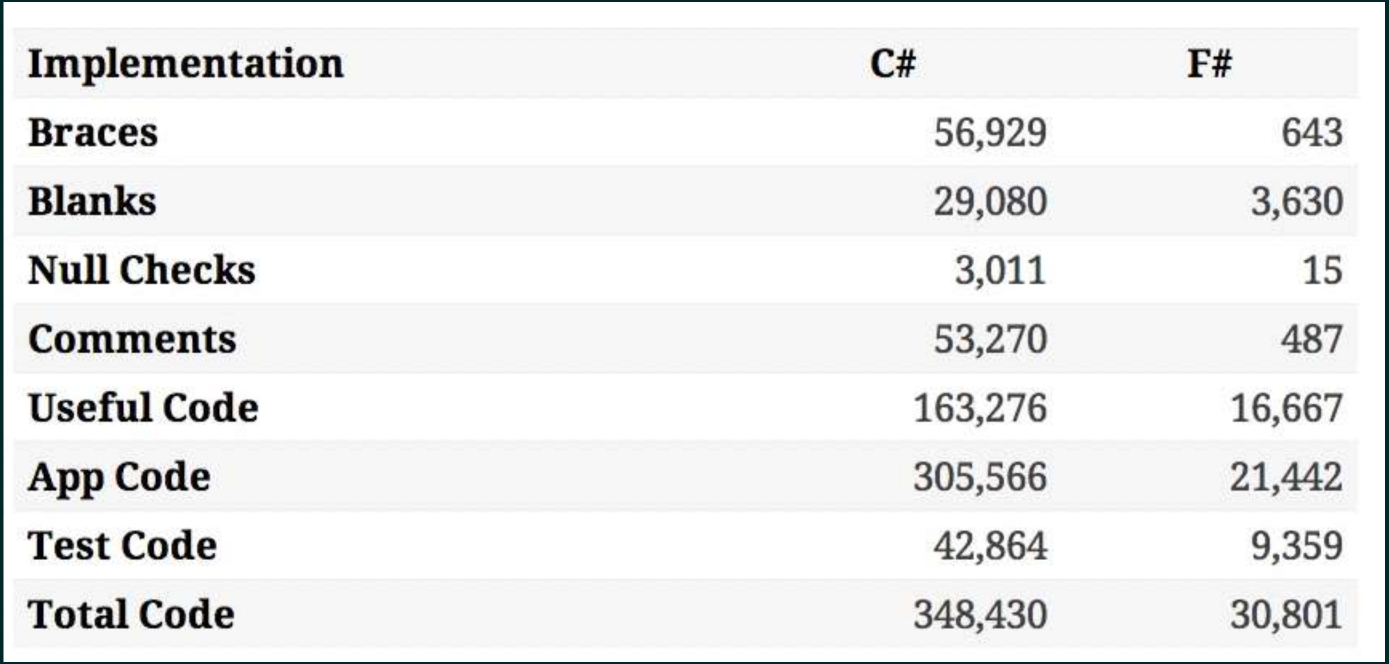
http://bit.ly/1MIokLd

- Simon Cousins

Time for Functions

http://bit.ly/1GTFpRw

Energy trading project statistics (by Simon Cousins)



Implementation	C#	F#
Braces	56,929	643
Blanks	29,080	3,630
Null Checks	3,011	15
Comments	53,270	487
Useful Code	163,276	16,667
App Code	305,566	21,442
Test Code	42,864	9,359
Total Code	348,430	30,801

Defining financial domain

```
type CardType = VISA | MasterCard | AmEx | Diners | Domestic

type CardNumber = string

type PaymentMethod =
    | Cash
    | Cheque of int
    | Card of CardType * CardNumber
```

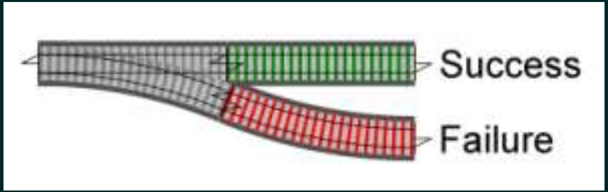
Working with financial domain

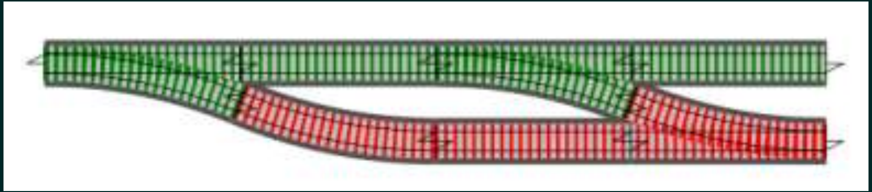
```
let calculateTips sum paymentMethod =
    match paymentMethod with
    | Cash -> sum * 0.05
    | Cheque _ -> sum * 0.20
    | Card (cardType, _) when cardType = Domestic -> sum * 0.10
    | Card (_, _) -> sum * 0.15
```

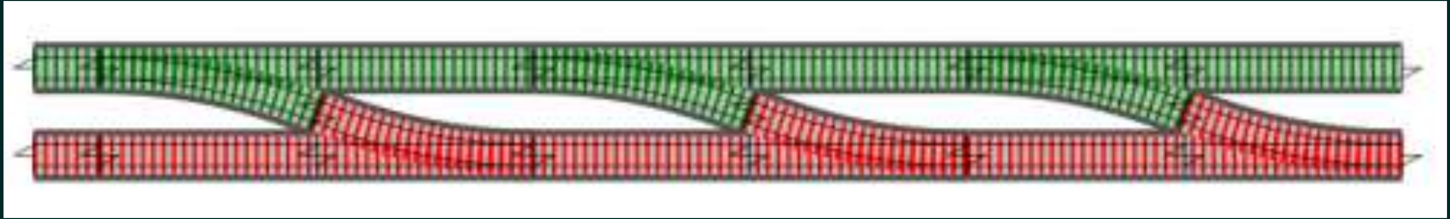
Error management monadic way

Term "Railway Oriented Programming" coined by Scott Wlaschin

Happy and error paths combined







Source: http://bit.ly/1CqGyAN

Easier code execution workflow management

```
Product
|> placeInShoppingCart
|> proceedToCheckout
|> selectShipmentMethod
|> selectPaymentMethod
|> authorizePayment
```

Testing functional way

```fsharp
let ``Block should not change``() =
    let population = [(1,1); (1,2); (2,1); (2,2)]
    population
    |> nextGeneration
    |> should equal population
```

BDD using functional languages

Scenario: Refunded items should be returned to stock

Given a customer buys a black jumper

And I have 3 black jumpers left in stock

When he returns the jumper for a refund

Then I should have 4 black jumpers in stock

Acceptance test functional way

```
let [<Given>] ``a customer buys a black jumper`` () = ()

let [<Given>] ``I have (.*) black jumpers left in stock`` (n : int) =
    stockItem <- { stockItem with Count = n }

let [<When>] ``he returns the jumper for a refund`` () =
    stockItem <- { stockItem with Count = stockItem.Count + 1 }

let [<Then>] ``I should have (.*) black jumpers in stock`` (n : int) =
    stockItem.Count |> should equal n
```

Summary

- Functional transformations bring you far without a single defined type
- Type inference makes your algorithms generic
- Immutable code can be easily parallelized
- Discriminated unions help you define human readable DSLs
- Pattern matching makes your processing rules easy to read too
- Error handling becomes part of functional transformations
- Start your adventure with functions by writing specifications and tests in a functional language

And now for something completely different

Living With No Sense Of Monads

Originally performed by Smokie
with slightly different words

I first met Alice in a small bookstore,
"What book, - I asked, - are you looking for?"
And she said: "About monads."

I understood that's a key to her heart,
I had no doubts that I was smart.
It should be easy reading, an easy start…

But I still don't get this wisdom,
It's really mystic word
I guess it has it's reasons,
Someone gets it, but I don't.
'Cos for twenty-four years
I've been trying to understand the monads.

Twenty-four years

Just waiting for a chance,

I have to keep on reading,

Maybe get a second glance,

I will never get used to not understanding sense of monads

Grew up together, went to the same class,
And to be honest: I was better in math
Than Alice.

Too old for rock-n-roll - I'm forty four,
Too young for monads - still have a hope.
That the day will come
When I let Alice know…

But I still don't get this wisdom,
It's really mystic word
I guess it has it's reasons,
Someone gets it, but I don't.
'Cos for twenty-four years
I've been trying to understand the monads.

(all together):
Monads! What the f*ck is monads?!

Twenty-four years

Just waiting for a chance,

I have to keep on reading,

Maybe get a second glance,

I will never get used to not understanding sense of monads

Thank you!

- I am Vagif Abilov, consultant in Miles
- Mail: vagif.abilov@gmail.com
- Twitter: @ooobject
- GitHub: object
- BitBucket: object
- Blog: http://vagifabilov.wordpress.com/
- Articles: http://www.codeproject.com