Concurrency paradigm and Functional Programming

BY RICCARDO TERRELL- @TRIKACE

“OOP makes code understandable by encapsulating moving parts. FP makes code understandable by eliminating moving parts”

— *Michael Feathers*
(author of Working with Legacy Code)

“It’s really clear that the imperative style of programming has run its course. … We’re sort of done with that. … However, in the declarative realm we can speculate a 10x improvement in productivity in certain domains”

— Anders Hejlsberg (C# Architect)

Agenda

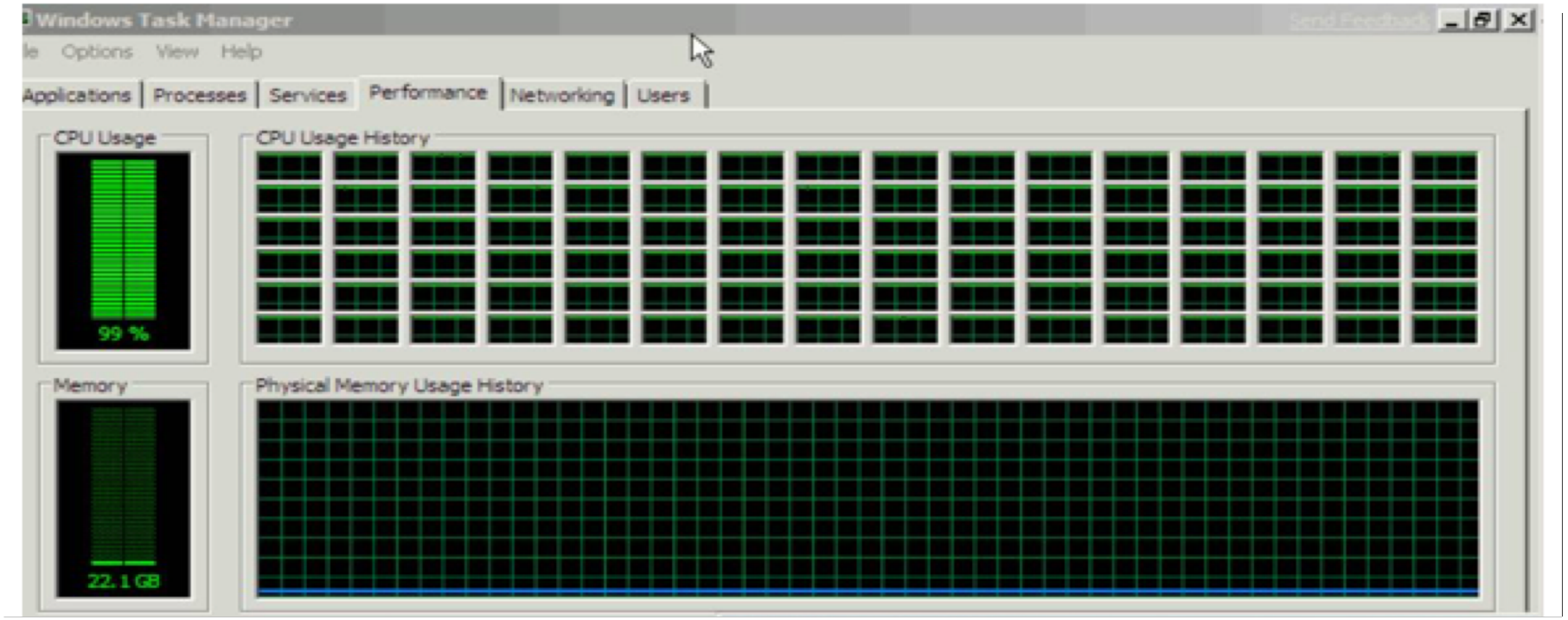
- History and influence of Functional Programming
- What is Functional Programming and motivation
- Functional Programming foundations

Objectives

- Use functions as building block and **composition** to solve complex problems with simple code
- **Immutability** and **Isolation** are your best friends to write concurrent applications
- **Functional** and **imperative** styles should NOT stand in opposition … they **should COEXIST**
  - Poly-paradigm programming is more powerful and effective than polyglot programming

Moore’s law - The Concurrency challenge

Modern computers come with multiple processors, each equipped with multiple cores, but the single processor is slower than it used to be!




MULTITHREADING
THEORY


PRACTICE

Concurrent Programming is HARD

Easy to write racy code

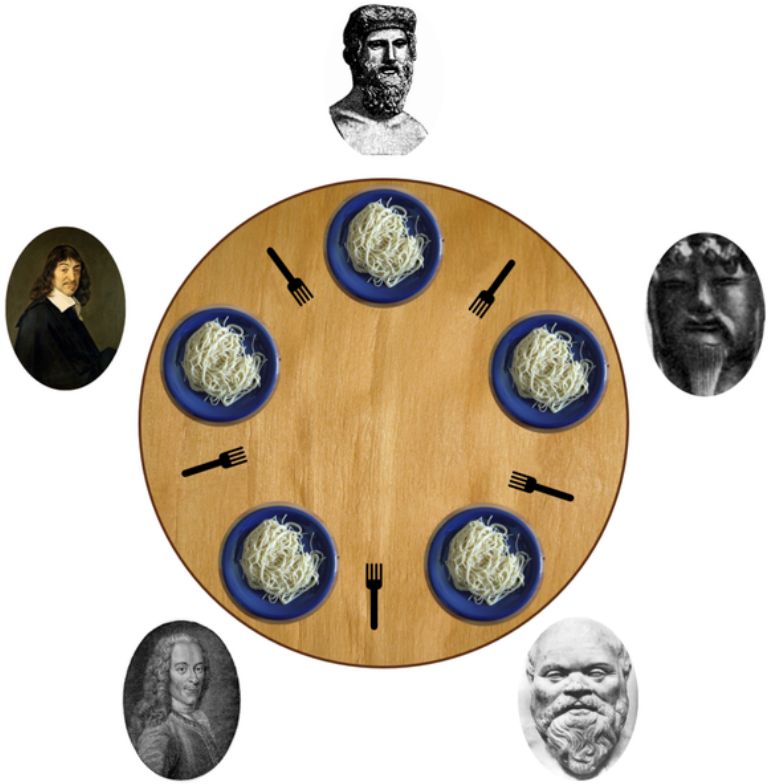
Deadlocks ☹

Issues

The issue is Shared Memory

- Shared Memory Concurrency
- Data Race / Race Condition
- Works in sequential single threaded environment
- Very difficult to parallelize
- Difficult to maintain and test
- Locking, blocking, call-back hell

Fully Synchronized Objects



Concurrency Primitives

```
public class BankAccount {
    Object _lock = new Object();
    int balance;

    public void adjustBalance(int amount) {
        lock(_lock) {
            balance = balance + amount;
        }
    }
```

Concurrent Code Some Important bugs

- Deadlock and starvation
- Livelock
- Race conditions

Deadlock and Starvation

Deadlock – A situation where no further progress can be made by the program (when all processes are blocked).

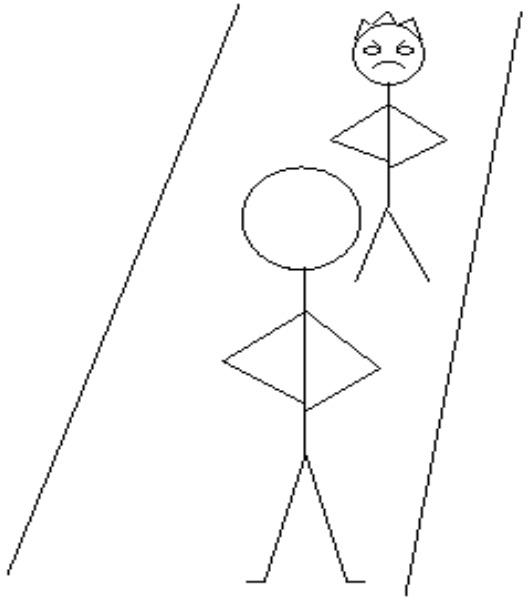
Let *S* and *Q* be two locks

P_0	P_1
S.Lock();	*Q.Lock()*;
Q.*Lock*();	*S.Lock()*;
⋮	⋮
*S.Un*lock();	*Q.Unlock()*;
Q.Unlock();	*S.Unlock()*;

Starvation - A process is ready to run or to use a resource, but is never given a chance by the scheduler.

Livelock

Threads are doing something… yet they are not doing anything useful!!



Livelock when two polite people are in the hallway

Where is the Livelock?

```
// thread 1
getLocks12(lock1, lock2)
{
  lock1.lock();
  while (lock2.locked())
  {
     lock1.unlock();
     wait();
     lock1.lock();
   }
   lock2.lock();
}
```

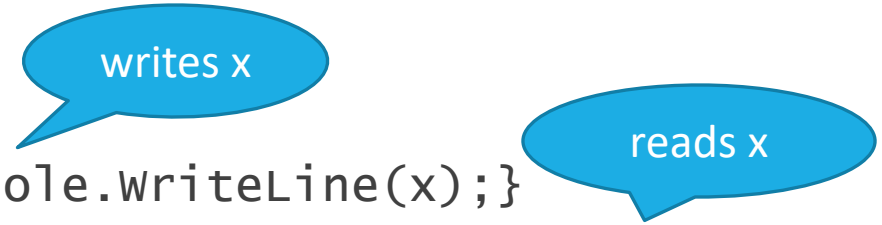
```
// thread 2
getLocks21(lock1, lock2)
{
  lock2.lock();
  while (lock1.locked())
  {
     lock2.unlock();
     wait();
     lock2.lock();
   }
   lock1.lock();
}
```

Data Race

Two concurrent accesses to a memory location where at least one of them is a write.

Example:

```
int x = 1;
Parallel.Invoke(
                () => { x = 2; },
                () => { System.Console.WriteLine(x);}
               );
```



Possible outcomes?

- 1
- 2

Immutability

Purely functional languages do not use mutable data or mutable states, and thus are inherently thread safe – the output value of a function depends entirely on its arguments.

This frees up the programmer from having to manage complex thread synchronization / locks

Immutability

Like string in C# **everything** is immutable

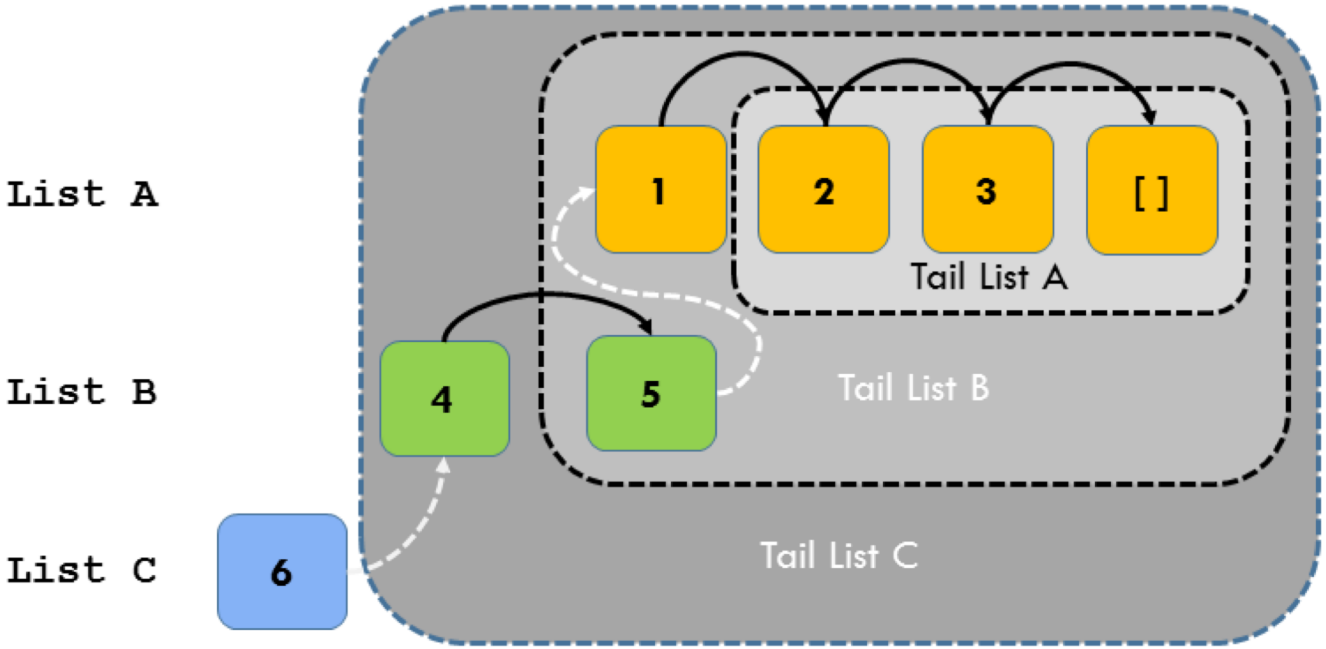
Help in concurrent/parallel code

Immutability is **default** in F#

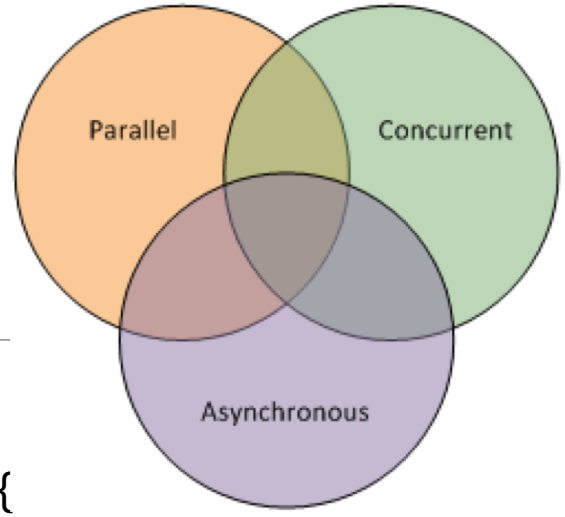
You can choose to go mutable (**use carefully!**)

Copy-on-Write

Structural Sharing

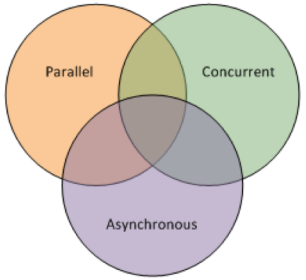


Immutability




```
public class ImmutablePerson
{
    public ImmutablePerson(string firstName, string lastName, int age) {
        FirstName = firstName;
        LastName = lastName;
        Age = age;
    }
    public string LastName { get; }
    public string FirstName { get; }
    public int Age { get; }

    public ImmutablePerson UpdateAge(int age)
    {
            return new ImmutablePerson(this.FirstName, this.LastName, age);
    }
}
```



Immutability

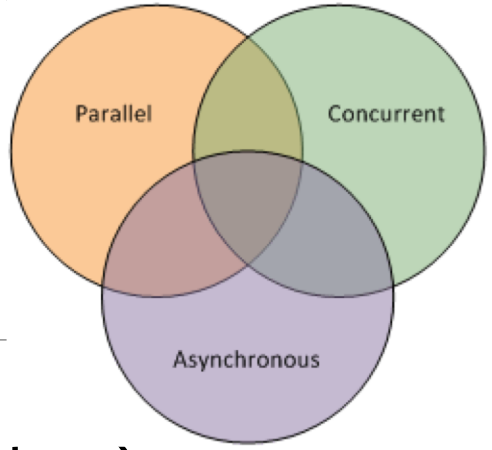
```
public class ImmutablePerson
{
    public ImmutablePerson(string firstName, string lastName, int age) {
        FirstName = firstName;
        LastName = lastName;
        Age = age;
    }
    public string LastName { get; }
    public string FirstName { get; }
    public int Age { get; }
    public ImmutablePerson UpdateAge(int age)
    {
            return new ImmutablePerson(this.FirstName, this.LastName, age);
    }
}
```

```
type Person = {FirstName:string; LastName:string; Age:int}
```

Passing immutable objects as messages between threads.

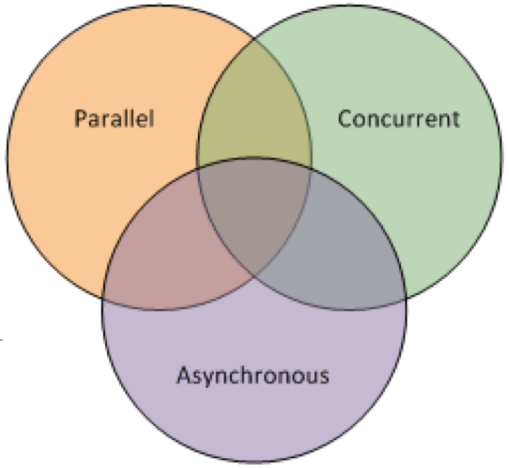
Concurrent modification ceases to be a problem!

Compare And Swap - CAS




```
void DoWithCAS<T>(ref T location, Func<T, T> generator)
                                              where T : class
      {
         T temp, replace;
         do
         {
             temp = location;
             replace = generator(temp);
         } while (Interlocked.CompareExchange(ref location,
                                        replace, temp) != temp);
      }
```

Compare And Swap - CAS




```
public class LockFreeStack<T> {
    private class Node {
        public T Data;
        public Node Next;
    }
    private Node head;

    public void Push(T element) {
        Node node = new Node {Data = element};
        DoWithCAS(ref head, h => {
            node.Next = h;
            return node;
        });
    }
```

The Solution is Immutability and Isolation

IMMUTABILITY +
ISOLATION =

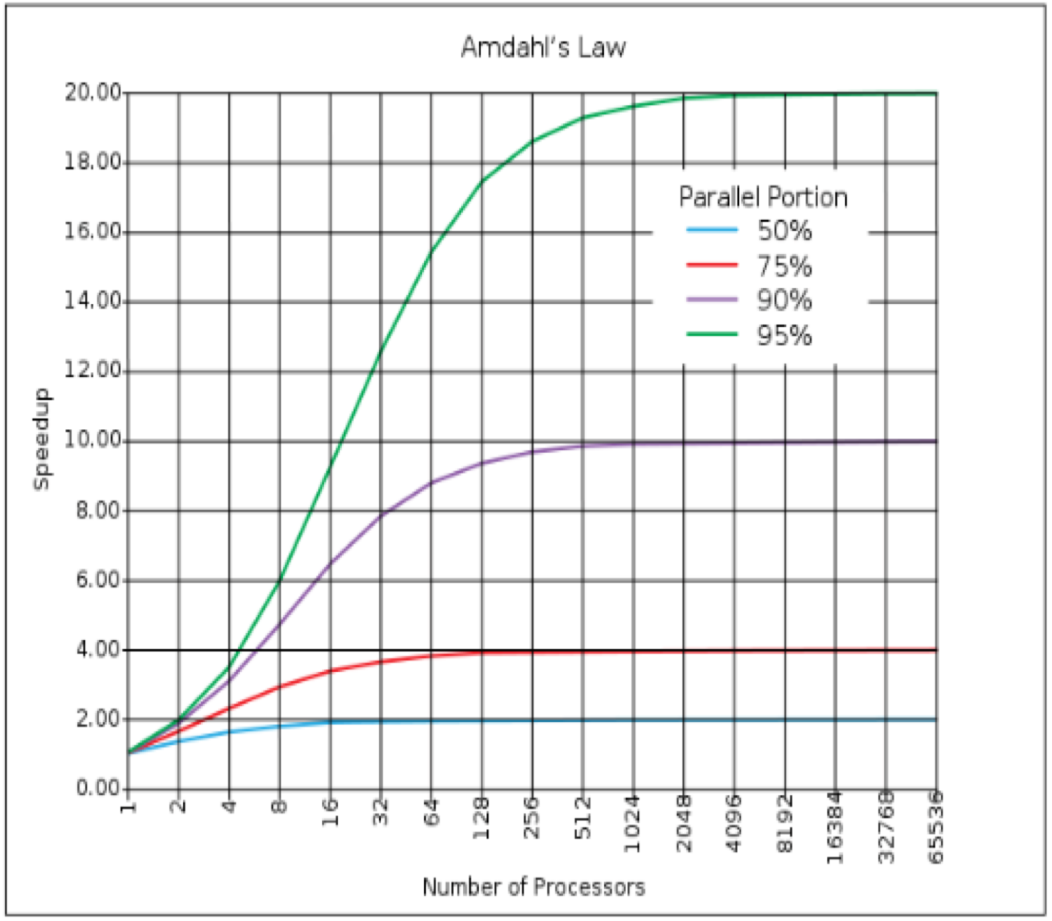
BEST CONCURRENT PROGRAMMING MODEL

Message Passing Concurrency (Actors)

The new reality : Amdahl's law

The speed-up of a program using multiple processors in parallel computing is limited by the sequential fraction of the program.

For example if 95% of the program can be parallelized, the theoretical maximum speed-up using parallel computing would be 20 times, no matter how many processors are used



Parallel and Concurrent Patterns

Why patterns?

- Patterns help “structure” your code
- Sharing code becomes easier – “I’m using a singleton here...”
- Patterns have been “proven” and are understood so if you follow one you benefit from common knowledge
- Languages often “support” a particular pattern – Ex: decorator, map, reducer
- Optimization can be done without the programmers knowledge
  - Ex: parallelize a map call, copy of modify factory

Parallel Patterns

The following additional parallel patterns can be used for “structured parallel programming”

- Speculative selection
- Map
- Recurrence
- Scan
- Reduce
- Fork/join
- Map-Reduce
- Partition
- Pipeline
- Segmentation
- Stencil
- Search/match
- Gather
- Mergescatter
- Permutationscatter
- Atomic

Functional Programming introduction

Agenda

- History and influence of Functional Programming
- What is Functional Programming
- Why Functional Programming

History

λ

1930

1958 LISP

1962 APL

1966 ISWIM

1973 ML

1977 FP

1986 Erlang

1988 Haskell

1996 Ocaml

2003 Scala

2005 F#

2007 Clojure

Twitter Paper

Your Server as a Function

Marius Eriksen

Twitter Inc.

marius@twitter.com

Abstract

Building server software in a large-scale setting, where systems exhibit a high degree of concurrency and environmental variability, is a challenging task to even the most experienced programmer. Efficiency, safety, and robustness are paramount—goals which have traditionally conflicted with modularity, reusability, and flexibility.

We describe three abstractions which combine to present a pow-

Services Systems boundaries are represented by asynchronous functions called *services*. They provide a symmetric and uniform API: the same abstraction represents both clients and servers.

Filters Application-agnostic concerns (e.g. timeouts, retries, authentication) are encapsulated by *filters* which compose to build services from multiple independent modules.

http://monkey.org/~marius/funsrv.pdf

Twitter Paper

Your Server as a Function

Servers in a large-scale setting are required to process tens of thousands, if not hundreds of thousands of requests concurrently; they need to handle partial failures, adapt to changes in network conditions, and be tolerant of operator errors. As if that weren't enough, harnessing off-the-shelf software requires interfacing with a heterogeneous set of components, each designed for a different purpose. These goals are often at odds with creating modular and reusable software [6].

We present three abstractions around which we structure our server software at Twitter. They adhere to the style of functional programming—emphasizing immutability, the composition of first-class functions, and the isolation of side effects—and combine to present a large gain in flexibility, simplicity, ease of reasoning, and robustness.

What is "Functional Programming?"

Functional programming is just a style, is a programming paradigm that treats computation as the evaluation of functions and avoids state and mutable data…

"Functions as primary building blocks" (first-class functions)

programming with *"immutable"* variables and assignments, no *state*

- Programs work by returning values instead of modifying data

What is "Functional Programming?"

Functional programming is so called because a program consists entirely of functions.

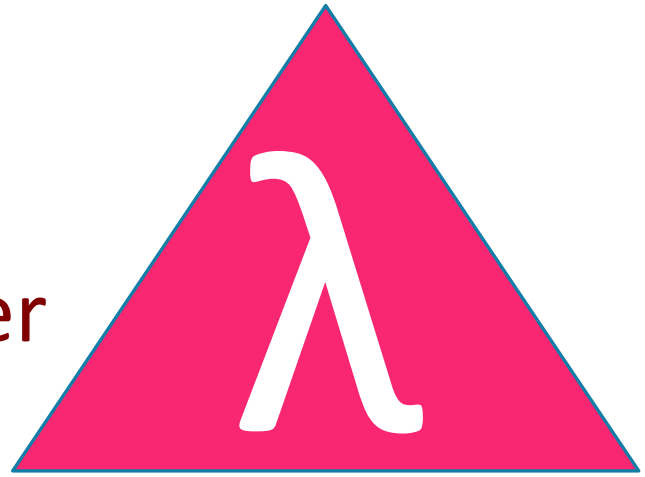
— John Hughes, Why Functional Programming Matters

Declarative

First class

Higher-Order

Functions



Immutability

… is all about functions! Identifying an abstraction and building a function, use existing functions to build more complex abstractions, pass existing functions to other functions to build even more complex abstractions

… but why? … *why FP?*

Motivation for Function Programming

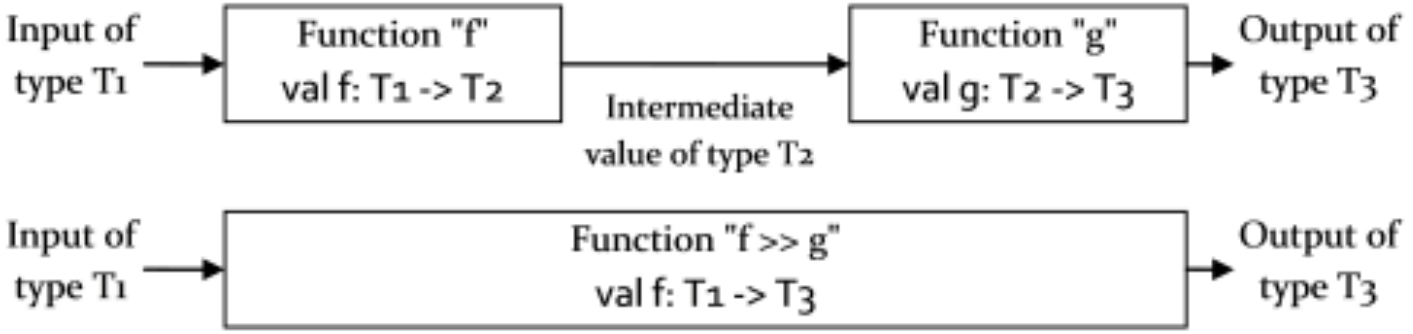
- Functional Programming promotes Composition and Modularity
- Declarative programming style
- Concurrency, FP is easy to parallelize
- Conciseness, less code (no null checking)
- Simplicity in life is good: cheaper, easier, faster, better
  - We typically achieve simplicity in software in two ways:
    - By raising the level of abstraction
    - Increasing modularity
- Better composition and modularity == reuse
- The Multi-core (r)evolution! Concurrency without pain

Functional Composition

Function Composition

Function Composition - Building with composition. Composition is the 'glue' that allows us build larger systems from smaller ones. This is the very heart of the functional style. Almost every line of code is a composable expression. Composition is used to build basic functions, and then functions that use those functions, and so on.

Composition — in the form of passed parameters plus first-class functions



Function Composition Operators

The forward pipe operator:

```
let (|>) arg func =          Type: 'a -> ('a -> 'b) -> 'b
     func arg
```

Allows you to reorder function calls to put the last argument first:

```
let three = 1 |> add 2
```

Produces function
int -> int

Invokes (int -> int)
with 1

Function Composition

```
open System

let evenNumbersAsWords =
    [ 1 .. 15 ]
    |> Seq.filter (fun i -> i % 2 = 0)
    |> Seq.map (fun i -> i.ToWords())
    |> Seq.reduce (sprintf "%s, %s")
```

Creates an F# List with the integers 1 to 15

Partial application of the Seq.filter function

F# Lambda

Partial application of sprintf

Map Reduce!

Immutability!

evenNumbersAsWords:
"two, four, six, eight, ten, twelve, fourteen"

Avoid Side Effects

Referential transparency

An expression is said to be referentially transparent if it can be **replaced** with its corresponding value **without changing the program's behavior**.

As a result, evaluating a referentially transparent function gives the same value for same arguments.

In general it means that the function **don’t have side effect** which may influence its result.

Referential transparency

Which of the following methods are an example of Referential Transparency?

C#

```
DateTime Add(DateTime date, TimeSpen duration) =>
                        date.Add(duration);

DateTime AddDay(DateTime date) =>
                        Add(date, TimeSpan.FromDay(1));

DateTime Tomorrow() => AddDay (DateTime.Now)

DateTime AddTrigger(DateTime date, Func<int> f) =>
                        Add(date, TimeSpan.FromDay(f()));
```

FP Preachings!

Avoid Side-Effects!

- Do not modify variables passed to them
- Do not modify any global variable

Avoid Mutation

Reasoning about your code

```
int Sum(List<int> values) { … }

List<int> values = new List<int>{ 1,2,3,4,5 };

int result1 = Sum(values); // 15

int result2 = Sum(values); // ??
```

Reasoning about your code

```
int Sum(List<int> values) {

      int sum = 0;
      for(int i = 0;i < arr.Length; i++){
            sum += values[i];
            values[i] = 0;

      }
}
```

Reasoning about your code

```
int Sum(List<int> values) { … }

List<int> values = new List<int>{ 1,2,3,4,5 };

int result1 = Sum(values); // 15

int result2 = Sum(values); // ??

int Sum(IEnumerable<int> values) { … }
```

Side Effects

Side effects aren't bad…

Unwanted Side Effects are Evil!

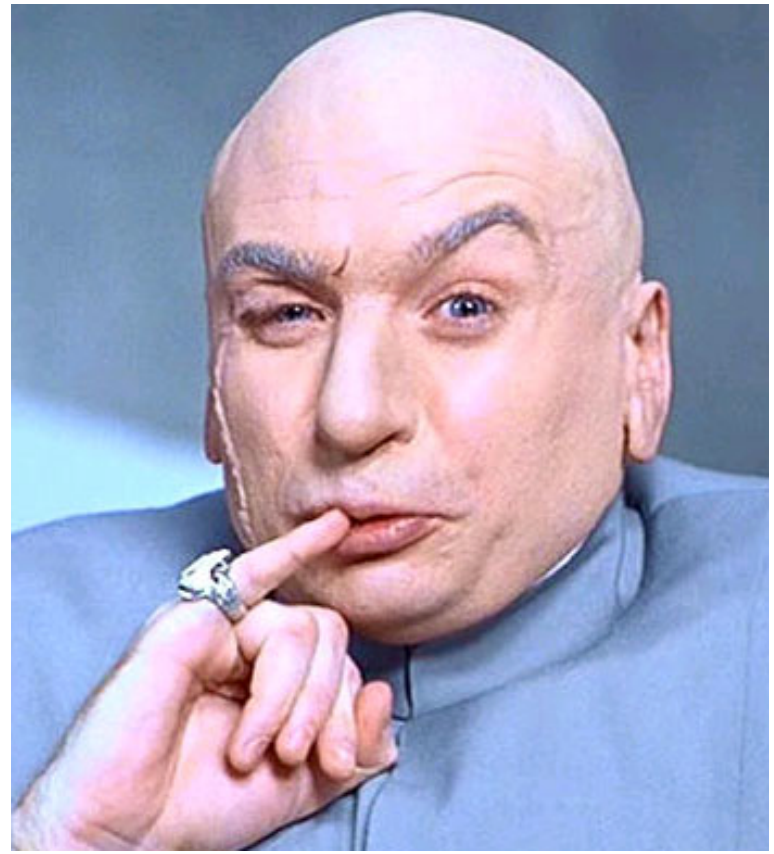
Side effects aren't bad…

unwanted side effects are EVIL

and the root of many bugs

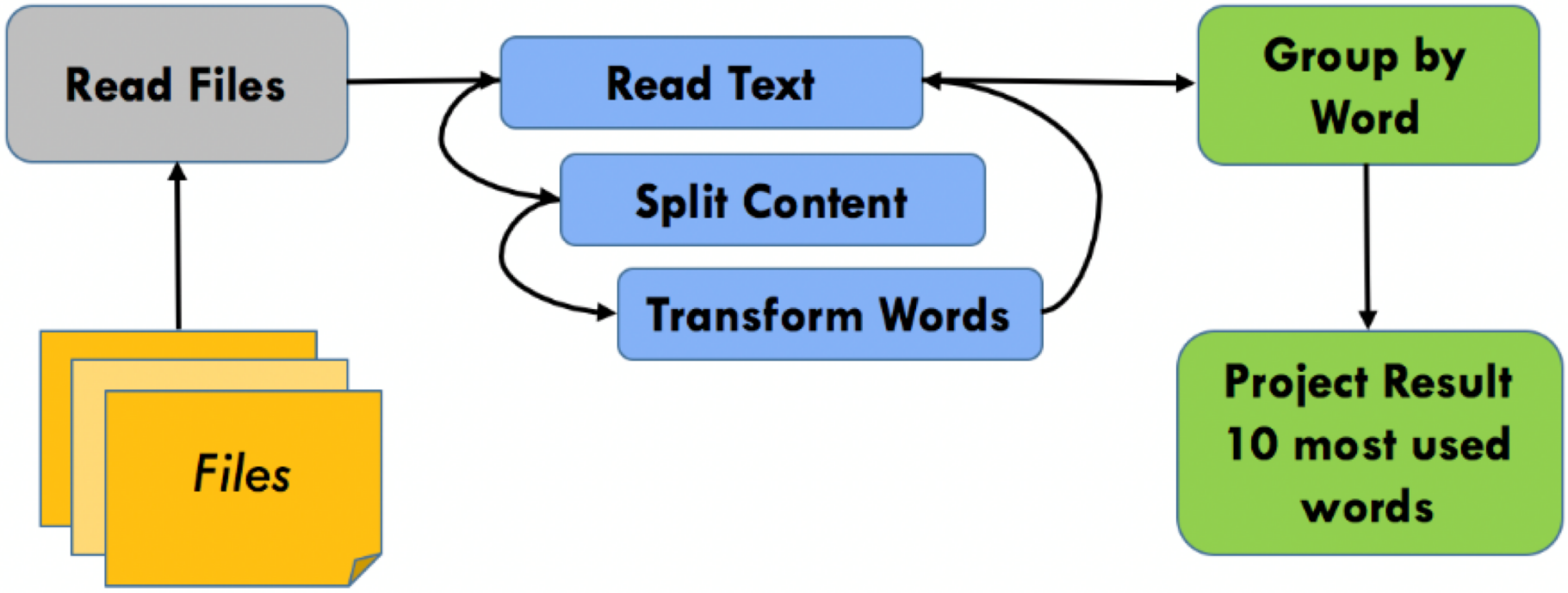
One of the biggest advantages with functional programming is that the order of execution of "side effect free functions" is not important.

… "side effect free functions" are easy to parallelize



Avoiding side effects with pure functions

Parallel words counter with side effects



Parallel words counter with side effects

```
public static Dictionary<string, int> WordsCounter(string source)
{
    var wordsCount =
            (from filePath in
                Directory.GetFiles(source, "*.txt").AsParallel()
                from line in File.ReadLines(filePath)
                from word in line.Split(' ')
                select word.ToUpper())
        .GroupBy(w => w)
        .OrderByDescending(v => v.Count()).Take(10); //#C
    return wordsCount.ToDictionary(k => k.Key, v => v.Count());
}
```

Why removing without side effects

- Easy to reason about the correctness of your program
- Easy to compose functions for creating new behavior
- Easy to be isolated and, therefore, easy to test and less bug prone
- Easy to be executed in parallel; because pure functions don’t have external dependencies, their order of execution (evaluation) doesn’t matter

LAB :

Avoiding side effects with pure function

Parallel words Counter with side effects

```
static Dictionary<string, int> PureWordsPartitioner
                               (IEnumerable<string> content) =>
    (from lines in content.AsParallel()
        from line in lines
        from word in line.Split(' ')
        select word.ToUpper())
               .GroupBy(w => w)
              .OrderByDescending(v => v.Count()).Take(10)
              .ToDictionary(k => k.Key, v => v.Count());

static Dictionary<string, int> WordsPartitioner(string source) {
    var contentFiles =
       from filePath in Directory.GetFiles(source, "*.txt")
            select File.ReadLines(filePath);

    return PureWordsPartitioner(contentFiles);
}
```

Functional Programming introduction

Function definition

A function always gives the **same output** value for a **given input** value

A function has no **side effects**

C#

```
public static int add(int a, int b)
{
    return a + b;
}
```

F#

```
let add a b = a + b
```

Function definition

C#

```
public static T1 First<T1, T2>((T1, T2) t)
{
    return t.Item1;
}
```

F#

```
let first (x, y) = x
```

Function as first class

Functions like **any values**

```
Func<int, int, int> add = (a, b) => a + b;
```

C#

- Function can be **passed as parameter** to other function

```
IEnumerable<T> Where<T> (IEnumerable<T>, Func<T, bool>)
```

C#

- Function can be **return** from other function

```
Func<int, int, int> GetCalcStrategy()
```

C#

Function as first class

C#

```
public static IEnumerable<T2> Select<T1, T2>(
    this IEnumerable<T1> xs,
    Func<T1, T2> f)
{
    foreach(var x in xs)
    {
        yield return f(x);
    }
}

var newList = list.Select(x => x + 1);
```

Function as first class

F#

```
let map (f : 'a -> 'b) (xs : 'a seq) =
    seq { for x in xs do
              yield f x }

let newList = [1..10]
              |> map (fun x -> x + 1)
```

Function as first class

C#

```
public static IEnumerable<T2> Select<T1, T2>(
    this IEnumerable<T1> xs,
    Func<T1, T2> f)
```

F#

```
let map (f : 'a -> 'b) (xs : 'a seq)
```

Higher-Order functions

A higher-order function is a function that takes another function as a parameter, or a function that returns another function as a value, or a function which does both.

```
Func<Developer, GoodSeniorDeveloper> convertToGSD = d => new
  GoodSeniorDeveloper
{
    FirstName = d.FirstName,
    LastName = d.LastName
};

Func<Developer, bool> isGSD = d => d.YearsOfExperience >= 5 &&
  d.IsFunctional;
      if (predicate(item))
IEnumerable<GoodSeniorDeveloper> goodSeniorDevelopers =
              Map(Filter(developers, isGSD), convertToGSD);
}
```

Higher-Order functions

A higher-order function is a function that takes another function as a parameter, or a function that returns another function as a value, or a function which does both.

```
Func<Developer, GoodSeniorDeveloper> convertToGSD = d => new
  GoodSeniorDeveloper
{
    FirstName = d.FirstName,
    LastName = d.LastName
};

Func<Developer, bool> isGSD = d => d.YearsOfExperience >= 5 &&
  d.IsFunctional;

IEnumerable<GoodSeniorDeveloper> goodSeniorDevelopers =
                developers.Where(isGSD)
                          .Select(convertToGSD);
```

Higher-Order functions

A higher-order function is a function that takes another function as a parameter, or a function that returns another function as a value, or a function which does both.

```
let getGoodSeniorDeveloper developers =
        // Developer -> GoodSeniorDeveloper
        let convertToGSD (d:Developer) = {FirstName = d.FirstName;
          LastName=d.LastName};
        // Developer -> bool
        let isGoodSeniorDeveloper d = d.IsFunctional && d.YearsOfExperience >= 5

        developers
        |> List.filter isGoodSeniorDeveloper
        |> List.map convertToGSD
```

Currying and Partial Application

Currying is the process of rewriting function with multiple parameters to a series of functions, each with only one parameter.

Currying is done **automatically** by the compiler.

Partial application is when function applied with some of the parameters and return new function

Currying and Partial Application

C#

```
public static Func<int, int> Add(int x)
{
    return y => x + y;
}

public static Func<int, int> Add1() => Add(1)

int valueA = Add  (1)(3); // Add  (1:x) (3:y)
int valueB = Add1 (3);    // Add1 (3:y)
```

Currying and Partial Application

F#

```
let add x y = x + y
let add1 = add 1
let value = add1 2
```

Values

Assign **name** to **value**

Not like C# **variables**

Usually **values** is **separated** from **behavior**

```
let num = 1
let str = "Hii"
let person = Person()
```

Recursion

Used a lot in functional programming

Used instead of loop

Simplify traversing complex data structure

Tail Call Optimization

F#

```
let rec factorial n =
    if n = 0 then
        1
    else n * factorial (n - 1)
```

Recursion

F#

```
let tailRecFactorial n =
    let rec fact n acc =
        if n = 0 then
            acc
        else
            fact (n - 1) acc * n
    fact n 1
```

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

-- Edsger Dijkstra