What & Why Functional Programming in C# & F#

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

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

Riccardo Terrell

Agenda

- History and influence of Functional Programming
- What is Functional Programming
- Functional Programming characteristics (Why FP take 1)
- Why Functional Programming (Why FP take 2)

Something about me – Riccardo Terrell

- Originally from Italy. I moved here ~8 years ago
- +/- 16 years in professional programming
  - C++/VB → Java → .Net C# → C# & F# → ??
- Organizer of the DC F# User Group
- Software Architect @ Microsoft
- Passionate in Technology, believer in polyglot programming as a mechanism in finding the right tool for the job
- Lucky husband

Objectives

- **Use functions as building block and composition to solve complex problem with simple code**
- **Immutability and Isolation are your best friends to write concurrent application**
  - Concurrency and reliability are becoming more and more required, FP will help you!
- **Functional and imperative styles should NOT stand in opposition …**

  … they should COEXIST
  - ***Poly-paradigm programming is more powerful and effective than polyglot programming***

History

λ

1930

1958 LISP

1962 APL

1966 ISWIM

1973 ML

1977 FP

1988 Haskell

1986 Erlang

1996 Ocaml

2003 Scala

2005 F#

2007 Clojure

Functional Programming Influence

- Continue Evolution of GC – Lisp ~1958
- Generics – ML ~ 1973
- In early 1994, support for lambda, filter, map, and reduce was added to Python
- List comprehension - Linq in C# 3.0 and Java 8
- First-class functions were also introduce Visual Basic 9, C#, Java 8 and C++ 11
- Java 8 supports lambda expressions as a replacement for anonymous classes
- Is Lazy Evaluation is part of the main stream languages such as C#
- Type Inference, example is the "var" keyword in C#

Programming in a functional style can also be accomplished in languages that aren't specifically designed for functional programming, Lots of modern languages have elements from FP

Real World Uses

F#	Microsoft (Halo-Game), Credit Suisse
Haskell	AT & T, Scrive
Erlang	Ericsson, Amazon (SimpleDB), T-Mobile, Facebook, WhatsApp
Scala	Twitter, LinkedIn

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

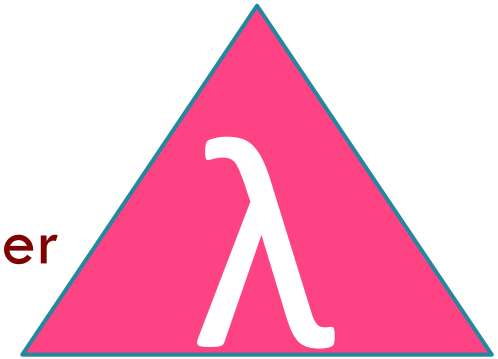
What is "Functional Programming?"

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

Functional characteristics

- Higher Order Function
- Immutability
- Composition
- Curry & Partial Application
- Recursion

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
```

Pure Functions - Properties and Benefits

- *A function always gives the same output value for a given input value*
  - *I can then cache the result*
- *A Pure Function has no side effects*
- Pure functions can be executed in parallel
- The function does not rely on any external state
  - *Increased readability and maintainability*
- Pure functions can more easily isolated
  - *easier testing and debugging*

Pure Functions - Properties and Benefits

```
private string aMember = "StringOne";

public void Concat(string appendStr)
{
        aMem[illegible] = '-' + appendStr;
}
```

Bad!

Pure Functions - Properties and Benefits

```
public string Concat(string s, string appendStr)
{
        return (s + '-' + appendStr);
}
```

Pure Functions - Properties and Benefits

Side Effects

Side effects aren't bad…

Unwanted Side Effects are Evil!

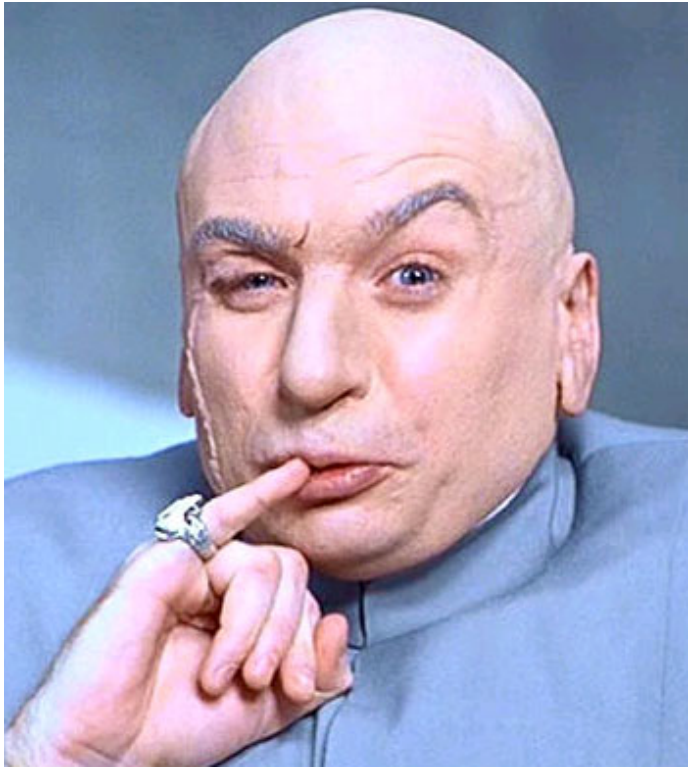
Side effects aren't bad…

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



Immutability

- Immutability is very important characteristic of FP but mostly is about transformation of state…
  Immutable data forces you to use a “transformational” approach

- When you’re writing programs using immutable types, the only “thing a method can do is return a result, it can’t modify the state of any objects
- Immutable data makes the code predictable is easier to work
- Prevent Bugs
- Concurrency is much simpler, as you don’t have to worry about using locks to avoid update conflicts
  - Automatic thread-safe, *controlling mutable state in concurrency is very hard*

Immutability

Mutable values cannot be captured by closures!

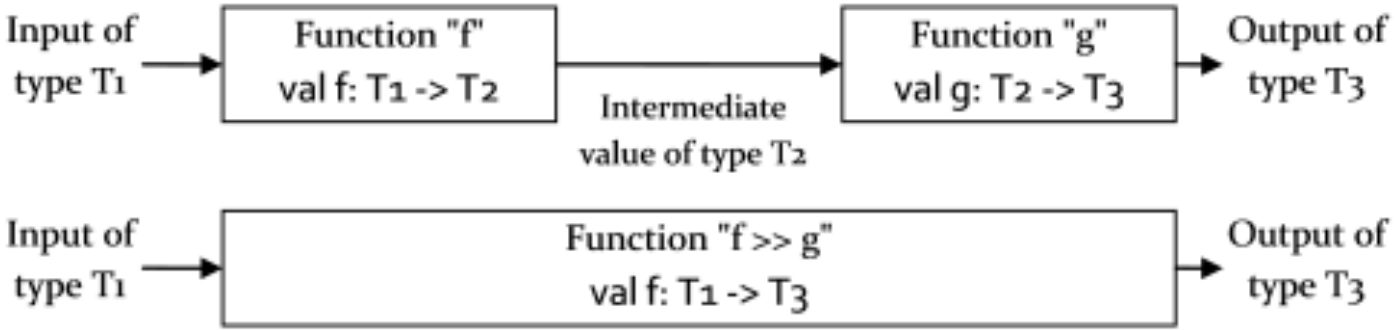
```
List<Func<int>> actions = new List<Func<int>> ();
for (int i = 0; i < 5; i++)
        actions.Add (() => i * 2);

foreach (var action in actions)
        Console.WriteLine (action ());
```


```
let actions = List.init 5 (fun i -> fun() -> i * 2)
for action in actions do
    printfn "%d " (action())
```

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



Using functions as building blocks

```
// string -> bool
let filterDocx = (fun f -> Path.GetExtension(f) = ".docx")
// string -> FileInfo
let toFileInfo = (fun file -> new FileInfo(file))
// FileInfo -> int64
let getLen = (fun (info:FileInfo) -> info.Length)

// string[] -> string[]
let filter = Array.filter filterDocx
// string[] -> FileInfo[]
let mapFileInfo = Array.map toFileInfo
// FileInfo[] -> int64[]
let allLen = Array.map getLen

// string -> int64
let getSizeDirectory =
    getFiles >> filter >> log >> mapFileInfo >> allLen >> Array.sum
```

Using functions as building blocks

```
// string -> bool
let filterDocx = (fun f -> Path.GetExtension(f) = ".docx")
// string -> FileInfo
let toFileInfo = (fun file -> new FileInfo(file))
// FileInfo -> int64
let getLen = (fun (info:FileInfo) -> info.Length)

// string[] -> string[]
let filter = Array.filter filterDocx
// string[] -> FileInfo[]
let mapFileInfo = Array.map toFileInfo
// FileInfo[] -> int64[]
let allLen = Array.map getLen

// string -> int64
let getSizeDirectory =
    getFiles >> filter >> log >> mapFileInfo >> allLen >> Array.sum
```

Sa far we talk about …

- Higher Order Function & Pure Function
- Immutability
- Composition

Compose Mario

Mario game sample, based on functional reactive Elm script (http://elm-lang.org/edit/examples/Intermediate/Mario.elm)

The functions that depend on keyboard take the current keyboard state as the first argument. This is represented as a tuple int*int (dir) consisting of x and y directions.

```
// If the Up key is pressed (y > 0) and Mario is on the ground,
// then create Mario with the y velocity 'vy' set to 5
let jump (_,y) m = if y > 0 && m.y = 0. then  { m with vy = 5. } else m
// If Mario is in the air, then his "up" velocity is decreasing
let gravity m = if m.y > 0. then { m with vy = m.vy - 0.1 } else m
// Apply physics - move Mario according to the current velocities
let physics m = { m with x = m.x + m.vx; y = max 0. (m.y + m.vy) }
// When Left or Right keys are pressed, change the 'vx' velocity and direction
let walk (x,_) m =
    { m with vx = float x
             dir = if x < 0 then "left" elif x > 0 then "right" else m.dir }
```

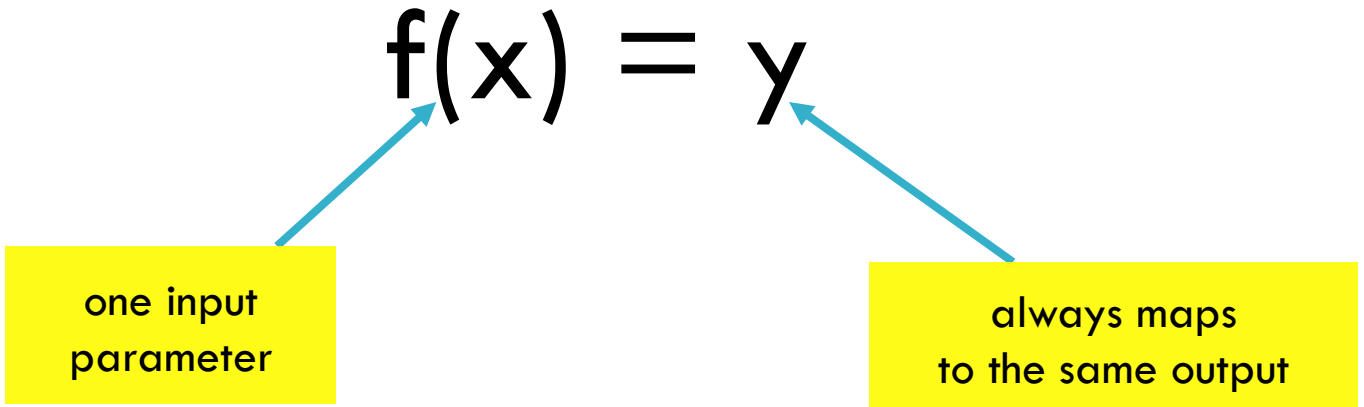
The step function of the game takes previous Mario value and returns a new one. It is composed from 4 functions that represent different aspects of the game.

```
let step dir mario = mario |> physics |> walk dir |> gravity |> jump dir
```


DEMO

Functions

Remember functions from high school math?



But I need multiple input parameters!

Currying

```
let f x y z =
    let w = 1
    … //do something
    w //return
```

wrapping single parameter functions within functions

$f(x\ \ g(y\ \ h(z))) = w$

```
f: x -> (y -> (z -> w))
```

function signature

Partial Application

- You don't really need to know currying
- Instead understand the inverse of currying
- Unwrap the parameters of a function
- Unwrapping is called "Partial Application"

Partial Application

Partial Application refers to the process of fixing a number of arguments to a function, producing another function of smaller arity

```
let multiply x y = x * y
let multiply = fun x -> fun y -> x * y
let multiplyBy2 = multiply 2

let result_is_10 = multiply 2 5
let result_is_12 = multiplyBy2 6
```

```
> y => x * y;
ly(x)(2);
```

With both currying and partial application, you supply argument values and return a function that's invokable with the missing arguments. But currying a function returns the next function in the chain, whereas partial application binds argument values to values that you supply during the operation, producing a function with a smaller arity

Currying & Partial Application

```
public string print(string name, int age) /*(...)*/

public Func<string, Func<int, string>> curry(Func<string, int,string> f){
    return (name) => (age) => f(name, age);
}

var curriedPrint = curry(print);
string cp = curriedPrint("Ricky")(39);

Func<int, string> curriedPrint2 = curriedPrint("Ricky");
string cp2 = curriedPrint2(39);
```

Currying & Partial Application

```
// string -> int -> string
let print name age =
    sprintf "name %s age %d" name age

let print' =
    (fun name -> fun age ->
    sprintf "name %s age %d" name age)

// int -> string
let printAge age = print "Ricky" age
let printAge = (fun age ->  print "Ricky" age)// age
let printAge = print "Ricky"
```

Recursion is the new iteration

- In functional programming “iteration” is ~usually~ accomplished via recursion
  - Recursive function invoke themselves, Tail recursion is optimized by compiler avoiding to maintain a stack ~stack over flow!

```
let rec getData (reader : SqlDataReader) list =
    match reader.Read() with
    | true ->   let id = Convert.ToInt32(reader.["ContactID"])
                let fname = reader.["FirstName"].ToString()
                let lname = reader.["LastName"].ToString()
                let person = {PersonID = id; FirstName = fname; LastName = lname}
                getData reader (person :: list)
    | false -> reader.Close()
               list

getData reader []
```


DEMO

Why Functional Programming?

Why Functional Programming?

- Changes the way you think about programming and problem solving
- Functional Programming promotes Composition and Modularity
- Simple Code for Complex Problems
- Declarative programming style
- Concurrency, FP is easy to parallelize
- Conciseness, less code *(no null checking)*
- Correctness & Reduced bugs *(immutability)*
- Rapid Prototyping

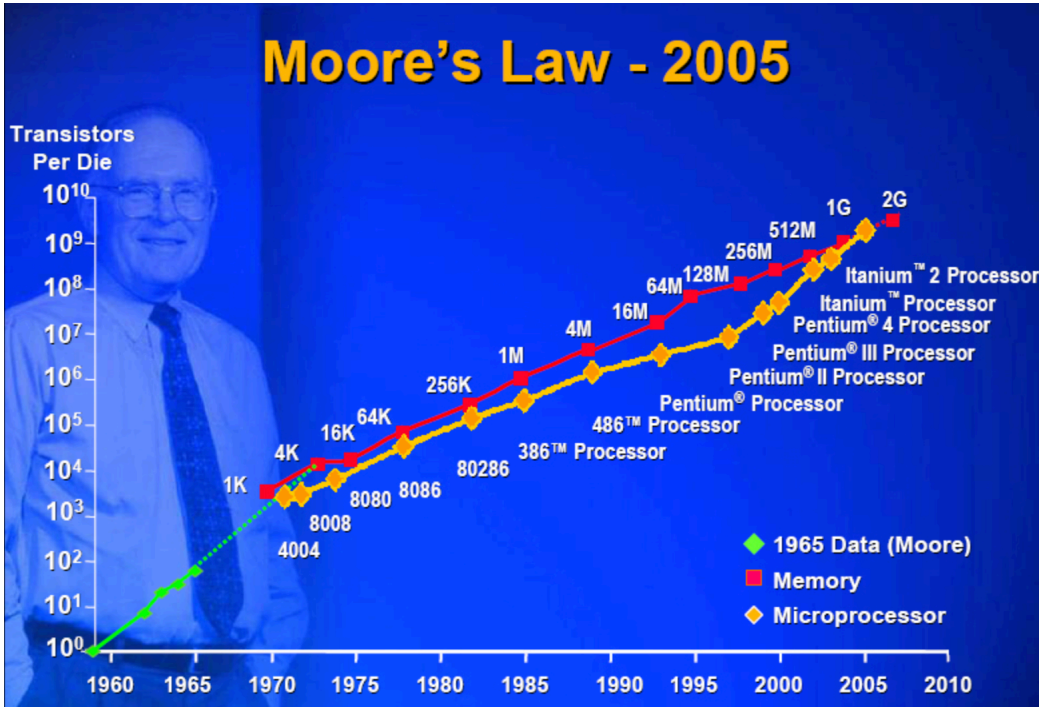
Concurrency

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

Moore's law http://en.wikipedia.org/wiki/Moore's_law

… to achieve great performances the application must be leveraging a

Concurrent Model

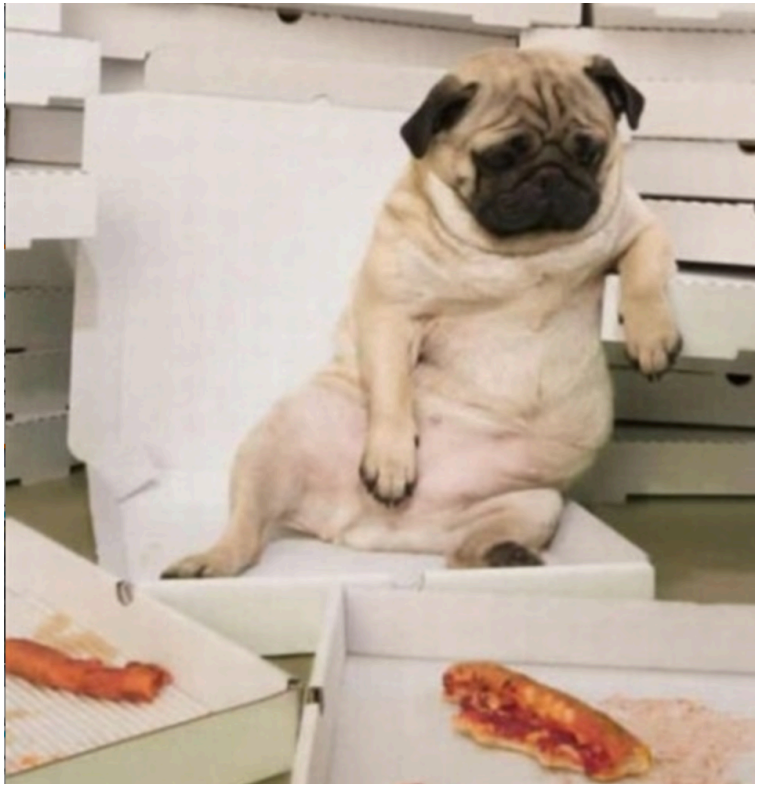


Concurrency

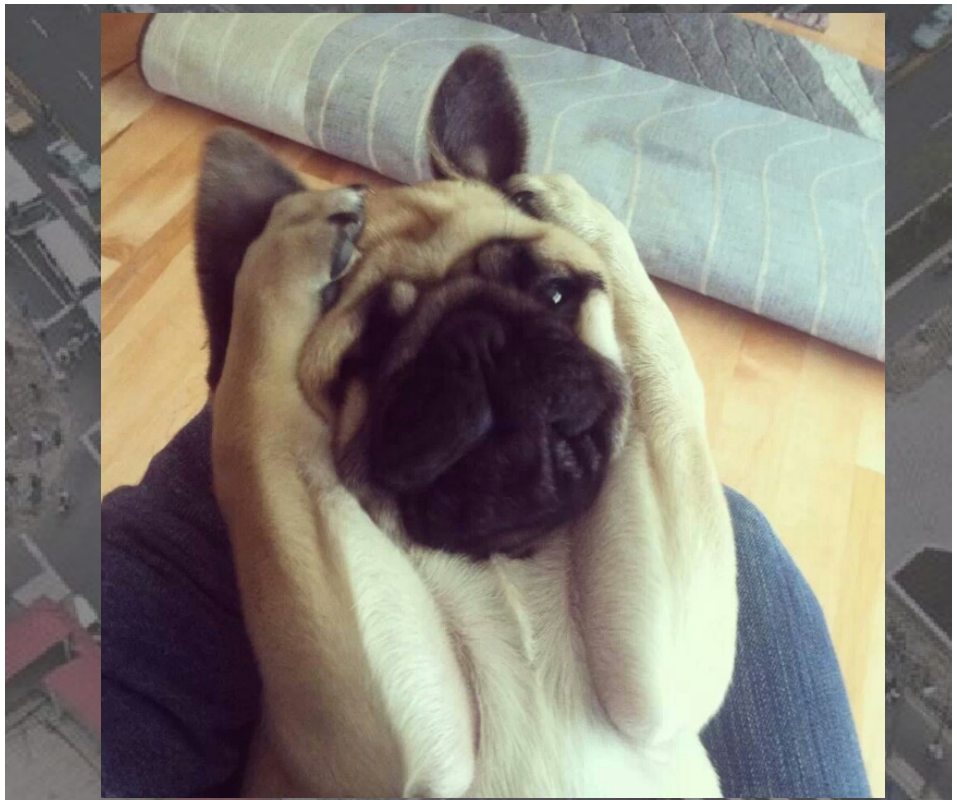
There is a problem…

the free lunch is over

- Programs are not doubling in speed every couple of years for free anymore
- We need to start writing code to take advantage of many cores



Concurrency… jams to avoid!



Parallel, asynchronous, concurrent, and reactive programs bring many challenges because these programs are nearly always *nondeterministic*

- This makes debugging/testing challenging
- Deadlocking
- Not easy to program
- Concurrency is a double-edged sword

Concurrency

- **Functional programming feels good:**
  - *Thanks to the **immutability** (and **isolation**), we avoid introducing race conditions and we can write lock-free code.*
  - *Using a **declarative programming** style we can introduce parallelism into existing code **easily**. We can replace a few primitives that specify how to combine commands with version that executes commands in parallel*

Concurrent Model Programming

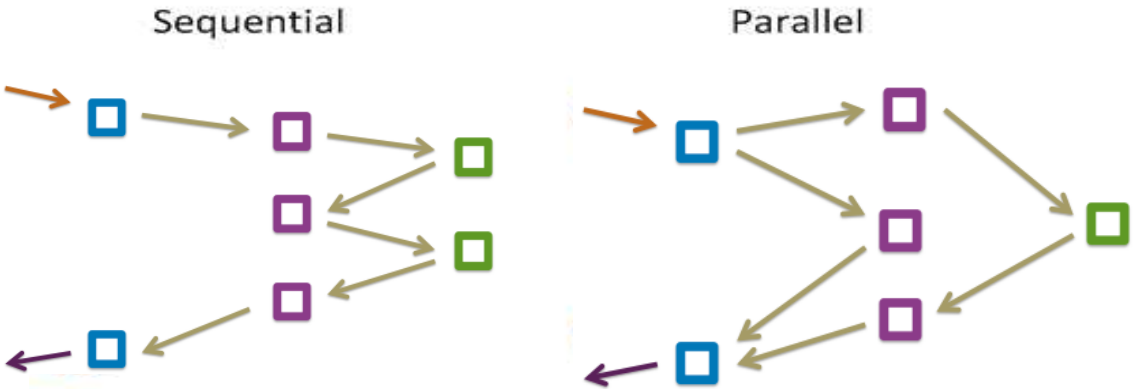
An Actor is an independent computational entity which contains a queue, and receives and processes messages

It provides immutability and isolation

(it enforces coarse-grained isolation through message-passing)

Actor Model

What does a system of actors look like?

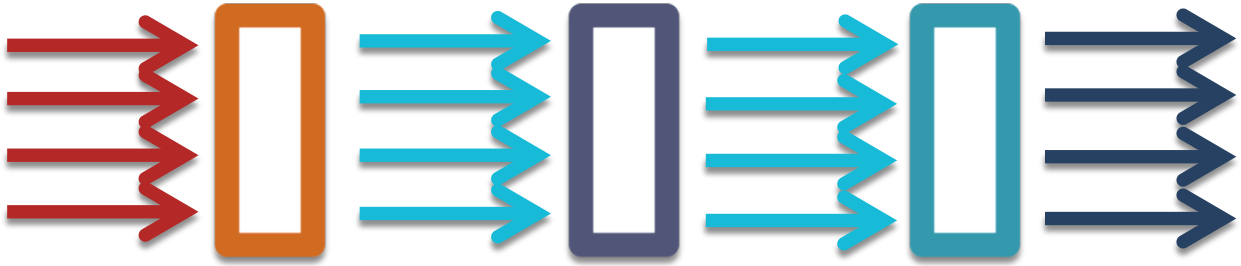


Agent-based

What does an Actor look like?

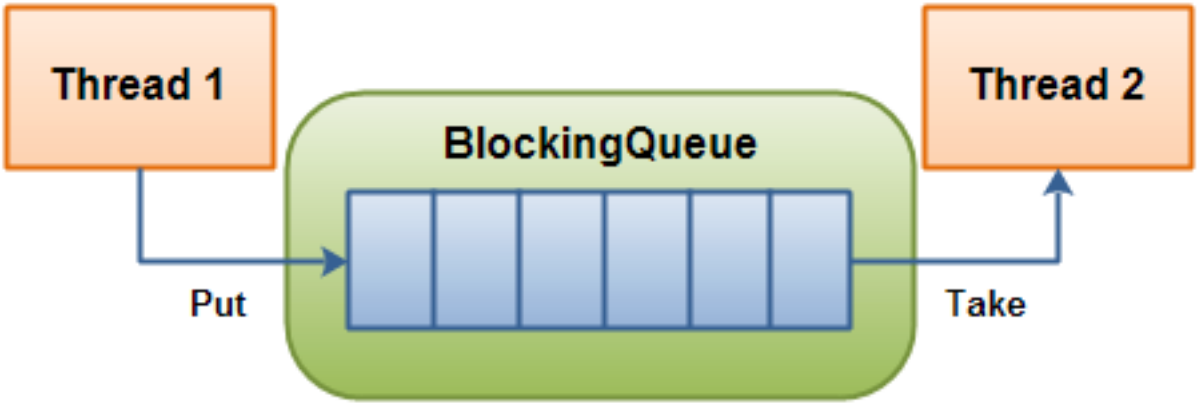
```
type Message = Add of int | Result of AsyncReplyChannel<int>

let agent = Agent.Start(fun inbox ->
        let rec loop count = async{
            let! msg = inbox.Receive()
            match msg with
            |Add n -> return! loop (count + n)
            | Result reply -> reply.Reply(count)
                              return! loop count }
        loop 0 )

agent.Post( Add(8) )
let result = agent.PostAndReply(Result)
```

Pipeline Processing



- Pipeline according to Wikipedia:
  - A pipeline is a set of data processing elements connected in series, so that the output of one element is the input of the next one. The elements of a pipeline are often executed in parallel or in time-sliced fashion; in that case, some amount of buffer storage is often inserted between elements

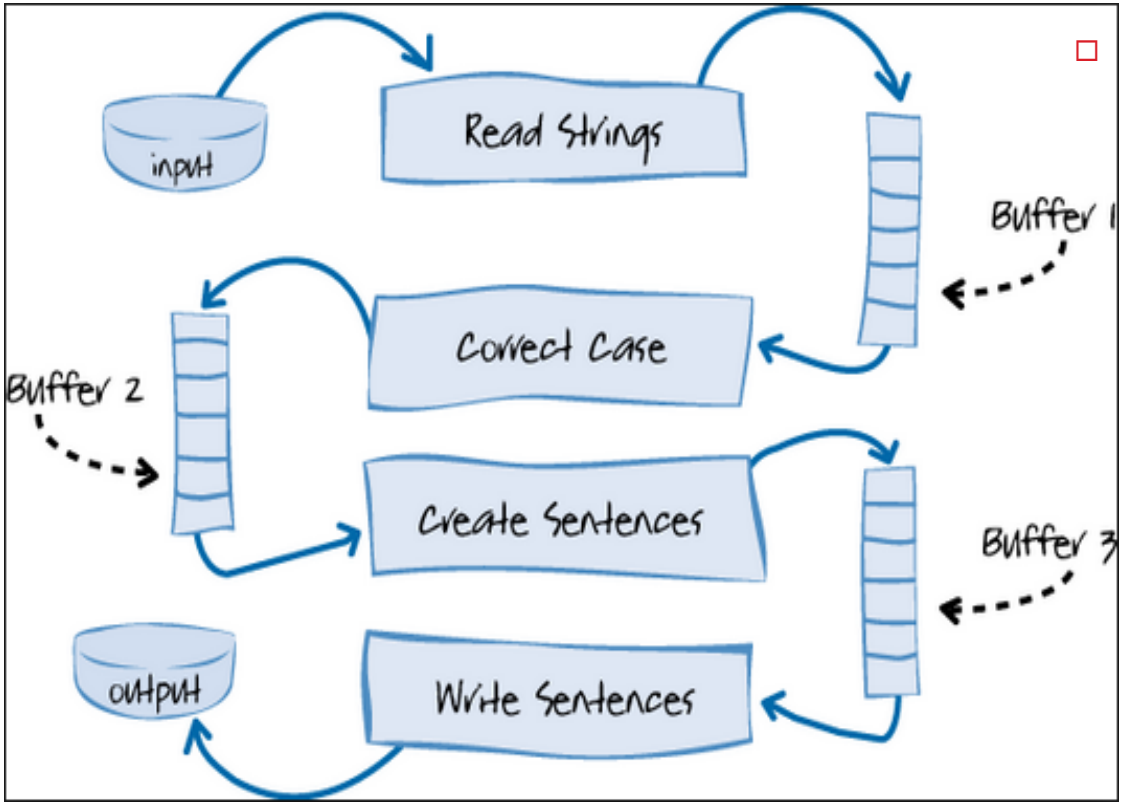
Agent Async-BoundedQueue



Agent Async-BoundedQueue

```
let agent = MailboxProcessor.Start((fun inbox ->
        let rec loop() = async {
                let! msg = inbox.Receive()
                match msg with
                | Enqueue(x, reply) -> waitingProducers.Enqueue (x, reply)
                | Dequeue reply -> waitingConsumers.Enqueue reply
                balance()
                return! loop() }
        loop()), cancelToken.Token)

        balance()
```

Pipeline Processing



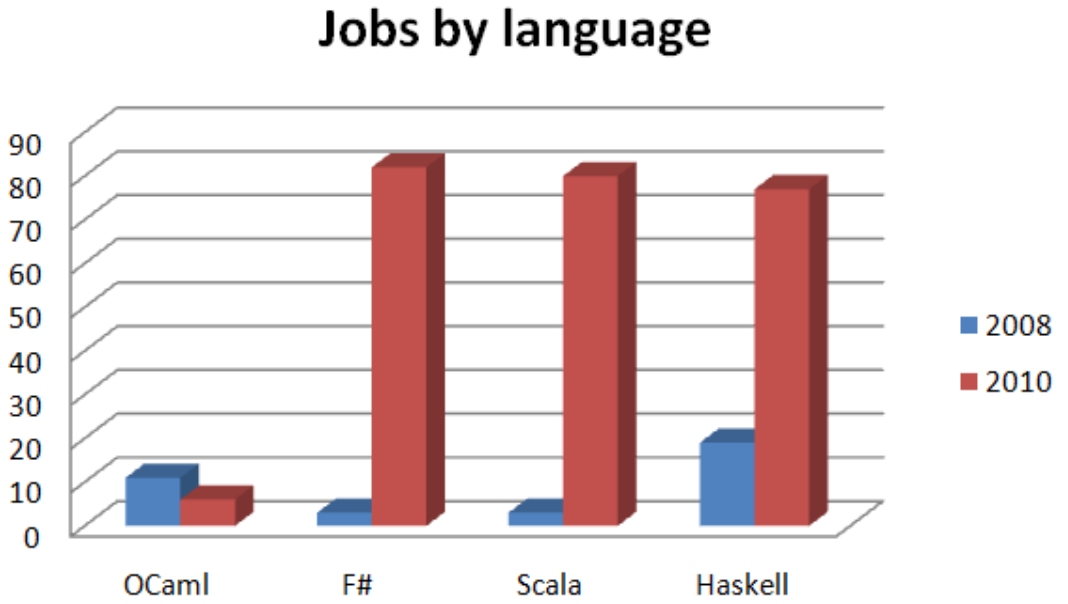
- Values processed in multiple steps
  - Worker takes value, processes it, and sends it
  - Worker is blocked when source is empty
  - Worker is blocked when target is full
  - Steps of the pipeline run in parallel


DEMO

When Functional Programming?

Functional Job Trend

Functional languages are sprouting not just on the JVM where the two most interesting new languages are Scala and Clojure, but also on the .NET platform, where F# is a first-class citizen



http://www.itjobswatch.co.uk

Wrap-Up

- **Use functions as building block and composition to solve complex problem with simple code**
- **Immutability and Isolation are your best friends to write concurrent application**
  - Concurrency and reliability are becoming more and more required, FP will help you!
- **Functional and imperative styles should NOT**

  …they should COEXIST
  - ***Poly-paradigm programming is more powerful and effective than polyglot programming***

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

-- Edsger Dijkstra

How to reach me

github.com/Rikace/WhyFunctionalITPro2015

meetup.com/DC-fsharp/

@TRikace @DCFsharp

rterrell@microsoft.com