Why F# ?

by Aly-Bocar CISSE on November 2021

updated on October 2024

“Point of view is worth 80 IQ points”

Alan Key

What is F# ?

General purpose language on .NET designed by Don Syme, from Microsoft Research

What is F# ?

- Strongly, statically typed, open sourced and cross platform language from Microsoft with deep inference
- 1.0 appeared on 2005 (older than: GO, Rust, Kotlin, Typescript ...)
- Heavily influenced by OCaml, Haskell & C#
- Multi-paradigm language : functional (by default), imperative and object-oriented
- Fully supported by .NET. Compiles to JavaScript

About functional paradigm…

Before going further…

Functional vs Object is irrelevant

Those are two different things with pros and cons…

Functional paradigm is old

Theory around since the 30's and first functional language in the 50's.

No need for math

…to deliver production ready software

…to use most functional programming languages, especially F#

Functional paradigm focus on data

…while Object Oriented paradigm focus on structuring links between data and methods

5 reasons to have a hard look at F#

...when coming from C#

#1 - F# is still .NET

F# in my C# and C# in my F#...

#1 - F# is still .NET

```
//This is a function.
//It wraps the call to the LocalDate object. LocalDate comes from NodaTime a C# library
//No need for the 'new' keyword here
let mkDate (year: Year) (month: Month) (day: Day): LocalDate =
        LocalDate(year, month, day)
```

- Same constraints as VB.NET
  - Anything written in C# can be used in F#
  - The reverse is true for all F# features not depending on the compiler

#1 - F# is still .NET

```fsharp
open System

type Month = int
type Year = int

let mkDate (y:Year) (m:Month) : DateOnly =
    DateOnly(1,m,y)
```

```csharp
using System;
using System.Reflection;
using Microsoft.FSharp.Core;

[assembly: FSharpInterfaceDataVersion(2, 0, 0)]
[assembly: AssemblyVersion("0.0.0.0")]
[CompilationMapping(SourceConstructFlags.Module)]
public static class @_
{
    [CompilationArgumentCounts(new int[] { 1, 1 })]
    public static DateOnly mkDate(int y, int m)
    {
        return new DateOnly(1, m, y);
    }
}
namespace <StartupCode$_>
{
    internal static class $_
    {
    }
}
```

Let’s have a look deeper look here

#2 - F# modern type system

Not an opinion, just have a look at the language design space of the last 15 to 20 years…

#2 – Product type

```
public class Something {
  public int SomeNumber { get; set; }
  public string SomeSentence {get; set;}
}
```

- An instance of class **Something**, can have any combination of **int** AND **string** values
- We say that **Something** is a product type of **int** and **string**, like (**int*string**) or **Tuple<int,string>**

#2 – Product type

```
type TupleAlias = int*string //let a : TupleAlias = (1,"b")

type Something' = Ctr of int*string // let b : Something' = Ctr(1,"b")

type ARefTypeRecord = { SomeNumber : int ; SomeSentence : string }
//let c : ARefTypeRecord = { SomeNumber = 1; SomeSentence = "b" }

type [<Struct>] AStructTypeRecord = { SomeNumber : int ; SomeSentence : string }
//let c : AStructTypeRecord = { SomeNumber = 1; SomeSentence = "b" }
```

#2 – Product type

If you wish for a class…

```
type Something() =
  member val SomeNumber = 0 with get, set
  member val SomeSentence = "" with get, set
```

#2 – Sum type

- A sum type allows for DIFFERENT representations for a type
- For example, you can represent in math a complex number as follow :
  - `r` θ `i`
  - `a + b i`

#2 – Sum type

```
type UserAndPassword = { Login : string ; Password : string }

//discriminated union here !
type Credentials =
| Token of string
| Classic of UserAndPassword

type User =
  {
    Cred : Credentials
    //other stuff there
  }

Credentials -> bool
let logInToSystem(cred:Credentials) =
  let hasSignedIn = false
  //imagination time ...
  hasSignedIn
```

#2 - Sum type

No sum type in C#, but you got Polymorphism…

```
public abstract class Credentials { //...
}

public class ClassicCredentials : Credentials {
  public string User { get; set; }
  public string Password { get; set; }
}

public class TokenCredentials : Credentials {
  public string Token {get;set;}
}

public interface IHaveCredentials {
  Credentials GetCredentials();
}

public class LoginSystem {
  public bool LogIn(IHaveCredentials subject) {
    //Imagine some code here ...
  }
}

public class UserWithLoginAndPassword : HasCredentials {
  //...
}

public class UserWithToken : HasCredentials {
  //...
}
```

#2 – Sum type

- In short, a sum type is a type with multiple representations
- They are called **Discriminated Union** in F#
- Not a new concept, you'll find them in TypeScript, Rust, Kotlin…

#3 - F# is immutable by default

…yes, it's a very big deal, have a look at C# records or Immutable collections…

#3 - F# is immutable by default

- Far less complicated to write concurrent code in F#
- Initialisation is at declaration
- F# supports **null** for compatibility with the .NET runtime
  - no implicit **null** assignation is done
  - only pure F# types are not nullable (still usable in C#)

#3 - F# is immutable by default

If mutability is what you want or need ...

```
string
let mutable myNameIs = "@essicf37"
myNameIs <- "Slim Shady"
```

Immutability have very nice properties but also some cons, have a look at [@KevlinHenney great 2018 talk] on the subject…

#4 - Pattern matching

…powerful and expressive !

#4 - Pattern matching

```
type LoginAndPassword = { login: string; password : string }
type Credentials =
    | Token of string
    | Classic of LoginAndPassword

Credentials -> unit
let logInToSystem (subject:Credentials) =
  match subject with
    | Token t -> () // We do something with it...
    | Classic c -> () // We do something else with it..
```

#4 - Pattern matching

```
string list -> unit
let printInfoOnList (a: string list) =
  match a with
  | [] -> printfn "List is empty !"
  | [_; _] -> printfn "List has only 2 elements"
  | _ -> printfn "List has more than 2 elements"
```

#4 - Pattern matching

```
'a -> unit
let someFunc a =
  let aBigTuple = ("@essiccf37","c# developer","f# deveoper")
  let handle, skill1, skill2 = aBigTuple
  //...
  ()
```

*F# allows for much more, you can check out [Microsoft doc] or the excellent site of **Scott Wlaschin** : [F# for fun and profit] for more*

#5 – F# is interactive

…work on .NET while feeling like a JS developer

#5 – F# is interactive

- F# has a REPL call FSI
- FSI is supported on all major IDE for a while now
- REPL is useful: ask Python and JavaScript developers or just dig into the history of the CScript initiative from MS…
  - You can create scripts (FAKE)
  - You can write code and try it directly
  - You can test use cases with DLLs from production to see what's happening

Let’s see some code!

F# Samples

```fsharp
[<EntryPoint>]
string [] -> int
let main argv =
    if argv.Length > 1 then printfn $"Hello world {argv.[0]}"
    else printfn "Hello world !"
    0
```

F# Samples

```
string -> int
let helloWorldFunc target =
    printfn $"Hello %s{target}!"
    0

[<EntryPoint>]
string [] -> int
let main argv =
    let target =
        if argv.Length > 1 then argv.[0]
        else "world"
    helloWorldFunc target
```

F# Samples

```
string -> int
let helloWorldFunc target =
    printfn $"Hello %s{target}!"
    0
string -> string array -> string
let tryGetTargetOrReturnDefaultValue defaultValue (source:string array) =
    if source.Length > 1 then source.[0]
    else defaultValue

// We use partial application to create a new function...
string array -> string
let newTryGet = tryGetTargetOrReturnDefaultValue "world"

[<EntryPoint>]
string [] -> int
let main argv =
    // '|>' is the pipe forward operator...
    // we can definite as (|>) : a-> ( a -> b) -> b
    argv |> newTryGet |> helloWorldFunc
    //same as ... helloWorldFunc ( newTryGet argv )
```

F# Samples

```fsharp
[<EntryPoint>]
string [] -> int
let main argv =
    //Array is a module which contains functions on the Array type
    // Array.tryHead : 'T[] -> 'T option
    // 'T option = Some of 'T | None
    match argv |> Array.tryHead with
    | Some target -> printfn $"Hello {target}!"
    | None -> printfn "Hello world!"
    0
```

In short…

- F# has been production ready for a while now…
- F# is another viable option on the .NET platform
  - A good fit for Business Line application
  - Immutability by default makes it less of a good fit when latency and raw performance is critical
- F# has influenced C# for at least a decade now…

Thank you !

Questions ?

Want to code?

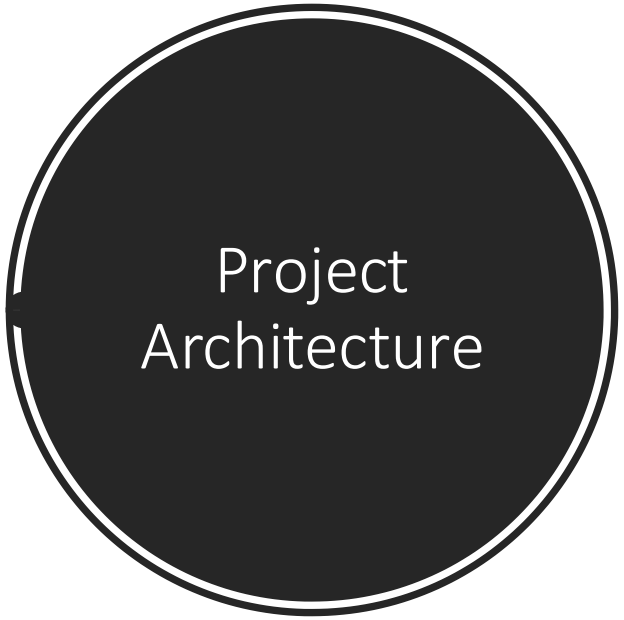
F# Workshop !

Let’s code something simple in a realistic production ready software architecture

We will implement a simple use case of a Time Tracker api in F# of course.

We aim here :

- to show a software architecture which take full advantage of the functional paradigm
- to show what it’s like to write C# code in a real life (like) scenario
- to show how to work alongside C#



We are using here **Functional Core/Imperative Shell** pattern (or architecture)

Functional Core is where the business logic (domain) resides. It must be:

- free of side effects
- fully synchronous

Imperative Shell is where use cases are implemented using functions from the Core. It’s also where side effects are handled.

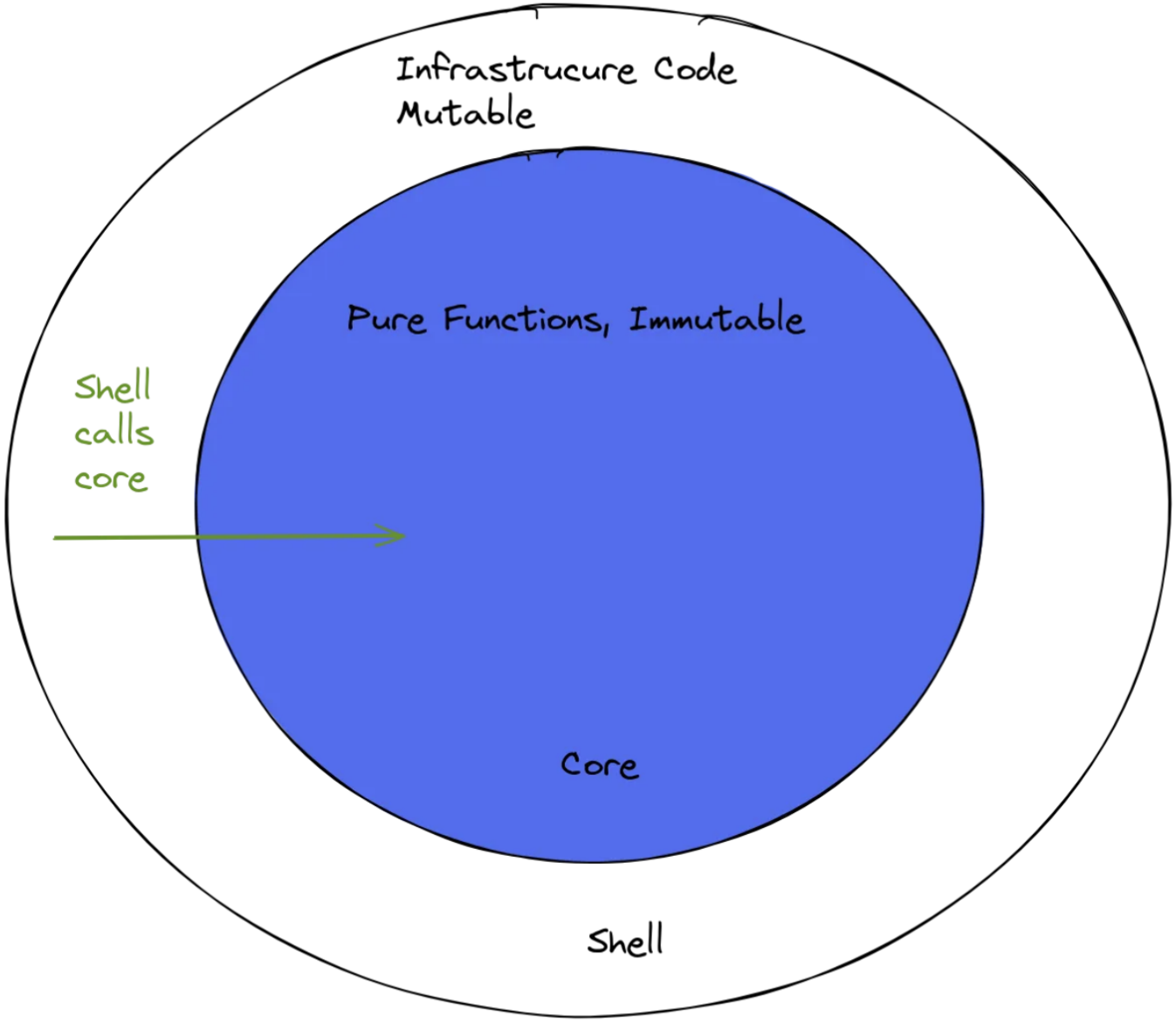


Figure 3. Imperative shell and functional core.

From a blog post by Mario Bittencourt

Let’s do this !