Intro Course in F#

Reliable, adj: To deliver the same result every time

2017-09-21, PROSA (PASCAL) @ Copenhagen

Overview

- About me (very shortly)
- Matching of expectation
- Agenda:
  - **17:00** |> A few basic concepts to get started
  - **17:15** |> Type (Domain) Driven Development
  - **18:00** |> .NET-applications and libraries
  - **18:45** |> Data and ~~TypeProviders~~
  - **19:30** |> Concurrency and parallelism
  - **20:15** |> Robust and error-free applications
  - **21:00** |> Summary

About me (very shortly)

- Ramón Soto Mathiesen (Spaniard + Dane)
- MSc. Computer Science ***DIKU/Pisa*** and minors in Mathematics ***HCØ***
- ***CompSci*** @ SPISE MISU ApS
  - *“**S**tay **P**ure, **I**solating **S**ide-**E**ffects”* -- Michael Werk Ravnsmed dixit
  - *“**M**ake **I**llegal **S**tates **U**nrepresentable”* -- Yaron Minsky dixit
  - Trying to solve EU GDPR with a scientific approach (Computer Science and Mathematics)
  - ***Elm*** (***JS*** due to ports) with a bit of ***Haskell*** and a bit of ***F#*** (fast prototyping)
- Elm / Haskell / TypeScript / F# / OCaml / Lisp / C++ / C# / JavaScript
- Founder of Meetup for F#unctional Copenhageners (MF#K)
- Volunteer at Coding Pirates (Captain at Valby Vigerslev Library Department)
- Blog: http://blog.stermon.com/ and Twitter: @genTauro42

F# Open Source projects

- Previous workplace (CTO of CRM @ Delegate A/S):
  - MS CRM Tools:
    - http://delegateas.github.io/
  - ~~Delegate.Sandbox~~:
    - http://delegateas.github.io/Delegate.Sandbox/
- Current workplace (SPISE MISU ApS):
  - Syntactic Versioning (SynVer @ F# Community Projects)
    - Mostly driven by Oskar Gewalli (@ozzymcduff)
  - Puritas, isolated side-effects at compile-time in F#
    - F# eXchange 2017 (talk and video)

Matching of expectations

- What are your expectations for this course?

Taken from a MF#K talk: (fun _ → why, where, how)

- Last but not least, Joakim and I have committed, in collaboration with PROSA, to provide two introductory courses in **Scala** (***Java*** people) and **F#** (***.NET*** people):
  - Date still to decide (most likely ~~February~~ or ~~March~~)
  - Free for PROSA members and a fee for non-members

PROSA
Forbundet af It-professionelle

Matching of expectations

- We expect the attendees to be able to:
  - Understand a few basic concepts:
    - The programming language is functional first
    - Algebraic data types (Sum and Product)
    - REPL, develop faster by making ad-hoc test from the IDE
    - The pipe operator and readability
  - Make production-ready applications or libraries

Overview

- **17:00** |> **A few basic concepts to get started**
- **17:15** |> Type (Domain) Driven Development
- **18:00** |> .NET-applications and libraries
- **18:45** |> Data and ~~TypeProviders~~
- **19:30** |> Concurrency and parallelism
- **20:15** |> Robust and error-free applications
- **21:00** |> Summary

A few basic concepts to get started: Functional first

- Functions are first class citizens
  - Higher-order functions → passing functions as arguments:

    `[0 .. 9] |> List.map (fun x -> x + x)`
  - Functions tend to use curried arguments:

    `fun x y -> x + y` or `fun x -> fun y -> x + y` instead of `fun (x,y) -> x + y`
- Everything is data and it's immutable by default (*)

  `let x = 42 in x <- 42 (* FS0027: This value is not mutable *)`
- The NULL concept is not used due to algebraic data types (**)

(*) - Values are used, not variables, and very few structures aren't immutable, mainly for performance purposes (ex: **arrays**)

(**) - .NET inheritance adds NULL to **strings**

A few basic concepts to get started: Algebraic data types

- **Product types**: think of it as tuples (pair, triple, ...): `(42,'c')`
  - Records are just tuples with labels:

    ```
    { foo: 42; bar: 'c' }
    ```

- **Sum types (also know as Union types)**: just think of it as disjoint sets (have no element in common). The element ***must*** be in one of the assigned disjoint sets:
  - A person is either a child or an adult:

    ```
    type Person = Child | Adult
    ```

  - Temperature is measured Celsius or Fahrenheit:

    ```
    type Temperature = C of float | F of int
    ```

Note: Record types are equivalent to single case Sum types, with named fields

```
type Baz = { baz: int; qux: char } => type Qux = Qux of baz:int * qux:char
```

A few basic concepts to get started: Algebraic data types

- With ADT you will be able to compose simpler types together in order to create more complex datastructures:

```
type ProductType1 = int * char                    (* = times)
type ProductType2 = { foo: int; bar: char }       (; = times)
type SumType = Foo of int | Bar of char           (| = addition)
```

- This is ideal for domain modeling (TDD/DDD) as it allows you to use these mathematically constriranst to "Make Illegal States Unrepresentable" -- Yaron Minsky

A few basic concepts to get started: Algebraic data types

- ADT allows you to pattern match on all branches:

```
type FooBar = { foo: int; bar: float }
let isFoo42 : FooBar -> bool = function
  | { foo = 42 } -> true
  | ____________ -> false

type Person = Child | Adult
let assertAge : int -> Person -> bool =
  fun age ->
    function
      | Child -> age <  18
      | Adult -> age >= 18
```

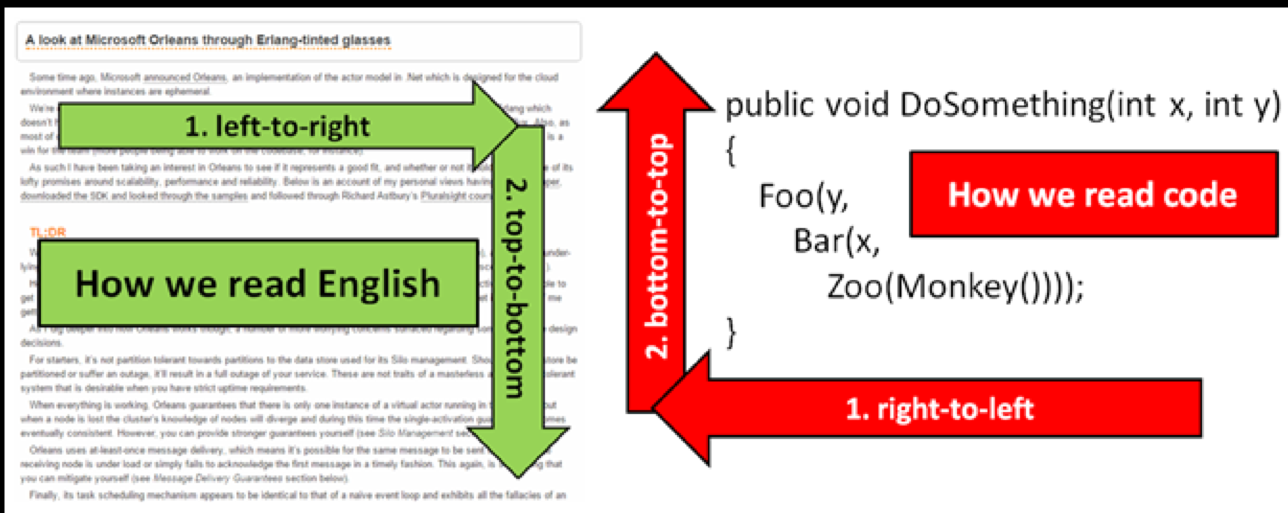
Note: For exhaustive pattern match, use the following compiler flag:

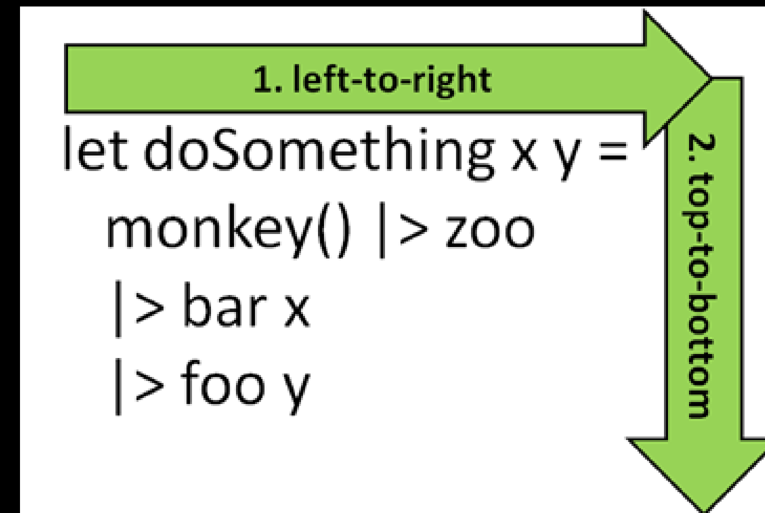
--warnaserror:25

A few basic concepts to get started: REPL

- Read, Evaluate, Print and Loop (REPL):
  - Possible to evaluate functions, modules and types directly from the IDE to F# interactive (interpreted code)
  - This makes it easy to reason about creating smaller pieces of logic and composing them to greater blocks
  - F# script files (.FSX) are also interpreted, which means that files are type checked before executing a single line

A few basic concepts to get started: |> and readability



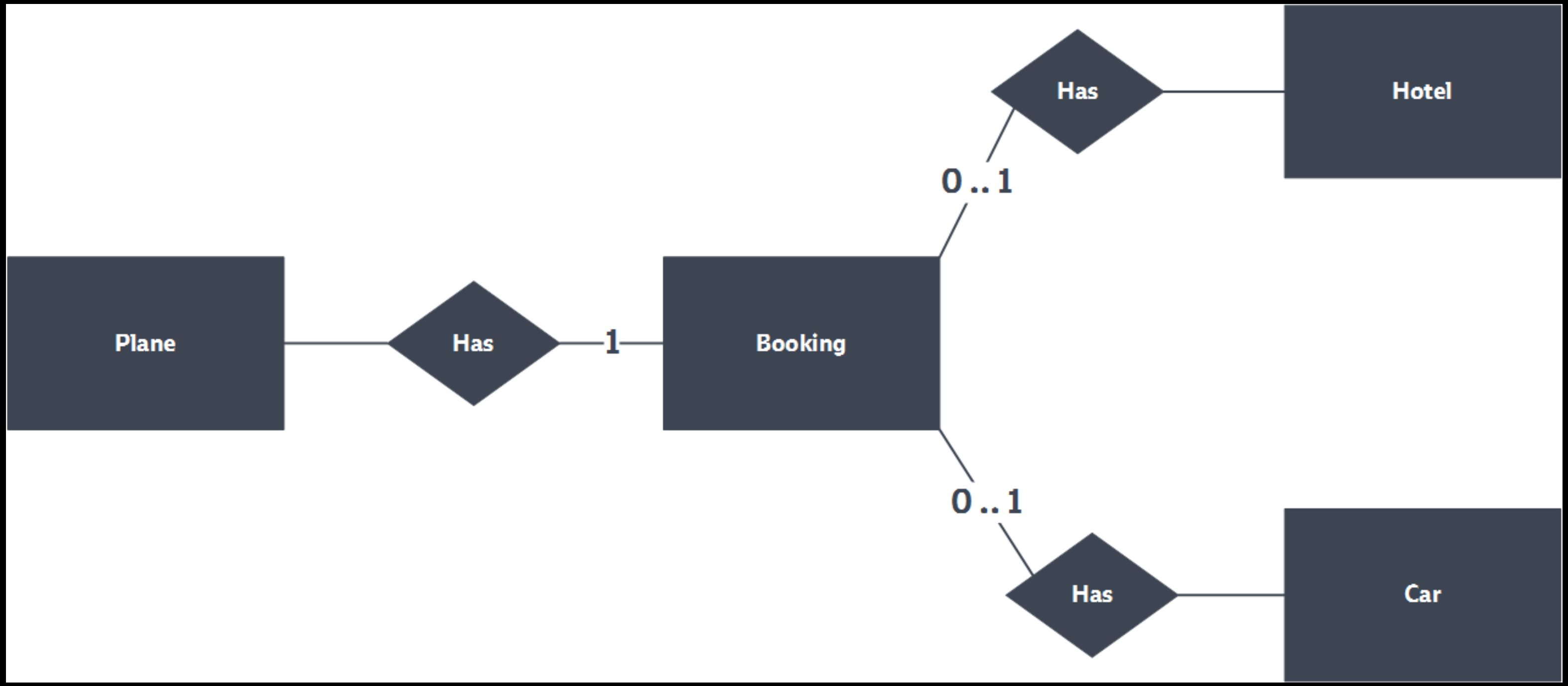


- Forced indentation, just like Python, in combination with |> makes it easy to read again and again

Overview

- **17:00** |> A few basic concepts to get started
- **17:15** |> **Type (Domain) Driven Development**
- **18:00** |> .NET-applications and libraries
- **18:45** |> Data and ~~TypeProviders~~
- **19:30** |> Concurrency and parallelism
- **20:15** |> Robust and error-free applications
- **21:00** |> Summary

Type/Domain Driven Development



Domain modelled in an ER-diagram

Type/Domain Driven Development

- It’s intuitive to see that I’m not able to make a booking unless a plane is specified (mandatory)
- Also, I can see that I might book a hotel or rent a car, but they are not required (optional)
- I don’t think I can get any other information out from this diagram unless I’m also reading some text
- Which products are they offering?

Type/Domain Driven Development

```
open System

type Booking =
   | Basic    of Plane
   | Combo    of Combo
   | FullPack of Plane * Hotel * Car
and Plane = { Outbound: DateTime; Return:    DateTime; Destination: City }
and Combo =
   | ``With Hotel`` of Plane * Hotel
   | ``With Car``   of Plane * Car
and Hotel = { Arrival:  DateTime; Departure: DateTime; Location:    City }
and Car   = { From:     DateTime; To:        DateTime; Location:    City }
and City  = String
```

Domain modelled in F# type definitions

Type/Domain Driven Development

- I can easily see the 3 product which are offered
  - Basic, Combo and Fullpack
- Combo products can be of two types
  - “With Hotel” and “With Car”

Type/Domain Driven Development

- I can see some constraints:
  - A Booking can either be Basic, Combo or Fullpack (disjoint union)
  - With each of these products requirements (tuples):
    - Basic → (Plane) single
    - Combo → (Plane,Hotel) pair or (Plane,Car) pair
    - Fullpack → (Plane,Hotel,Car) triple
  - I can also see that a Plane will require the following information (still a tuple):
    - Plane → (Outbound date and time, Return date and time, Destination country)

Notice: Domain definition and implementation are still separated with this approach

Type/Domain Driven Development

- **Tasks 01:**
  - Implement the domain of a Book, that could be used for a Bookstore or a Library:
    - Types: Audio, electronic and physical
    - Formats:
      - AAC, MP3, M4B and WAV
      - EPUB, MOBI and PDF
      - Hardcover and Paperback
    - Info:
      - Mandatory: title, authors, publisher, language, isbn10 and isbn13
      - Optional: pages

Overview

- **17:00** |> A few basic concepts to get started
- **17:15** |> Type (Domain) Driven Development
- **18:00** |> **.NET-applications and libraries**
- **18:45** |> Data and ~~TypeProviders~~
- **19:30** |> Concurrency and parallelism
- **20:15** |> Robust and error-free applications
- **21:00** |> Summary

.NET-applications and libraries

- With **libs** you expose some logic to be consumed by other **libs** or **apps**

```
module Lib =
  let someLogic () = 42
let logic = Lib.someLogic ()
```

- In **apps** you will have a main entry point function from which you will have access to the passed arguments of compiled applications:

```
[<EntryPoint>]
let main : string array -> int = fun args -> (* do *) 0
```

- **Note**: Access to passed arguments when executing script files, can be done through:

```
fsi.CommandLineArgs
```

.NET-applications and libraries

- When working with libs, limit exposure by using modules and private constructs. The functional way implies, in some cases, to keep Sum Types constructors private to modules (encapsulation):

```
module FooBar =
  type Foo = private Bar of int
  let  foo = Bar
FooBar.Bar 42
(* FS1093: The union cases … not accessible from this code location *)
FooBar.foo 42
(* > val it : FooBar.Foo = FSI_0007+FooBar+Foo *)
```

  Remark: If libraries have interoperability with the rest of the .NET ecosystem, define an extra .NET library, that wraps the functional library, instead of refactoring

.NET-applications and libraries

- Creating libs and apps with .NET Core 2.x:
  - libs:
    ```
    > dotnet new classlib -lang F# -o Library
    > cd Library
    > dotnet restore
    > dotnet build -c Release
    ```
  - apps:
    ```
    > dotnet new console -lang F# -o Console
    > cd Console
    > dotnet restore
    > dotnet build -c Release
    ```

.NET-applications and libraries

- **Tasks 02**:
  - Place the created domain under a module called **Book** and place it in your **Library project**
  - Add to your **Console.fsproj** the following to point to your **Library project**:

```
<ItemGroup>
  <Reference Include="Library">
    <HintPath>..\Library\bin\Release\netstandard2.0\Library.dll</HintPath>
  </Reference>
</ItemGroup>
```

  - Create an instance of a book in your app, suggestion, and just print it out to standard console output

    Note: To print any complex types, just use: `printfn "%A"`

  - Execute the application to see the output: `> dotnet run`

Note: In script files, you can just do any of these two approaches, to access logic from libs:

```
#load "Library.fs"                          (* source   files *)
#r "bin/Release/netstandard2.0/Library.dll" (* assembly files *)
```

Overview

- **`17:00`** |> A few basic concepts to get started
- **`17:15`** |> Type (Domain) Driven Development
- **`18:00`** |> .NET-applications and libraries
- **`18:45`** |> **Data and ~~TypeProviders~~**
- **`19:30`** |> Concurrency and parallelism
- **`20:15`** |> Robust and error-free applications
- **`21:00`** |> Summary

Data and ~~TypeProviders~~

- There are a few built-in collections (datastructures) that you will be using again and again with:
  - The following high-order functions:
    - `map    : (('a -> 'b) -> 'a collection -> 'b collection)`
    - `fold   : (('a -> 'b -> 'a) -> 'a -> 'b collection -> 'a)`
    - `reduce : (('a -> 'a -> 'a) -> 'a collection -> 'a)`
    - `zip    : ('a collection -> 'b collection -> ('a * 'b) collection)`
  - Collections:
    - List: Ideal for constant additions and linear reads. **Note:** prepend (::) vs concatenate (@)
    - Array: (vector) and multi-dimensional arrays (matrices). Constant reads and updates (mutable)
    - Sequences: (lazy evaluation). Ideal for creating infinite sequences yielding new values
    - Map: Ideal for storing unique keys and its corresponding value. Replacement for Dictionaries (slower)
    - Set: Ideal for storing unique values

Data and ~~TypeProviders~~

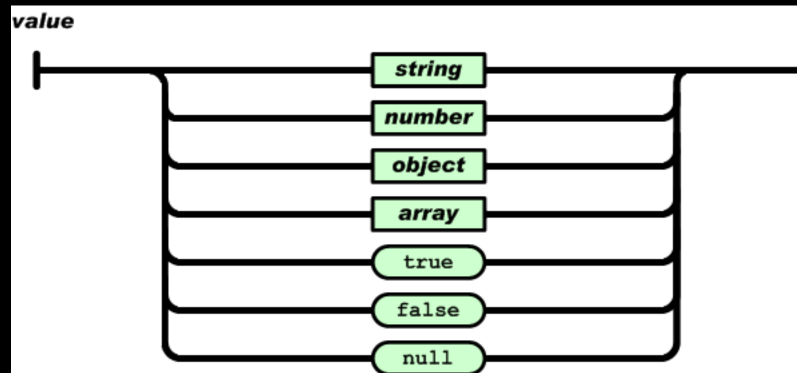
- But it's very easy to implement well known datastructures like trees or lazy lists:

```
type tree<'a> =
  | Leaf | Branch of tree<'a> * 'a * tree<'a>
type lazylist<'a> =
  | Nil  | Cons of 'a * lazylist<unit -> 'a>
```

  - ML compatibility versions:

```
type 'a tree =
  | Leaf | Branch of 'a tree * 'a * 'a tree
type 'a lazylist =
  | Nil  | Cons of 'a * (unit -> 'a lazylist)
```

Data and ~~TypeProviders~~



- **Tasks 03:**
  - a) Implement the infinite sequence: $\{0\} \cup \mathbb{N}$
  - b) Implement the JSON value as a F# (recursive) datastructure
    - Overload `ToString()` to print out a JSON string

```
type Foo = Foo with override x.ToString() = "Foo"
```

Overview

- **`17:00`** |> A few basic concepts to get started
- **`17:15`** |> Type (Domain) Driven Development
- **`18:00`** |> .NET-applications and libraries
- **`18:45`** |> Data and ~~TypeProviders~~
- **`19:30`** |> **Concurrency and parallelism**
- **`20:15`** |> Robust and error-free applications
- **`21:00`** |> Summary

Concurrency and parallelism

- In some cases, map/reduce pattern, it's easy to go from sequential calculations to true parallelism with a bit of code refactoring:

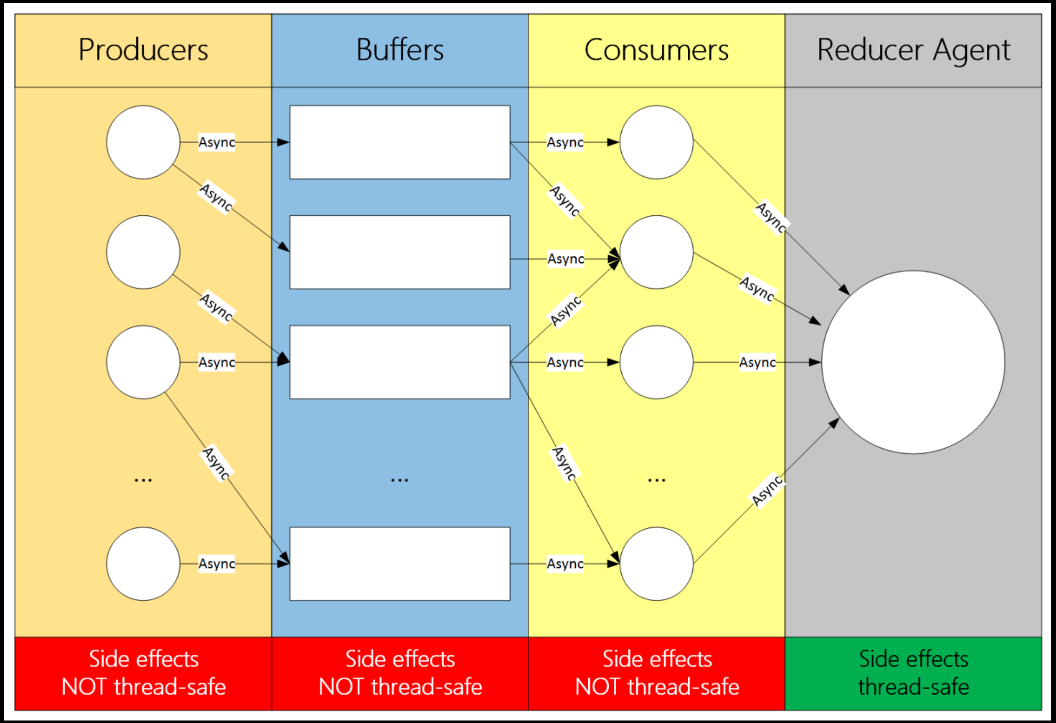
```
[| 0 .. 10 .. (1 <<< 16) |]
|> Array.map (fun x -> x * x)

[| 0 .. 10 .. (1 <<< 16) |]
|> Array.Parallel.map (fun x -> x * x)
```

- When calculations not only depend on the CPU, as the example showed above, there is also support for concurrent non-blocking asynchronous processes for when I/O (storage, network, ...) are involved in the computation:

```
[| "https://duckduckgo.com/"; "https://google.com"; "https://bing.com" |]
|> Array.Parallel.map asyncHttp (* Create async load, no I/O involved  *)
|> Async.Parallel               (* Retrieve sites concurrently        *)
|> Async.RunSynchronously       (* Wait for all processes to terminate *)
```

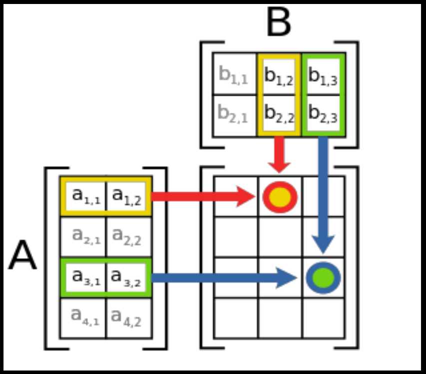
Note: Think of `Async.Parallel` and `Async.RunSynchronously` as `fork` and `join`

Concurrency and parallelism



Built-in actor model for concurrent async flows

Concurrency and parallelism



- **Tasks 04a**:
  - Implement Parallel versions of Matrix:
    - Addition:
      - https://en.wikipedia.org/wiki/Matrix_addition
    - Multiplication:
      - https://en.wikipedia.org/wiki/Matrix_multiplication

Note: Use `#time` in your F# scripts to measure execution time

Concurrency and parallelism

- **Tasks 04b**:
  - Retrieve the following wikipedia article:
    - List of programming languages
  - Crawl all the url to each of the programming languages and retrieve those articles
  - Decide if the article have an "Hello World" example

  Note: For this task, we will be using both `syncHttp` and `asyncHttp` from Don Symes blog post:
    - Introducing F# Asynchronous Workflows

Overview

- **17:00** |> A few basic concepts to get started
- **17:15** |> Type (Domain) Driven Development
- **18:00** |> .NET-applications and libraries
- **18:45** |> Data and ~~TypeProviders~~
- **19:30** |> Concurrency and parallelism
- **20:15** |> **Robust and error-free applications**
- **21:00** |> Summary

Robust and error-free applications

- In order to make applications more robust, so they don't break unexpectedly, you should be using types to wrap in calculated values. A know approach is to be using the Maybe Monad, called Option in F#, where you have **Some**thing of the assigned type or Nothing (**None**). Here is an example that solves the problem of **dividing by zero**:

```
let (>>=) m f = Option.bind f m (* define operator for easy writing      *)

let inc     = (+) 1                                 >> Some
let dob     = (*) 2                                 >> Some
let div d n = if d = 0 then None else n / d |> Some (* Avoid div by zero *)

42 |> inc >>= div 0 >>= dob (* > val it : int option = None              *)
42 |> inc >>= div 1 >>= dob (* > val it : int option = Some 86           *)
```

Note: Notice how both code branches are equal? This is ideal to write code flows of logic

Robust and error-free applications

- When using the Maybe Monad, you don't really see what is wrong in your computation. For that reason, there is another monad called the **Either Monad**, very similar to Maybe, which allows you to add a context of failure. This monad in F# is called Choice or very recently **Result** (F# 4.1). Here is the updated example from previous slide:

```
let bind     f = function | Choice1Of2 r -> f r | Choice2Of2 l -> Choice2Of2 l
let (>>=) m f = bind f m (* define operator for easy writing          *)

let inc      = (+) 1                                              >> Choice1Of2
let dob      = (*) 2                                              >> Choice1Of2
let div d n = if d = 0 then Choice2Of2 "Div by zero" else n / d |> Choice1Of2

42 |> inc >>= div 0 >>= dob (* > val it ... = Choice2Of2 "Div by zero" *)
42 |> inc >>= div 1 >>= dob (* > val it ... = Choice1Of2 86           *)
```

Note: Notice how both code branches are still equal but now we have a bit more information

Robust and error-free applications

- **Tasks 05**:
  - Implement some logic in order to rate a movie with 0-5 stars
  - Build into the logic so that fake reviews will not be accepted
  - Calculate the average of a given movie review
    - Use random values to create 25 reviews (-10 lower and 10 upper)
  - Use the minimal amount of memory to store movie reviews

Note: Greater than 5 or less than 0 are fake reviews

Overview

- `17:00` |> A few basic concepts to get started
- `17:15` |> Type (Domain) Driven Development
- `18:00` |> .NET-applications and libraries
- `18:45` |> Data and ~~TypeProviders~~
- `19:30` |> Concurrency and parallelism
- `20:15` |> Robust and error-free applications
- `21:00` |> **Summary**

Summary

- **αΩ** for most functional programming languages is **composition**
- Use **modules + ADT + curried functions** rather than type classes as libraries and methods
- Use Type (Domain) Driven Development (**T/DDD**) to model your business logic
  - Use **module encapsulation** for **Making Ilegal States Unrepresentable** (MISU)
- **Don't expose mutability**. The language is **functional first**, therefore people using your logic will expect you to follow this approach
- **Concurrency and parallelism isn't that hard** when you use the right tools …
  - "If all you have is a hammer, everything looks like a nail"
- **Correctness $\gg$ performance**

  Note: The notation $\gg$, reads much greater than

Summary

Correctness ≫ performance

There is a reason we don't fly with these anymore …