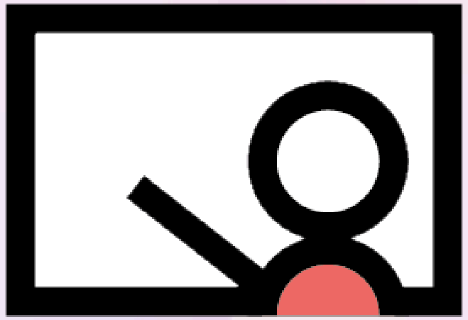
F♯ Training

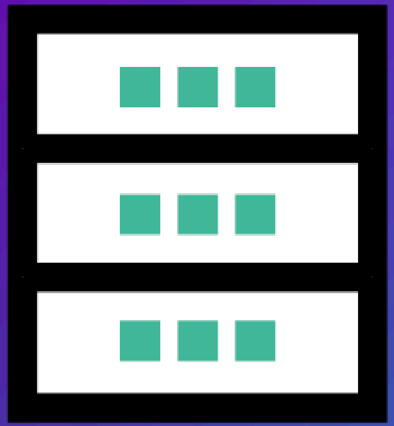
Monadic Types

2025 April

Table of contents



1. Type `Option`



Type `Option`

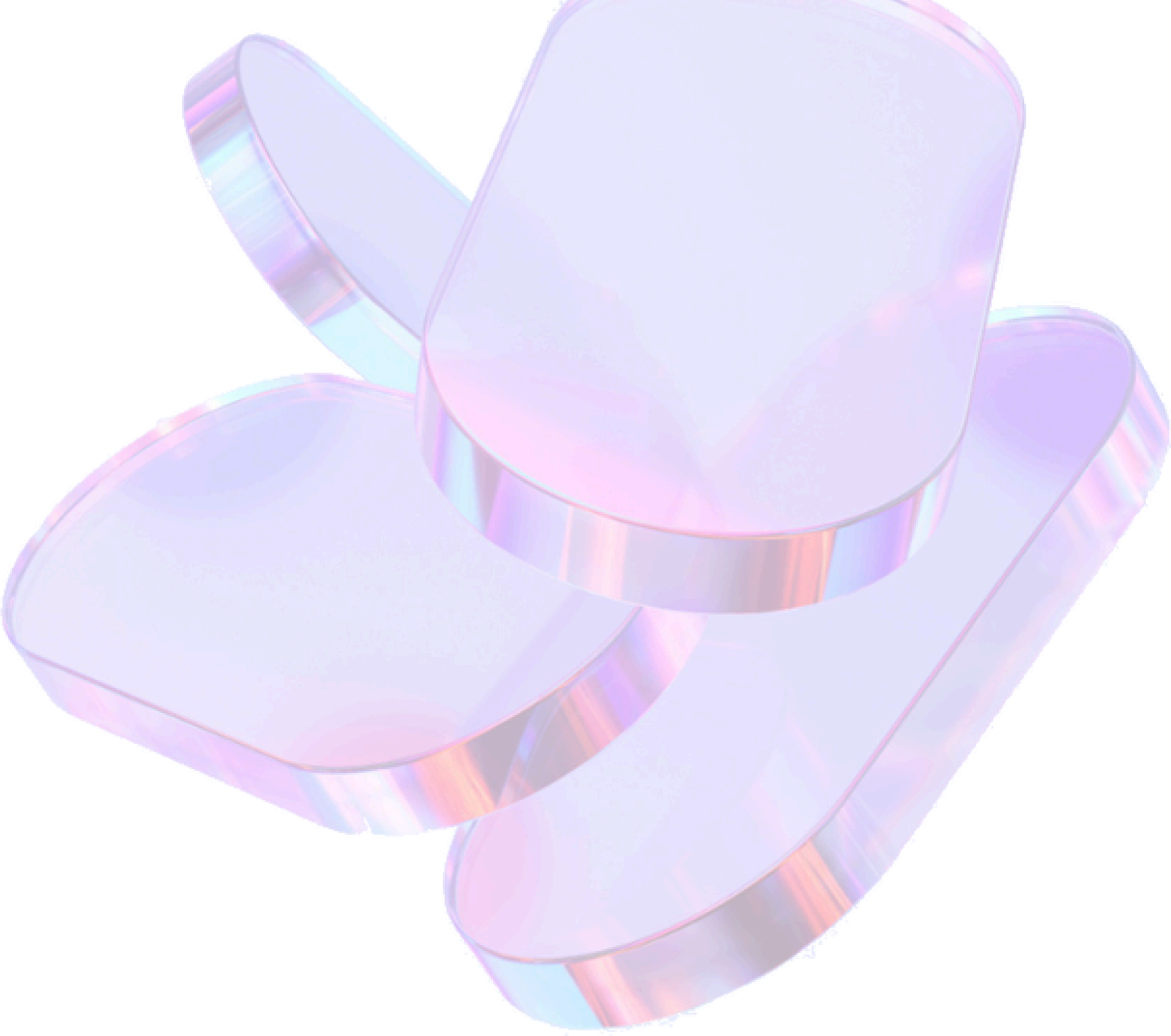
A.k.a `Maybe` *(Haskell)*, `Optional` *(Java 8)*

Models the absence of value
→ Defined as a union with 2 *cases*

```
type Option<'Value> =
    | None              // Case without data → when value is missing
    | Some of 'Value    // Case with data → when value is present
```

`Option` » Use cases

1. Modeling an optional field
2. Partial operation



Case 1: Modeling an optional field

```fsharp
type Civility = Mr | Mrs

type User = { Name: string; Civility: Civility option } with
    static member Create(name, ?civility) = { Name = name; Civility = civility }

let joey  = User.Create("Joey", Mr)
let guest = User.Create("Guest")
```

→ Make it explicit that `Name` is mandatory and `Civility` optional

☝️ **Warning:** this design does not prevent `Name = null` here *(BCL limit)*

Case 2. Partial operation

Operation where no output value is possible for certain inputs.

Example 1: inverse of a number

```
let inverse n = 1.0 / n

let tryInverse n =
    match n with
    | 0.0 -> None
    | n   -> Some (1.0 / n)
```

Function	Operation	Signature	`n = 0.5`	`n = 0.0`
`inverse`	Partial	`float -> float`	`2.0`	`infinity` ❓
`tryInverse`	Total	`float -> float option`	`Some 2.0`	`None` 👌

Case 2. Partial operation (2)

Example 2: find an element in a collection

- ➔ Partial operation: `find predicate` → 💥 when item not found
- ➔ Total operation: `tryFind predicate` → `None` or `Some item`

Benefits 👍

- ➔ Explicit, honest / partial operation
  - ➔ No special value: `null`, `infinity`
  - ➔ No exception
- ➔ Forces calling code to handle all cases:
  - ➔ `Some value` → output value given
  - ➔ `None .....` → output value missing

`Option` » Control flow

To test for the presence of the value *(of type* `'T`*)* in the option

- ➔ ❌ Do not use `IsSome`, `IsNone` and `Value` (🤞💥)
  - ➔ ~~if option.IsSome then option.Value...~~
- ➔ 👌 By hand with *pattern matching*.
- ➔ ✅ `Option.xxx` functions 📍

Manual control flow with *pattern matching*

Example:

```fsharp
let print option =
    match option with
    | Some x -> printfn "%A" x
    | None   -> printfn "None"

print (Some 1.0)  // 1.0
print None        // None
```

Control flow with `Option.xxx` helpers

Mapping of the inner value (of type `'T`) **if present**:
→ `map f option` with `f` total operation `'T → 'U`
→ `bind f option` with `f` partial operation `'T → 'U option`

Keep value **if present** and if conditions are met:
→ `filter predicate option` with `predicate: 'T → bool` called only if value present

🧑‍🏫 **Demo**
→ Implementation of `map`, `bind` and `filter` with *pattern matching*

🧑‍🏫 Demo » Solution

```fsharp
let map f option =              // (f: 'T → 'U) → 'T option → 'U option
    match option with
    | Some x → Some (f x)
    | None   → None             // 🎁 1. Why can't we write `None → option`?

let bind f option =             // (f: 'T → 'U option) → 'T option → 'U option
    match option with
    | Some x → f x
    | None   → None

let filter predicate option =  // (predicate: 'T → bool) → 'T option → 'T option
    match option with
    | Some x when predicate x → option
    | _ → None                  // 🎁 2. Implement `filter` with `bind`?
```

🎁 Bonus questions » Answers

```fsharp
// 🎁 1. Why can't we write `None → option`?
let map (f: 'T → 'U) (option: 'T option) : 'U option =
    match option with
    | Some x → Some (f x)
    | None   → (*None*) option  // 💥 Type error: `'U option` given ≠ `'T option` expected
```

```fsharp
// 🎁 2. Implement `filter` with `bind`?
let filter predicate option =  // (predicate: 'T → bool) → 'T option → 'T option
    option ▷ bind (fun x → if predicate x then option else None)
```

Integrated control flow » Example

```fsharp
// Question/answer console application
type Answer = A | B | C | D

let tryParseAnswer =
    function
    | "A" → Some A
    | "B" → Some B
    | "C" → Some C
    | "D" → Some D
    | _   → None

/// Called when the user types the answer on the keyboard
let checkAnswer (expectedAnswer: Answer) (givenAnswer: string) =
    tryParseAnswer givenAnswer
    ▷ Option.filter ((=) expectedAnswer)
    ▷ Option.map (fun _ → "✅")
    ▷ Option.defaultValue "❌"

["X"; "A"; "B"] ▷ List.map (checkAnswer B)  // ["❌"; "❌"; "✅"]
```

Integrated control flow » Advantages

Makes business logic more readable

- ➔ No `if hasValue then / else`
- ➔ Highlight the *happy path*
- ➔ Handle corner cases at the end

💡 The *computation expressions* 📍 provide an alternative syntax + lightweight

`Option`: comparison with other types

1. `Option` *vs* `List`
2. `Option` *vs* `Nullable`
3. `Option` *vs* `null`

`Option` *vs* `List`

Conceptually closed
→ Option ≃ List of 0 or 1 items
→ See `Option.toList` function: `'t option → 't list` (`None → []`, `Some x → [x]`)

💡 `Option` & `List` modules: many functions with the same name
→ `contains`, `count`, `exist`, `filter`, `fold`, `forall`, `map`

☝ A `List` can have more than 1 element
→ Type `Option` models absence of value better than type `List`

`Option` *vs* `Nullable`

`System.Nullable<'T>` ≃ `Option<'T>` but more limited

- ➔ ❗ Does not work for reference types
- ➔ ❗ Lacks monadic behavior i.e. `map` and `bind` functions
- ➔ ❗ Lacks built-in pattern matching `Some x | None`
- ➔ ❗ In F♯, no magic as in C♯ / keyword `null`

👉 C♯ uses nullable types whereas F♯ uses only `Option`

`Option` *vs* `null`

Due to the interop with the BCL, F♯ has to deal with `null` in some cases.

👉 **Good practice**: isolate these cases and wrap them in an `Option` type.

```fsharp
let readLine (reader: System.IO.TextReader) =
    reader.ReadLine() // Can return `null`
    ▷ Option.ofObj    // `null` becomes None

    // Same than:
    match reader.ReadLine() with
    | null → None
    | line → Some line
```

2. Type `Result`

Type `Result`

A.k.a `Either` *(Haskell)*

Models a *double-track* Success/Failure

```
type Result<'Success, 'Error> = // 2 generic parameters
    | Ok of 'Success   // Success Track
    | Error of 'Error // Failure Track
```

Functional way of dealing with business errors *(expected errors)*
→ Allows exceptions to be used only for exceptional errors
→ As soon as an operation fails, the remaining operations are not launched

🔗 *Railway-oriented programming (ROP)*
https://fsharpforfunandprofit.com/rop/

Module `Result`

Contains less functions than `Option` ⁉️

`map f result` : to map the success

- `('T → 'U) → Result<'T, 'Error> → Result<'U, 'Error>`

`mapError f result` : to map the error

- `('Err1 → 'Err2) → Result<'T, 'Err1> → Result<'T, 'Err2>`

`bind f result` : same as `map` with `f` returning a `Result`

- `('T → Result<'U, 'Error>) → Result<'T, 'Error> → Result<'U, 'Error>`
- 💡 The result is flattened, like the `flatMap` function on JS arrays
- ⚠️ Same type of `'Error` for `f` and the input `result`.

Quiz *Result* 🕹️

Implement `Result.map` and `Result.bind`

💡 **Tips:**

- ➔ *Map* the *Success* track
- ➔ Access the *Success* value using pattern matching

Quiz *Result* 🎲

Solution: implementation of `Result.map` and `Result.bind`

```
// ('T → 'U) → Result<'T, 'Error> → Result<'U, 'Error>
let map f result =
    match result with
    | Ok x    → Ok (f x)  // ☝️ Ok → Ok
    | Error e → Error e   // ⚠️ The 2 `Error e` don't have the same type!

// ('T → Result<'U, 'Error>) → Result<'T, 'Error>
//                           → Result<'U, 'Error>
let bind f result =
    match result with
    | Ok x    → f x       // ☝️ `f x` already returns a `Result`
    | Error e → Error e
```

`Result` : Success/Failure tracks

`map` : no track change

```
Track        Input              Operation       Output
Success ─ Ok x      ───► map( x → y ) ───► Ok y
Failure ─ Error e ───► map(  ....   ) ───► Error e
```

`bind` : eventual routing to Failure track, but never vice versa

```
Track       Input                  Operation             Output
Success ─ Ok x      ─┬─► bind( x → Ok y      ) ───► Ok y
                     └─► bind( x → Error e2 ) ─┐
Failure ─ Error e ────► bind(      ....      ) ─┴─► Error ~
```

☝️ The *mapping/binding* operation is never executed in track Failure.

`Result` *vs* `Option`

`Option` can represent the result of an operation that may fail
☝️ But if it fails, the option doesn't contain the error, just `None`

`Option<'T>` ≃ `Result<'T, unit>`
→ `Some x` ≃ `Ok x`
→ `None` ≃ `Error ()`
→ See `Result.toOption` *(built-in)* and `Result.ofOption` *(below)*

```
[<RequireQualifiedAccess>]
module Result =
    let ofOption error option =
        match option with
        | Some x -> Ok x
        | None -> Error error
```

`Result` vs `Option` (2)

📅 **Dates:**

- The `Option` type is part of F# from the get go
- The `Result` type is more recent: introduced in F# 4.1 (2016)
  → After numerous articles on *F# for fun and profit*

📝 **Memory:**

- The `Option` type (alias: `option`) is a regular union: a reference type
- The `Result` type is a *struct* union: a value type
- The `ValueOption` type (alias: `voption`) is a *struct* union
  → `ValueNone | ValueSome of 't`

`Result` *vs* `Option` » Example

Let's change our previous `checkAnswer` to indicate the `Error`:

```fsharp
type Answer = A | B | C | D
type Error = InvalidInput of string | WrongAnswer of Answer

let tryParseAnswer =
    function
    | "A" -> Ok A
    | "B" -> Ok B
    | "C" -> Ok C
    | "D" -> Ok D
    | s   -> Error(InvalidInput s)

let checkAnswerIs expected actual =
    if actual = expected then Ok actual else Error(WrongAnswer actual)

// ...
```

`Result` vs `Option` » Example (2)

```fsharp
// ...

let printAnswerCheck (givenAnswer: string) =
    tryParseAnswer givenAnswer
    |> Result.bind (checkAnswerIs B)
    |> function
       | Ok x                  -> printfn $"%A{x}: ✅ Correct"
       | Error(WrongAnswer x)  -> printfn $"%A{x}: ❌ Wrong Answer"
       | Error(InvalidInput s) -> printfn $"%s{s}: ❌ Invalid Input"

printAnswerCheck "X";;  // X: ❌ Invalid Input
printAnswerCheck "A";;  // A: ❌ Wrong Answer
printAnswerCheck "B";;  // B: ✅ Correct
```

3. *Smart constructor*

Smart constructor: Purpose

> Making illegal states unrepresentable

🔗 https://kutt.it/MksmkG *F♯ for fun and profit, Jan 2013*

- ➔ Design to prevent invalid states
  - ➔ Encapsulate state *(all primitives)* in an object
- ➔ *Smart constructor* guarantees a valid initial state
  - ➔ Validates input data
  - ➔ If Ko, returns "nothing" (`Option`) or an error (`Result`)
  - ➔ If Ok, returns the created object wrapped in an `Option` / a `Result`

Encapsulate the state in a type

→ *Single-case (discriminated) union* 👌 : `Type X = private X of a: 'a ...`
🔗 https://kutt.it/mmMXCo *F♯ for fun and profit, Jan 2013*

→ *Record* 👍 : `Type X = private { a: 'a ... }`
🔗 https://kutt.it/cYP4gY *Paul Blasucci, Mai 2021*

☝️ `private` keyword:
→ Hide object content
→ Fields and constructor no longer visible from outside
→ Smart constructor defined in companion module or static method

Smart constructor » Example #1

Smart constructor :
→ `tryCreate` function in companion module
→ Returns an `Option`

```fsharp
type Latitude = private { Latitude: float } // 👈 A single field, named like the

[<RequireQualifiedAccess>]                  // 👈 Optional
module Latitude =
    let tryCreate (latitude: float) =
        if latitude >= -90. && latitude <= 90. then
            Some { Latitude = latitude }    // 👈 Constructor accessible here
        else
            None

let lat_ok = Latitude.tryCreate 45.  // Some { Latitude = 45.0 }
let lat_ko = Latitude.tryCreate 115. // None
```

Smart constructor » Example #2

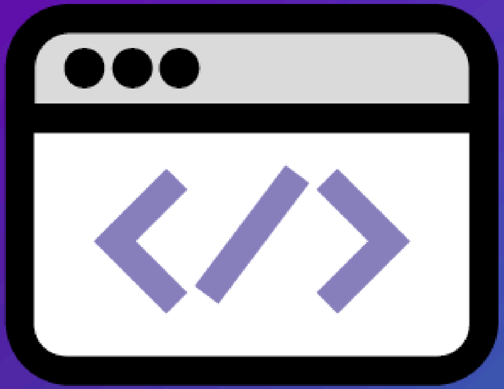
Smart constructor:
→ Static method `Of`
→ Returns `Result` with error of type `string`

```fsharp
type Tweet =
    private { Tweet: string }

    static member Of tweet =
        if System.String.IsNullOrEmpty tweet then
            Error "Tweet shouldn't be empty"
        elif tweet.Length > 280 then
            Error "Tweet shouldn't contain more than 280 characters"
        else Ok { Tweet = tweet }

let tweet1 = Tweet.Of "Hello world" // Ok { Tweet = "Hello world" }
```

4. Computation expression 🚀



Computation expression (CE)

Syntactic sugar hiding a "machinery"
→ Applies the *Separation of Concerns* principle
→ Code should be more readable inside the *computation expression*

Syntax: `builder { expr }`
→ `builder` instance of a "Builder" 📍
→ `expr` can contain `let`, `let!`, `do!`, `yield`, `yield!`, `return`, `return!`

💡 **Note:** `seq`, `async` and `task` are CEs

Builder

A *computation expression* relies on an object called *Builder*.
→ This object can be used to store a background state.

For each supported keyword (`let!`, `return`...), the *Builder* implements one or more related methods. Examples:

- `builder { return expr }` → `builder.Return(expr)`
- `builder { let! x = expr; cexpr }` → `builder.Bind(expr, (fun x -> {| cexpr |}))`

The *builder* can also wrap the result in a type of its own:

- `async { return x }` returns an `Async<'X>` type
- `seq { yield x }` returns a type `Seq<'X>`

Builder desugaring

The compiler translates to the *builder* methods.
→ The CE hides the complexity of these calls, which are often nested:

```
seq {
    for n in list do
        yield n
        yield n * 10 }

// Desugared as:
seq.For(list, fun () ->
    seq.Combine(seq.Yield(n),
                seq.Delay(fun () -> seq.Yield(n * 10)) ) )
```

Builder - Example : `logger`

Need: log the intermediate values of a calculation

```fsharp
let log value = printfn $"{value}"

let loggedCalc =
    let x = 42
    log x   // ❶
    let y = 43
    log y   // ❶
    let z = x + y
    log z   // ❶
    z
```

Problems ⚠️

① Verbose: the `log x` interfere with reading

② *Error prone*: forget a `log`, log wrong value…

Builder - Example : `logger` (2)

💡 Make logs implicit in a CE when `let!` / `Bind` :

```fsharp
type LoggingBuilder() =
    let log value = printfn $"{value}"; value
    member _.Bind(x, f) = x ▷ log ▷ f
    member _.Return(x) = x

let logger = LoggingBuilder()

//---

let loggedCalc = logger {
    let! x = 42
    let! y = 43
    let! z = x + y
    return z
}
```

Builder - Example : `maybe`

Need: simplify the sequence of "trySomething" returning an `Option`

```fsharp
let tryDivideBy bottom top = // (bottom: int) -> (top: int) -> int option
    if (bottom = 0) or (top % bottom <> 0)
    then None
    else Some (top / bottom)

// W/o CE
let division =
    36
    |> tryDivideBy 2                 // Some 18
    |> Option.bind (tryDivideBy 3)   // Some 6
    |> Option.bind (tryDivideBy 2)   // Some 3
```

Builder - Example : `maybe` (2)

```fsharp
// With CE
type MaybeBuilder() =
    member _.Bind(x, f) = x ▷ Option.bind f
    member _.Return(x) = Some x

let maybe = MaybeBuilder()

let division' = maybe {
    let! v1 = 36 ▷ tryDivideBy 2
    let! v2 = v1 ▷ tryDivideBy 3
    let! v3 = v2 ▷ tryDivideBy 2
    return v3
}
```

Result: ✅ Symmetry, ❌ Intermediate values

Limit : nested CEs

✅ Different CEs can be nested
❌ But code becomes difficult to understand

Example: combining `logger` and `maybe` ❓

Alternative solution 🚀🚀:

```fsharp
// Define an operator for `bind`
let inline (>>=) x f = x |> Option.bind f

let logM value = printfn $"{value}"; Some value  // 'a -> 'a option

let division' =
    36 |> tryDivideBy 2 >>= logM
       >>= tryDivideBy 3 >>= logM
       >>= tryDivideBy 2 >>= logM
```

Limit: combining CEs

How to combine `Async` + `Option`/`Result` ?
→ `asyncResult` CE + helpers in [FsToolkit]

```fsharp
type LoginError =
    | InvalidUser | InvalidPassword
    | Unauthorized of AuthError | TokenErr of TokenError

let login username password =
    asyncResult {
        // tryGetUser: string -> Async<User option>
        let! user = username |> tryGetUser |> AsyncResult.requireSome InvalidUser
        // isPasswordValid: string -> User -> bool
        do! user |> isPasswordValid password |> Result.requireTrue InvalidPassword
        // authorize: User -> Async<Result<unit, AuthError>>
        do! user |> authorize |> AsyncResult.mapError Unauthorized
        // createAuthToken: User -> Result<AuthToken, TokenError>
        return! user |> createAuthToken |> Result.mapError TokenErr
    } // Async<Result<AuthToken, LoginError>>
```

CE: the Swiss army knife ✨

The *computation expressions* serve different purposes:

- C♯ `yield return` → F♯ `seq {}`
- C♯ `async/await` → F♯ `async {}`
- C♯ LINQ expressions `from … select` → F♯ `query {}`
- …

Underlying theoretical foundations :

- Monoid
- Monad
- Applicative

Monoid

≃ Type `T` defining a set with:

1. Operation `(+): T → T → T`
   → To combine sets and keep the same "type"
   → Associative: `a + (b + c)` ≡ `(a + b) + c`
2. Neutral element *(aka identity)* ≃ empty set
   → Combinable with any set without effect
   → `a + e` ≡ `e + a` ≡ `a`

CE monoidal

The builder of a monoidal CE *(such as `seq`)* has *at least* :

- ➔ `Yield` to build the set element by element
- ➔ `Combine` ≡ `(+)` (`Seq.append`)
- ➔ Zero `≡ neutral element (` Seq.empty`)

Generally added (among others):

- ➔ `For` to support `for x in xs do ...`
- ➔ `YieldFrom` to support `yield!`

Monad

≃ Generic type `M<'T>` with:

1. `return` construction function
   - ➔ Signature : `(value: 'T) -> M<'T>`
   - ➔ ≃ Wrap a value
2. Link function `bind` *(aka* `>>=` *operator)*
   - ➔ Signature : `(f: 'T -> M<'U>) -> M<'T> -> M<'U>`
   - ➔ Use wrapped value, map with `f` function
     to a value of another type and re-wrap the result

Monad: laws

`return` ≡ neutral element for `bind`

- ➔ Left: `return x ▷ bind f` ≡ `f x`
- ➔ Right: `m ▷ bind return` ≡ `m`

`bind` is associative

- ➔ `m ▷ bind f ▷ bind g` ≡ `m ▷ bind (fun x → f x ▷ bind g)`

Monads and languages

Haskell

- Monads used a lot. Common ones: `IO`, `Maybe`, `State`, `Reader`.
- `Monad` is a *type class* for easily creating your own monads.

F♯

- Some CEs allow monadic operations.
- More rarely used directly *(except by Haskellers, OCamlers…)*

C♯

- Monad implicit in LINQ
- LanguageExt library for functional programming

Monadic CE

The builder of a monadic CE has `Return` and `Bind` methods.

The `Option` and `Result` types are monadic.
→ We can create their own CE :

```
type OptionBuilder() =
    member _.Bind(x, f) = x ▷ Option.bind f
    member _.Return(x) = Some x

type ResultBuilder() =
    member _.Bind(x, f) = x ▷ Result.bind f
    member _.Return(x) = Ok x
```

Monadic and generic CE

FSharpPlus provides a `monad` CE
→ Works for all monadic types: `Option`, `Result`, ... and even `Lazy`!

```
#r "nuget: FSharpPlus"
open FSharpPlus

let lazyValue = monad {
    let! a = lazy (printfn "I'm lazy"; 2)
    let! b = lazy (printfn "I'm lazy too"; 10)
    return a + b
} // System.Lazy<int>

let result = lazyValue.Value
// I'm lazy
// I'm lazy too
// val result : int = 12
```

Monadic and generic CE (2)

Example with `Option` type:

```
#r "nuget: FSharpPlus"
open FSharpPlus

let addOptions x' y' = monad {
    let! x = x'
    let! y = y'
    return x + y
}

let v1 = addOptions (Some 1) (Some 2) // Some 3
let v2 = addOptions (Some 1) None     // None
```

Monadic and generic CE (3)

⚠️ **Limit:** several monadic types cannot be mixed!

```fsharp
#r "nuget: FSharpPlus"
open FSharpPlus

let v1 = monad {
    let! a = Ok 2
    let! b = Some 10
    return a + b
} // 💥 Error FS0043 ...

let v2 = monad {
    let! a = Ok 2
    let! b = Some 10 |> Option.toResult
    return a + b
} // val v2 : Result<int,unit> = Ok 12
```

Specific monadic CE

FsToolkit.ErrorHandling library provides:

- CE `option {}` specific to type `Option<'T>` *(example below)*
- CE `result {}` specific to type `Result<'Ok, 'Err>`

☝️ Recommended as it is more explicit than `monad` CE.

```fsharp
#r "nuget: FSToolkit.ErrorHandling"
open FsToolkit.ErrorHandling

let addOptions x' y' = option {
    let! x = x'
    let! y = y'
    return x + y
}

let v1 = addOptions (Some 1) (Some 2) // Some 3
let v2 = addOptions (Some 1) None     // None
```

Applicative *(a.k.a Applicative Functor)*

≃ Generic type `M<'T>` -- 3 styles:

Style A: Applicative with `apply`/`<*>` and `pure`/`return`

- ❌ Not easy to understand
- ☝️ Not recommended by Don Syme in the [Nov. 2020 note]()

Style B: Applications with `mapN`

- `map2`, `map3`… `map5` combines 2 to 5 wrapped values

Style C: Applicatives with `let! ... and! ...` in a CE

- Same principle: combine several wrapped values
- Available from F♯ 5 (*[announcement Nov. 2020]()*)

☝️ **Tip:** Styles B and C are equally recommended.

Applicative CE

Library [FsToolkit.ErrorHandling] offers:

- Type `Validation<'Ok, 'Err>` ≡ `Result<'Ok, 'Err list>`
- CE `validation {}` supporting `let! … and! …` syntax.

Allows errors to be accumulated → Uses:

- Parsing external inputs
- *Smart constructor (Example code slide next…)*

Applicative CE: example

```
#r "nuget: FSToolkit.ErrorHandling"
open FsToolkit.ErrorHandling

type [<Measure>] cm
type Customer = { Name: string; Height: int<cm> }

let validateHeight height =
    if height ⩽ 0<cm>
    then Error "Height must me positive"
    else Ok height

let validateName name =
    if System.String.IsNullOrWhiteSpace name
    then Error "Name can't be empty"
    else Ok name

module Customer =
    let tryCreate name height : Result<Customer, string list> =
        validation {
            let! validName = validateName name
            and! validHeight = validateHeight height
            return { Name = validName; Height = validHeight }
        }

let c1 = Customer.tryCreate "Bob" 180<cm>  // Ok { Name = "Bob"; Height = 180 }
let c2 = Customer.tryCreate "Bob" 0<cm> // Error ["Height must me positive"]
let c3 = Customer.tryCreate "" 0<cm>    // Error ["Name can't be empty"; "Height must me positive"]
```

Applicative *vs* Monad

The `Result` type is "monadic": on the 1st error, we "unplug".

There is another type called `Validation` that is "applicative": it allows to accumulate errors.

- ≃ `Result<'ok, 'error list>`\
- Handy for validating user input and reporting all errors

🔗 Ressources

- [FsToolkit.ErrorHandling]
- [Validation with F# 5 and FsToolkit]

Applicative *vs* Monad (2)

Example: `Validation.map2` to combine 2 results and get the list of their eventual errors.

```
module Validation =
    // f : 'T → 'U → Result<'V, 'Err list>
    // x': Result<'T, 'Err list>
    // y': Result<'U, 'Err list>
    //  → Result<'V, 'Err list>
    let map2 f x' y' =
        match x', y' with
        | Ok x, Ok y → f x y
        | Ok _, Error errors | Error errors, Ok _ → Error errors
        | Error errors1, Error errors2 → Error (errors1 @ errors2) // 👈 ②
```

Other CE

We've seen 2 libraries that extend F♯ and offer their CEs:

- ➔ FSharpPlus → `monad`
- ➔ FsToolkit.ErrorHandling → `option`, `result`, `validation`

Many libraries have their own DSL *(Domain Specific Language.)*
Some are based on CE:

- ➔ Expecto: Testing library (`test "..." {...}`)
- ➔ Farmer: Infra as code for Azure (`storageAccount {...}`)
- ➔ Saturn: Web framework on top of ASP.NET Core (`application {...}`)

5. Wrap up

Union types: `Option` and `Result`

- What they are used for:
  - Model absence of value and business errors
  - Partial operations made total `tryXxx`
    - *Smart constructor* `tryCreate`
- How to use them:
  - Chaining: `map`, `bind`, `filter` → *ROP*
  - Pattern matching
- Their benefits:
  - `null` free, `Exception` free → no guard clauses Cluttering the code
  - Makes business logic and *happy path* more readable

Computation expression (CE)

- Syntactic sugar: inner syntax standard or "banged" (`let!`)
- *Separation of Concerns*: business logic *vs* "machinery"
- Compiler is linked to *builder*
  - Object storing a state
  - Builds an output value of a specific type
- Can be nested but not easy to combine!
- Underlying theoretical concepts
  - Monoid → `seq` *(of composable elements and with a "zero")*
  - Monad → `async`, `option`, `result`
  - Applicative → `validation`/`Result<'T, 'Err list>`
- Libraries: FSharpPlus, FsToolkit, Expecto, Farmer, Saturn

🔗 Additional ressources

- Compositional IT *(Isaac Abraham)*
  - *Writing more succinct C# – in F#! (Part 2)* · 2020
- F# for Fun and Profit *(Scott Wlaschin)*
  - *The Option type* · 2012
  - *Making illegal states unrepresentable* · 2013
  - *The "Map and Bind and Apply, Oh my!" series* · 2015
  - *The "Computation Expressions" series* · 2013
- Extending F# through Computation Expressions
  - 📹 Video
  - 📜 Article
- Computation Expressions Workshop
- Applicatives IRL by Jeremie Chassaing

Thanks 🙏