



→ Digitalize society



Formation F# 5.0

Les types « monadiques »



Décembre 2021

SOAT.FR

About me



Romain DENEAU

- SOAT depuis 2009
- Senior Developer C# F# TypeScript
- Passionné de Craft
- Auteur sur le blog de SOAT



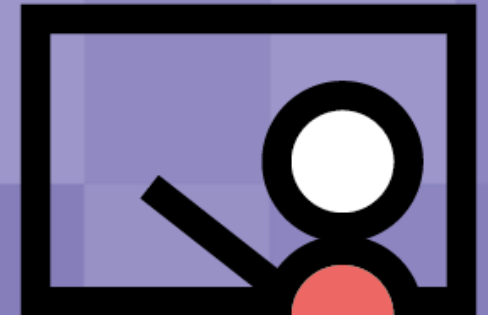
DeneauRomain



rdeneau

Sommaire

- Type `Option`
- Type `Result`
- *Smart constructor*
- *Computation expression*



1.

Type

Option



Type Option

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

Modélise l'absence de valeur

→ Défini sous la forme d'une union avec 2 *cases*

```
type Option<'Value> =  
    | None           // Case sans donnée → quand valeur absente  
    | Some of 'Value // Case avec donnée → quand valeur présente
```

F#

Option >> Cas d'utilisation

Cas 1. Modéliser un champ optionnel

```
type Civility = Mr | Mrs
type User = { Name: string; Civility: Civility option }

let joey = { Name = "Joey"; Civility = Some Mr }
let guest = { Name = "Guest"; Civility = None }
```

F#

→ Rend explicite le fait que `Name` est obligatoire et `Civility` facultatif

👉 **Attention** : ce design n'empêche pas ici d'avoir `Name = null` (*limite BCL*)

Cas 2. Opération partielle

Opération où aucune valeur de sortie n'est possible pour certaines entrées.

Exemple 1 : inverse d'un nombre

```
let inverse n = 1.0 / n

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

F#

Fonction	Opération	Signature	n = 0.5	n = 0.0
inverse	Partielle	float → float	2.0	infinity ?
tryInverse	Totale	float → float option	Some 2.0	None 🙌

Exemple 2 : recherche d'un élément dans une collection

- Opération partielle : `find predicate` → 💥 quand élément non trouvé
- Opération totale : `tryFind predicate` → `None` ou `Some item`

Avantages 👍

- Explicite, honnête / partialité de l'opération
 - Pas de valeur spéciale : `null`, `infinity`
 - Pas d'exception
- Force le code appelant à gérer la totalité des cas :
 - Présence d'une valeur en sortie : `Some value`
 - Absence d'une valeur en sortie : `None`

Option >> Flux de contrôle

Pour tester la présence de la valeur (de type `'T`) dans l'option

- ❌ Ne pas utiliser `IsSome`, `IsNone` et `Value` (👉💣)
- ~~if option.IsSome then option.Value...~~
- 👉 A la main avec *pattern matching*
- ✅ Fonctions du module `Option`

Flux de contrôle manuel avec *pattern matching*

Exemple :

```
let print option =  
    match option with  
    | Some x → printfn "%A" x  
    | None   → printfn "None"  
  
print (Some 1.0) // 1.0  
print None      // None
```

F#

Flux de contrôle intégré au module **Option**

Opération de *Mapping* de la valeur (de type **'T**) **si** **∃** :

- **map f option** avec **f** opération totale **'T → 'U**
- **bind f option** avec **f** opération partielle **'T → 'U option**

Conserver la valeur **si** **∃** et si respecte condition :

- **filter predicate option** avec **predicate: 'T → bool** appelé que si valeur **∃**



Démo

- Implémentation de **map**, **bind** et **filter** avec *pattern matching*



Démo » Solution


```
let map f option = // (f: 'T → 'U) → 'T option → 'U option
    match option with
    | Some x → Some (f x)
    | None   → None // 🎁 1. Pourquoi on ne peut pas écrire `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. Implémenter `filter` avec `bind` ?
```

F#

Questions bonus >> Réponses

```
//  1. Pourquoi on ne peut pas écrire `None → option` : F#  
let map (f: 'T → 'U) (option: 'T option) : 'U option =  
    match option with  
    | Some x → Some (f x)  
    | None   → (*None*) option // ✨ Erreur de typage : `'U option` attendu ≠ `'T option`
```

```
//  2. Implémenter `filter` avec `bind` : F#  
let filter predicate option = // (predicate: 'T → bool) → 'T option → 'T option  
    let f x = if predicate x then option else None  
    bind f option
```

Flux de contrôle intégré » Exemple

```
// Application console de questions/réponses
type Answer = A | B | C | D

let tryParseAnswer text =
    match text with
    | "A" → Some A
    | "B" → Some B
    | "C" → Some C
    | "D" → Some D
    | _   → None

// Fonction appelée quand l'utilisateur saisit la réponse au clavier à la question posée
let checkAnswer (expectedAnswer: Answer) (givenAnswer: string) =
    tryParseAnswer givenAnswer
    ▷ Option.filter ((=) expectedAnswer)
    ▷ Option.map (fun _ → "✓")
    ▷ Option.defaultValue "✗"

["✗"; "A"; "B"] ▷ List.map (checkAnswer B) // ["✗"; "✗"; "✓"]
```

F#

Flux de contrôle intégré » Bénéfices

Rend logique métier + lisible

- Pas de `if hasValue then / else`
- Met en valeur le *happy path*
- Centralise à la fin la gestion de l'absence de valeur

💡 Les *computation expressions* 📌 fournissent une syntaxe alternative + légère

Option VS List

Option \simeq Liste de 0 ou 1 élément

→ Cf. fonction `Option.toList`

```
let noneIsEmptyList      = Option.toList(None)    = []    // true
let someIsListWithOneItem = Option.toList(Some 1) = [1]    // true
```

F#

👉 Une `List` peut avoir + de 1 élément

→ Type `Option` modélise mieux l'absence de valeur que type `List`

💡 Module `Option` : beaucoup de même fonctions que module `List`

→ `contains`, `count`, `exist`, `filter`, `fold`, `forall`, `map`

Option vs Nullable

Type `System.Nullable<'T>` \simeq `Option<'T>` en + limité

- ! Ne marche pas pour les types références
- ! Manque comportement monadique i.e. fonctions `map` et `bind`
- ! En F#, pas de magie comme en C# / mot clé `null`

👉 `option` est le type idiomatique en F#

Option VS null

De part ses interactions avec la BCL, F# autorise parfois la valeur `null`

👉 **Bonne pratique** : isoler ces cas de figure et wrapper dans un type `Option`

```
let readLine (reader: System.IO.TextReader) =  
    reader.ReadLine() ▷ Option.ofObj
```

F#

```
// Équivalent à faire :  
match reader.ReadLine() with  
| null → None  
| line → Some line
```

2.

Type
Result



Type Result

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

Modélise une *double-track* Succès/Échec

```
type Result<'Success, 'Error> = // 2 paramètres génériques
| Ok of 'Success // Track "Succès"
| Error of 'Error // Track "Échec"
```

F#

Gestion fonctionnelle des erreurs métier (*les erreurs prévisibles*)

- Permet de limiter usage des exceptions aux erreurs exceptionnelles
- Dès qu'une opération échoue, les opérations restantes ne sont pas lancées
- *Railway-oriented programming* • <https://fsharpforfunandprofit.com/rop/>

Module **Result**

Ne contient que 3 fonctions

map f option : sert à mapper le résultat

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

mapError f option : sert à mapper l'erreur

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

bind f option : idem **map** avec fonction **f** qui renvoie un **Result**

- `('T → Result<'U, 'Error>) → Result<'T, 'Error> → Result<'U, 'Error>`
- 💡 Le résultat est aplati, comme la fonction **flatMap** sur les arrays JS
- ⚠️ Même type d'erreur **'Error** pour **f** et le **result** en entrée

Quiz *Result*

Implémenter `Result.map` et `Result.bind`

💡 **Tips :**

- *Mapping* sur la track *Succès*
- Accès à la valeur dans la track *Succès* :
 - Utiliser *pattern matching* (`match result with ...`)
- Retour : simple `Result`, pas un `Result<Result>` !



Quiz *Result*

Solution : implémentation de `Result.map` et `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  // ⚠️ Les 2 `Error e` n'ont pas le même type !

// ('T → Result<'U, 'Error>) → Result<'T, 'Error>
//                               → Result<'U, 'Error>
let bind f result =
    match result with
    | Ok x      → f x      // 🙌 Ok → Ok ou Error !
    | Error e   → Error e
```

F#

Result : tracks Success/Failure

map : pas de changement de track

Track	Input	Operation	Output
Success	Ok x	$\longrightarrow \text{map}(x \rightarrow y)$	$\longrightarrow \text{Ok } y$
Failure	Error e	$\longrightarrow \text{map}(\dots)$	$\longrightarrow \text{Error } e$

bind : routage possible vers track Failure mais jamais l'inverse

Track	Input	Operation	Output
Success	Ok x	$\begin{array}{l} \longrightarrow \text{bind}(x \rightarrow \text{Ok } y) \\ \quad \searrow \text{bind}(x \rightarrow \text{Error } e2) \end{array}$	$\longrightarrow \text{Ok } y$
Failure	Error e	$\longrightarrow \text{bind}(\dots)$	$\begin{array}{l} \searrow \\ \longrightarrow \text{Error } \sim \end{array}$

👉 Opération de *mapping/binding* jamais exécutée dans track Failure

Result vs Option

`Option` peut représenter le résultat d'une opération qui peut échouer
👉 Mais en cas d'échec, l'option ne contient pas l'erreur, juste `None`

`Option<'T> \simeq Result<'T, unit>`

→ `Some x \simeq Ok x`

→ `None \simeq Error ()`

→ Cf. fonctions `Option.toResult` et `Option.toResultWith error` de [FSharpPlus](https://fsprojects.github.io/FSharpPlus/)

```
let toResultWith (error: 'Error) (option: 'T option) : Result<'T, 'Error> =  
    match option with  
    | Some x → Ok x  
    | None   → Error error
```

F#

Result vs Option >> Example

Modification de la fonction `checkAnswer` précédente pour indiquer l'erreur :

```
open FSharpPlusF#

type Answer = A | B | C | D
type Error = InvalidInput | WrongAnswer

let tryParseAnswer text = ... // string → Answer option

let checkAnswer (expectedAnswer: Answer) (givenAnswer: string) =
    let check answer = if answer = expectedAnswer then Ok answer else Error WrongAnswer
    tryParseAnswer givenAnswer // Answer option
    ▷ Option.toResultWith InvalidInput // Result<Answer, Error>
    ▷ Result.bind check
    ▷ function
        | Ok _ → "✅"
        | Error InvalidInput → "❌ Invalid Input"
        | Error WrongAnswer → "❌ Wrong Answer"

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

Result vs Validation

`Result` est "monadique" : à la 1ère erreur, on "débranche"

`Validation` est "applicatif" : permet d'accumuler les erreurs

→ \simeq `Result<'ok, 'error list>`

→ Pratique pour valider saisie utilisateur et remonter Σ erreurs

→ Dispo dans librairies [FSharpPlus](#), [FsToolkit.ErrorHandling](#).

Plus d'info :

<https://kutt.it/pke2i1> *Validation with F# 5 and FsToolkit - Dec 2020*

3 ■ *Smart constructor*



« Making illegal states unrepresentable »

<https://kutt.it/MksmkG> F# for fun and profit, Jan 2013

- Avoir un design qui empêche d'avoir des états invalides
 - Encapsuler état (Σ *primitives*) dans un objet
- *Smart constructor* sert à garantir un état initial valide
 - Valide les données en entrée
 - Si Ko, renvoie "rien" (`Option`) ou l'erreur (`Result`)
 - Si Ok, renvoie l'objet créé wrappé dans l'`Option` / le `Result`

Encapsuler état dans un 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*

👉 Mot clé `private` :

→ Cache contenu de l'objet

→ Champs et constructeur ne sont plus visibles de l'extérieur

→ Smart constructeur défini dans module compagnon 👍 ou méthode statique

Smart constructor » Exemple 1

Smart constructeur :

→ Fonction `tryCreate` dans module compagnon

→ Renvoie une `Option`

```
type Latitude = private { Latitude: float } // ➡ Un seul champ, nommé comme le type

[<RequireQualifiedAccess>] // ➡ Optionnel
module Latitude =
    let tryCreate (latitude: float) =
        if latitude ≥ -90. && latitude ≤ 90. then
            Some { Latitude = latitude } // ➡ Constructeur accessible ici
        else
            None

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

F#

Smart constructor » Exemple 2

Smart constructeur :

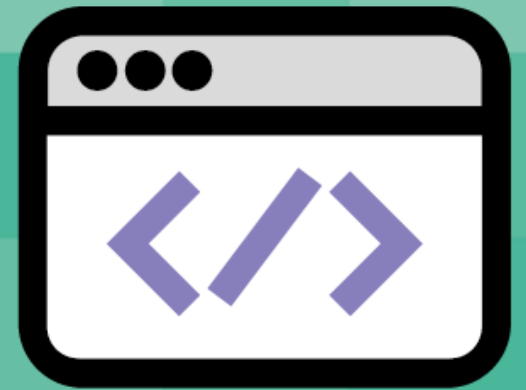
→ Méthode statique `of`

→ Renvoie `Result` avec erreur de type `string`

```
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" }
```

F#

4. ■ Computation expression



Computation expression

Sucre syntaxique cachant une « machinerie »

→ Applique la *Separation of Concerns*

→ Code + lisible à l'intérieur de la *computation expression* (CE)

Syntaxe : `builder { expr }`

→ `builder` instance d'un « Builder » 

→ `expr` peut contenir `let`, `let!`, `do!`, `yield`, `yield!`, `return`, `return!`

 **Note :** `seq`, `async` et `task` sont des CE

Builder

Une *computation expression* s'appuie sur un objet appelé *Builder*.

→ Cet objet permet éventuellement de stocker un état en background.

Pour chaque mot-clé supporté (`let!`, `return`...), le *Builder* implémente une ou plusieurs méthodes associées. Exemples :

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

Le *builder* peut également wrapper le résultat dans un type qui lui est propre :

- `async { return x } renvoie un type Async<'X>`
- `seq { yield x } renvoie un type Seq<'X>`

Builder desugaring

Le compilateur opère la traduction vers les méthodes du *builder*.

→ La CE masque la complexité de ces appels, souvent imbriqués :

```
seq {  
    for n in list do  
        yield n  
        yield n * 10 }  
  
// Traduit en :  
seq.For(list, fun () →  
    seq.Combine(seq.Yield(n),  
                seq.Delay(fun () → seq.Yield(n * 10)) ) )
```

F#

Builder - Exemple : **logger**

Besoin : logger les valeurs intermédiaires d'un calcul

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

F#

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

Problèmes ⚠

- ❶ Verbeux : les `log x` gênent lecture
- ❷ *Error prone* : oublier un `log`, logger mauvaise valeur...

Builder - Exemple : **logger** (2)

💡 Rendre les logs implicites dans une CE lors du **let!** / **Bind** :

```
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  
}
```

F#

Builder - Example : **maybe**

Besoin : simplifier enchaînement de "trySomething" renvoyant une **Option**

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

F#

```
// Sans CE
```

```
let division =
    36
    ▷ tryDivideBy 2           // Some 18
    ▷ Option.bind (tryDivideBy 3) // Some 6
    ▷ Option.bind (tryDivideBy 2) // Some 3
```

Builder - Example : maybe (2)

```
// Avec 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
}
```

F#

Bilan :  Symétrie,  Valeurs intermédiaires

Limite : imbrication de CE

- ✓ On peut imbriquer des CE différentes
- ✗ Mais code devient difficile à comprendre

Exemple : combiner `logger` et `maybe` ?

Solution alternative :

```
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
```

F#

Limite : combinaison de CE

Combiner `Async` + `Option` / `Result` ?

→ Solution : CE `asyncResult` + helpers dans [FsToolkit](#)

```
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>>
```

F#

CE : le couteau suisse ✨

Les *computation expressions* servent à différentes choses :

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

Fondements théoriques sous-jacents :

- Monoïde
- Monade
- Applicative

Monoïde

\simeq Type T définissant un ensemble comportant :

1. Opération $(+) : T \rightarrow T \rightarrow T$

→ Pour combiner des ensembles et garder le même "type"

→ Associative : $a + (b + c) \equiv (a + b) + c$

2. Élément neutre (*aka identity*) \simeq ensemble vide

→ Combinable à tout ensemble sans effet

→ $a + e \equiv e + a \equiv a$

CE monoïdale

Le builder d'une CE monoïdale (*telle que `seq`*) dispose *a minima* de :

- `Yield` pour construire l'ensemble élément par élément
- `Combine` \equiv `(+)` (`Seq.append`)
- `Zero` \equiv élément neutre (`Seq.empty`)

S'y ajoute généralement (entre autres) :

- `For` pour supporter `for x in xs do ...`
- `YieldFrom` pour supporter `yield!`

Monade

≈ Type générique `M<'T>` comportant :

1. Fonction `return` de construction

→ Signature : `(value: 'T) → M<'T>`

→ ≈ Wrap une valeur

2. Fonction `bind` de "liaison" (*aka opérateur `>=>`*)

→ Signature : `(f: 'T → M<'U>) → M<'T> → M<'U>`

→ Utilise la valeur wrappée, la "map" avec la fonction `f` vers une valeur d'un autre type et "re-wrap" le résultat

Monade : lois

`return` \equiv élément neutre pour `bind`

→ À gauche : `return x ▷ bind f` \equiv `f x`

→ À droite : `m ▷ bind return` \equiv `m`

`bind` est associatif

→ `m ▷ bind f ▷ bind g` \equiv `m ▷ bind (fun x → f x ▷ bind g)`

Monades et langages

Haskell

- Monades beaucoup utilisées. Les + communes : `IO`, `Maybe`, `State`, `Reader`.
- `Monad` est une *classe de type* pour créer facilement ses propres monades.

F#

- Certaines CE permettent des opérations monadiques.
- Plus rarement utilisées directement (*sauf par des Haskellers*)

C#

- Monade implicite dans LINQ
- Librairie [LanguageExt](#) de programmation fonctionnelle

CE monadique

Le builder d'une CE monadique dispose des méthodes `Return` et `Bind`.

Les types `Option` et `Result` sont monadiques.

→ On peut leur créer leur propre 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
```

F#

CE monadique et générique

[FSharpPlus](#) propose une CE `monad`

→ Marche pour tous les types monadiques : `Option`, `Result`, ... et même `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
```

F#

CE monadique et générique (2)

Exemple avec le type `Option` :

```
#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
```

F#

CE monadique et générique (3)

⚠ **Limite** : on ne peut pas mélanger plusieurs types monadiques !

```
#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
```

F#

CE monadiques spécifiques

Librairie [FsToolkit.ErrorHandling](#) propose :

- CE `option {}` spécifique au type `Option<'T>` (*exemple ci-dessous*)
- CE `result {}` spécifique au type `Result<'Ok, 'Err>`

👉 Recommandé car + explicite que CE `monad`

```
#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
```

F#

Applicative (a.k.a Applicative Functor)

≈ Type générique `M<'T>` -- 3 styles :

Style A: Applicatives avec `apply` / `<*>` et `pure` / `return`

- ❌ Pas facile à comprendre
- 🙅 Déconseillé par Don Syme dans cette [note de nov. 2020](#)

Style B: Applicatives avec `mapN`

- `map2`, `map3`... `map5` combine 2 à 5 valeurs wrappées

Style C: Applicatives avec `let! ... and! ...` dans une CE

- Même principe : combiner plusieurs valeurs wrappées
- Disponible à partir de F# 5 ([annonce de nov. 2020](#))

🙅 **Conseil :** Styles B et C sont autant recommandés l'un que l'autre.

CE applicative

Librairie [FsToolkit.ErrorHandling](#) propose :

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

Permet d'accumuler les erreurs → Usages :

- Parsing d'inputs externes
- *Smart constructor (Exemple de code slide suivante...)*

```
#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 be positive"
    else Ok height

let validateName name =
    if System.String.IsNullOrEmpty 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 be positive"]
let c3 = Customer.tryCreate "" 0<cm> // Error ["Name can't be empty"; "Height must be positive"]
```


Applicative vs Monad

“ Soit N opérations `tryXxx` renvoyant un `Option` ou `Result` ”

Style monadique :

- Avec `bind` ou CE `let! ... let! ...`
- **Chaîne** les opérations, exécutée 1 à 1, la N dépendant de la N-1
- S'arrête à 1ère opération KO → juste 1ère erreur dans `Result` ①
- [*Railway-oriented programming*](#) de Scott Wlaschin

```
module Result =  
    // f : 'T → Result<'U, 'Err>  
    // x': Result<'T, 'Err>  
    // → Result<'U, 'Err>  
    let bind f x' =  
        match x' with  
        | Error e → Error e // ➡ ①  
        | Ok value → f value
```

F#

Applicative vs Monad (2)

Style applicatif :

- Avec `mapN` ou CE `let! ... and! ...`
- **Combine** 2..N opérations indépendantes → parallélisables 👍
- Permet de combiner les cas `Error` contenant une `List` ②

```
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
```

F#

Autres CE

On a vu 2 librairies qui étendent F# et proposent leurs CE :

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

Beaucoup de librairies ont leur propre DSL (*Domain Specific Language.*)
Certaines s'appuient alors sur des CE :

- Expecto
- Farmer
- Saturn

Expecto

“ Librairie de testing : assertions + runner ”

 <https://github.com/haf/expecto>

```
open Expecto

let tests =
    test "A simple test" {
        let subject = "Hello World"
        Expect.equal subject "Hello World" "The strings should equal"
    }

[<EntryPoint>]
let main args =
    runTestsWithCLIArgs [] args tests
```

F#

Farmer

“ *Infrastructure-as-code* pour Azure ”

<https://github.com/compositionalit/farmer>

```
// Create a storage account with a container
let myStorageAccount = storageAccount {
    name "myTestStorage"
    add_public_container "myContainer"
}

// Create a web app with application insights that's connected to the storage account
let myWebApp = webApp {
    name "myTestWebApp"
    setting "storageKey" myStorageAccount.Key
}

// [ ... ]
```

F#

Farmer (2)

```
// [ ... ]  
  
// Create an ARM template (Azure Resource Manager)  
let deployment = arm {  
    location Location.NorthEurope  
    add_resources [  
        myStorageAccount  
        myWebApp  
    ]  
}  
  
// Deploy it to Azure!  
deployment  
▷ Writer.quickDeploy "myResourceGroup" Deploy.NoParameters
```

F#

Saturn

“ Framework Web au-dessus de [ASP.NET](https://saturnframework.org/) Core, pattern MVC ”

 <https://saturnframework.org/>

```
open Saturn
open Giraffe

let app = application {
    use_router (text "Hello World from Saturn")
}

run app
```

F#

CE : aller + loin

Extending F# through Computation Expressions

 <https://youtu.be/bYorooBgvws>

 <https://panesofglass.github.io/computation-expressions/#/>

→ Références en [slide 14](#).

Computation Expressions Workshop

 <https://github.com/panesofglass/computation-expressions-workshop>

5. ■ Le Récap'



Types unions : `Option` et `Result`

- A quoi ils servent :
 - Modéliser absence de valeur et erreurs métier
 - Opérations partielles rendues totales `tryXxx`
 - *Smart constructor* `tryCreate`
- Comment on s'en sert :
 - Chaînage : `map`, `bind`, `filter` → *ROP*
 - Pattern matching
- Leurs bénéfices :
 - `null` free, `Exception` free → pas de guard polluant code
 - Rend logique métier et *happy path* + lisible

Computation expression (CE)

- Sucre syntaxique : syntaxe intérieure standard ou "bangée" (`let!`)
- *Separation of Concerns* : logique métier vs « machinerie »
- Compilateur fait lien avec *builder*
 - Objet stockant un état
 - Build une valeur en sortie, d'un type spécifique
- Imbricables mais pas faciles à combiner !
- Concepts théoriques sous-jacents
 - Monoïde → `seq` (*d'éléments composables et avec un "zéro"*)
 - Monade → `async`, `option`, `result`
 - Applicative → `validation` / `Result<'T, 'Err list>`
- Bibliothèques : FSharpPlus, FsToolkit, Expecto, Farmer, Saturn



Ressources complémentaires

Compositional IT (*Isaac Abraham*)

→ <https://kutt.it/gplgfD> • *Writing more succinct C# – in F#! (Part 2)* • Jul 2020

F# for Fun and Profit (*Scott Wlaschin*)

→ <https://kutt.it/e78rNj> • *The Option type* • Jun 2012

→ <https://kutt.it/7J5Krc> • *Making illegal states unrepresentable* • Jan 2013

→ <https://kutt.it/drchkQ> • Série de 11 articles sur les CE • Jan 2013

→ <https://kutt.it/ebfGNA> • Série de 7 articles sur monades 'n co • Aug 2015

Merci 🙏

SOAT

→ Digitalize society



SOAT.FR