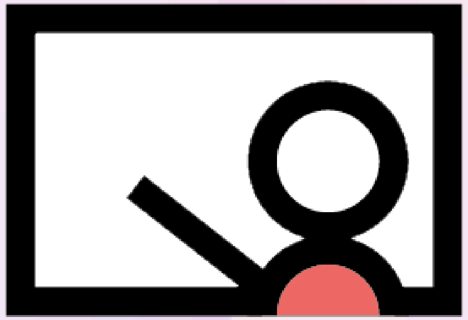
F♯ Training 📎

Functional patterns for computation expressions

2025 July

Table of contents



1. Intro

Preliminary note ☝️

The word **“pattern”** is used here rather than *concept* or *abstraction*, with a broader meaning than the ~~design patterns~~ from OOP.

Languages hidden patterns 🔎

F♯ uses functional patterns under the hood:

- `Option` and `Result` are monadic types
- `Async` is monadic
- Collection types `Array`, `List` and `Seq` are monadic types too!
- Computation expressions can be monadic or applicative or monoidal

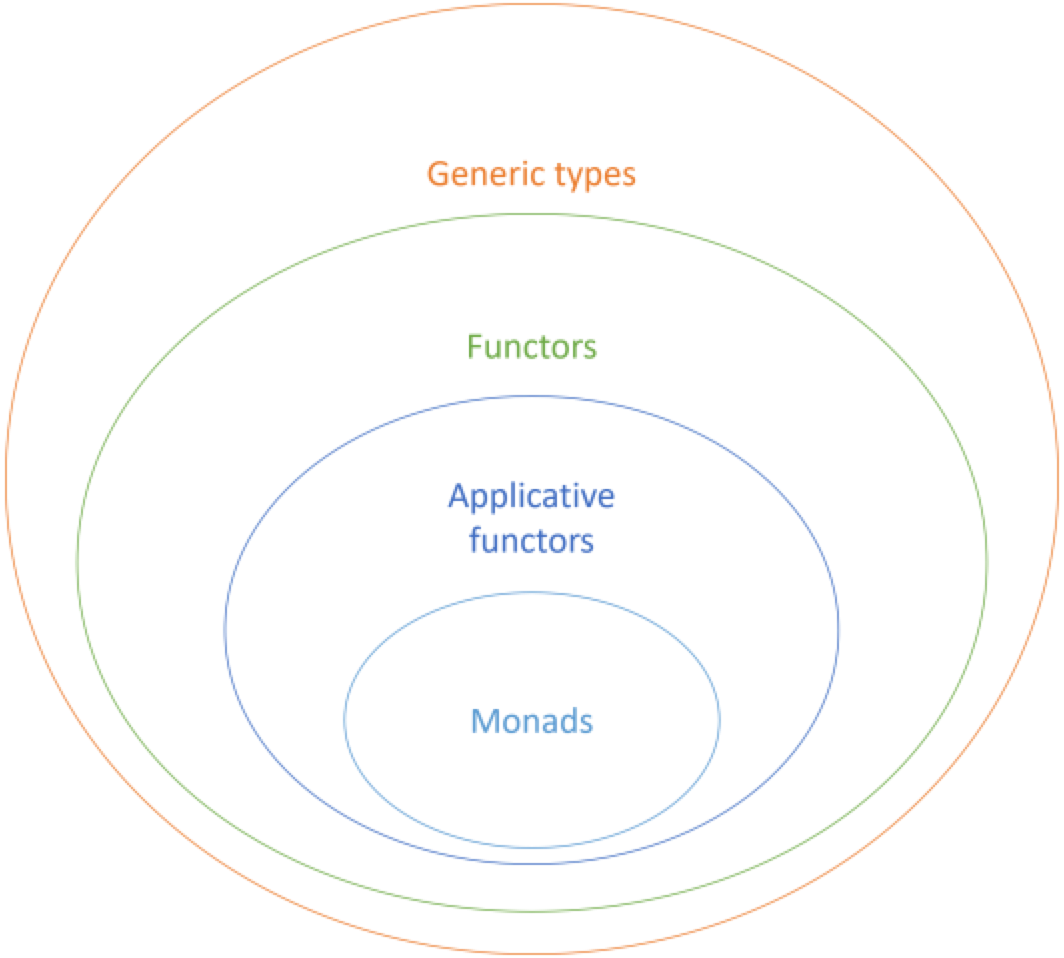
C♯ uses functor and monad under the hood too, via the LINQ query syntax.

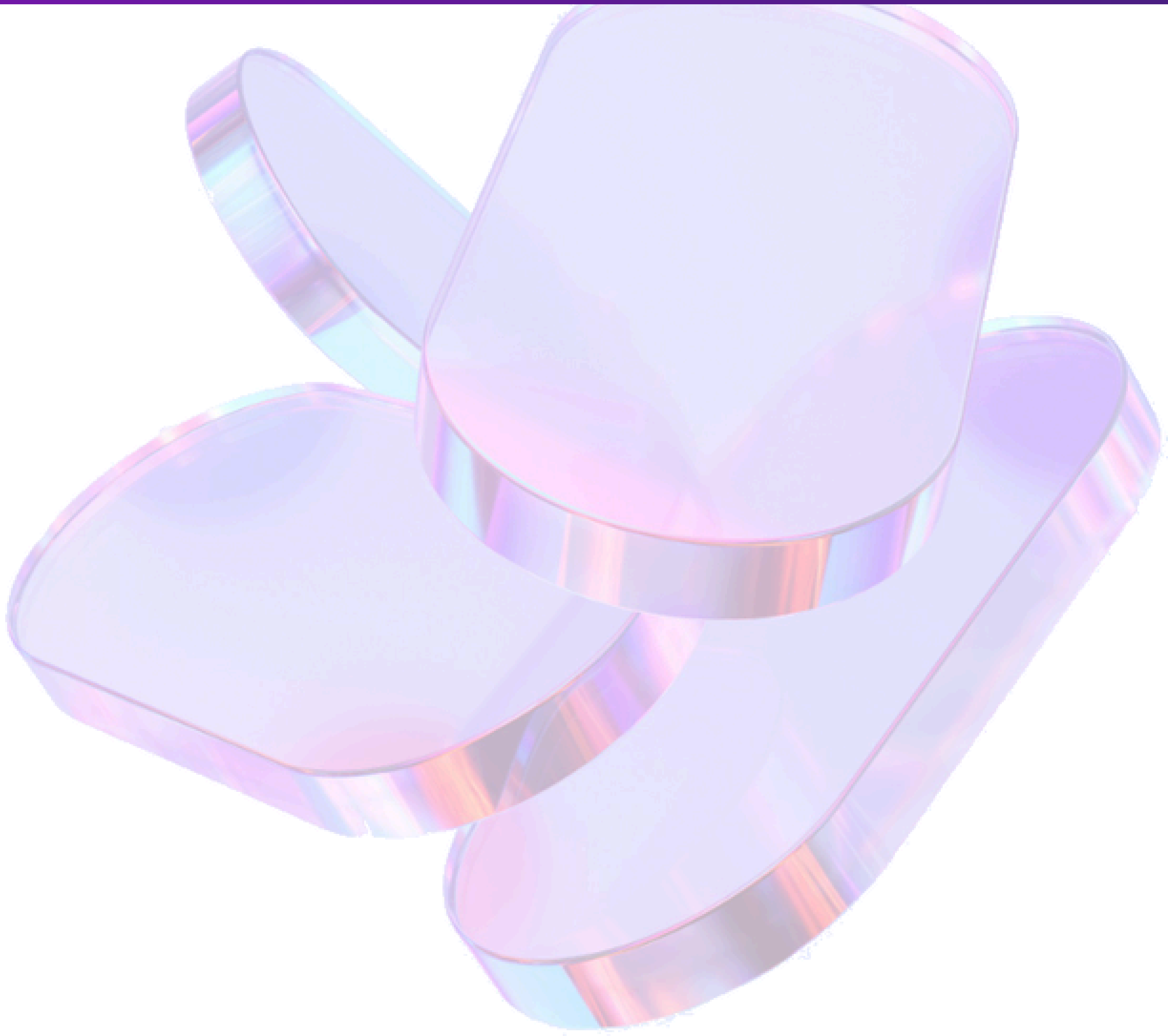
Functional patterns overview 🗺️

The patterns presented here:

- Come from the *category theory*, a branch of mathematics
- Are useful to understand how to write computation expressions in F♯
- Consist of
  - A type, mainly a *generic type* `X<'a>` containing elements of type `'a`
  - 1 or 2 operations on this type
  - An eventual special instance of this type
  - Some laws constraining/shaping the whole

Monad big picture





🔗 Monads Series *(by Mark Seemann)*

2. Monoid

Monoid definition

Etymology (Greek): `monos` *(single, unique)* · `eidos` *(form, appearance)*

≃ Type `T` defined with:

- Binary operation `+`: `T → T → T`
  → To *combine* 2 elements into 1
- Neutral element `e`
  (a.k.a. *identity*)

Monoid laws

1. Associativity

`+` is associative
→ `a + (b + c)` ≡ `(a + b) + c`

2. Identity Element

`e` is combinable with any instance `a` of `T` without effect
→ `a + e` ≡ `e + a` ≡ `a`

Monoid examples

Type	`+`	Identity `e`	Law 2
`int`	`+` *(add)*	`0`	`i + 0 = 0 + i = i`
`int`	`*` *(multiply)*	`1`	`i * 1 = 1 * i = i`
`string`	`+` *(concat)*	`""` *(empty string)*	`s + "" = "" + s = s`
`'a list`	`@` (`List.append`)	`[]` *(empty list)*	`l @ [] = [] @ l = l`
Functions	`>>` *(compose)*	`id` (`fun x -> x`)	`f >> id = id >> f = f`

💡 The monoid is a generalization of the **Composite** *OO design pattern*
🔗 Composite as a monoid *(by Mark Seemann)*

3. Functor

Functor definition

≃ Any generic type, noted `F<'T>`, with a `map` operation:

- Signature: `map: (f: 'T → 'U) → F<'T> → F<'U>`

`map` preserves the structure: e.g. mapping a `List` returns another `List`.

Functor laws

Law 1 - Identity law

Mapping the `id` function over a Functor `F` should not change `F`.
→ `map id F` ≡ `F`

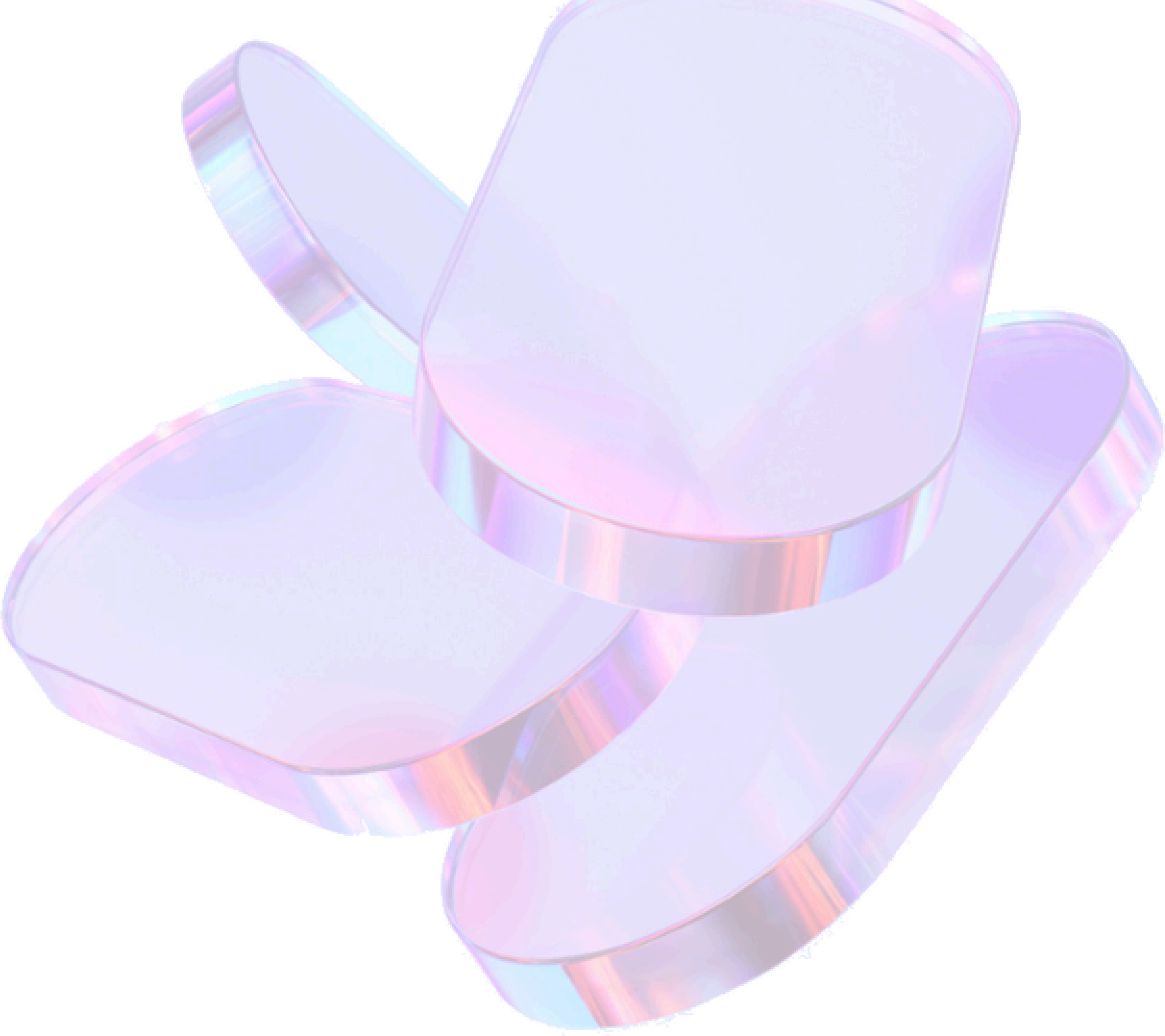
Law 2 - Composition law

Mapping the composition of 2 functions `f` and `g` is the same as mapping `f` and then mapping `g` over the result.
→ `map (f >> g)` ≡ `map f >> map g`

Functor examples

Type	Map
`Option<'T>`	`Option.map`
`Result<'T, _>`	`Result.map`
`List<'T>`	`List.map`
`Array<'T>`	`Array.map`
`Seq<'T>`	`Seq.map`

`Async<'T>` too, but through the `async` CE 📍



4. Monad

Monad definition

≃ Any generic type, noted `M<'T>`, with:

- Construction function `return`
  - Signature : `(value: 'T) → M<'T>`
  - ≃ Wrap (lift/elevate) a value
- Chaining function `bind`
  - Noted `>>=` (`>` `>` `=`) as an infix operator
  - Signature : `(f: 'T → M<'U>) → M<'T> → M<'U>`
  - Take a **monadic function** `f`
  - Call it with the eventual wrapped value(s)
  - Get back a new monadic instance of this type

Monad laws (1-2/3)

1. Left Identity

`return` then `bind` are neutral.
→ `return >> bind f` ≡ `f`

2. Right Identity

`bind return` is neutral, equivalent to the `id` function:
→ `m ▷ bind return` ≡ `m ▷ id` ≡ `m`

☝️ It's possible because `return` has the signature of a monadic function.

Monad laws (3/3)

3. Associativity

`bind` is associative.

Given 2 monadic functions `f: 'a → M<'b>` and `g: 'b → M<'c>`
→ `(m ▷ bind f) ▷ bind g` ≡ `m ▷ bind (f >> bind g)`

💡 `bind` allows us to chain monadic functions, like the `▷` for regular functions

Monad examples

Type	Bind	Return
`Option<'T>`	`Option.bind`	`Some`
`Result<'T, _>`	`Result.bind`	`Ok`
`List<'T>`	`List.collect`	`List.singleton`
`Array<'T>`	`Array.collect`	`Array.singleton`
`Seq<'T>`	`Seq.collect`	`Seq.singleton`

`Async<'T>` too, but through the `async` CE 📍

Monad *vs* Functor

- A monad is also a **functor**
- `map` can be expressed in terms of `bind` and `return`:
  `map f` ≡ `bind (f >> return)`

☝️ **Note:** Contrary to the monad with its `return` operation, the functor concept does not need a "constructor" operation.

Monad alternative definition

A monad can be defined with the `flatten` operation instead of the `bind`
→ Signature: `M<M<'T>> → M<'T>`

Then, the `bind` function can be expressed in terms of `map` and `flatten`:
→ `bind` ≡ `map >> flatten`

💡 This is why `bind` is also called `flatMap`.

Regular functions *vs* monadic functions

Function	Op	Signature
Pipeline		
Regular	`|>` *pipe*	`(f: 'a -> 'b)      -> (x: 'a)     -> 'b`
Monadic	`>>=` *bind*	`(f: 'a -> M<'b>) -> (x: M<'a>) -> M<'b>`
Composition		
Regular	`>>` *comp.*	`(f: 'a -> 'b)      -> (g: 'b -> 'c)     -> ('a -> 'c)`
Monadic	`>=>` *fish*	`(f: 'a -> M<'b>) -> (g: 'b -> M<'c>) -> ('a -> M<'c>)`

- Fish operator definition: `let (>=>) f g = fun x -> f x |> bind g` ≡ `f >> (bind g)`
- Composition of monadic functions is called *Kleisli composition*

Monads *vs* Effects

Effect *(a.k.a. "side effect")*:
→ change somewhere, inside the program *(state)* or outside
→ examples:

- **I/O** *(Input/Output)*: file read, console write, logging, network requests
- **State Management:** global variable update, database/table/row delete
- **Exceptions/Errors:** program crash
- **Non-determinism:** same input → ≠ value: random number, current time
- **Concurrency/Parallelism:** thread spawn, shared memory

Pure function causes no side effects → deterministic, predictable
→ FP challenge: separate pure/impure code *(separation of concerns)*

Monads *vs* Effects (2)

Monads purposes:

- Encapsulate and sequence computations that involve effects,
- Maintain purity of the surrounding functional code,
- Provide a controlled environment in which effects can happen.

Dealing with an effectful computation using monads means:

1. **Wrapping:** we don't get a value directly, we get a monadic value that represents the computation and its associated effect.
2. **Sequencing:** `bind` (or `let!` in a monadic CE) allows you to chain together effectful computations in a sequential order.
3. **Returning:** `return` wraps a **pure** value → computation w/o effects.
   👉 A monadic sequence can mix pure and effectful computations.

Monads *vs* Effects (3)

From the *caller* perspective, a function returning a monadic value is **pure.**
→ Encapsulated effects only "happen" when monadic value is **evaluated.**

Examples in F♯:

- `Async`: by calling `Async.RunSynchronously`/`Start`
- `Option`/`Result`: by pattern matching and handle all cases
- `Seq`: by iterating the delayed sequence of elements

👉 Monads effectively bridge the gap between:

- mathematical elegance of pure functional programming
- practical necessity of interacting with an impure, stateful world

Other common monads

☝️ *Rarely used in F♯, but common in Haskell*

- **Reader**: to access a read-only environment (like configuration) throughout a computation without explicitly passing it around
- **Writer**: accumulates monoidal values (like logs) alongside a computation's primary result
- **State**: manages a state that can be read and updated during a computation
- **IO**: handles I/O effects (disk, network calls...)
- **Free**: to build series of instructions, separated from their execution (interpretation phase)

4. Applicative (Functor)

Applicative definition

≃ Any generic type, noted `F<'T>`, with:

- Construction function `pure` (≡ monad's `return`)
  - Signature : `(value: 'T) → F<'T>`
- Application function `apply`
  - Noted `<*>` (same `*` than in tuple types)
  - Signature : `(f: F<'T → 'U>) → F<'T> → F<'U>`
  - Similar to functor's `map`, but where the mapping function `'T → 'U` is wrapped in the applicative object

Applicative laws (1/4)

There are 4 laws:

- *Identity* and *Homomorphism* relatively easy to grasp
- *Interchange* and *Composition* more tricky

Law 1 - Identity

Same as the functor identity law applied to applicative:

Pattern	Equation
Functor	`map id F` ≡ `F`
Applicative	`apply (pure id) F` ≡ `F`

Applicative laws (2/4)

Law 2 - Homomorphism

💡 *Homomorphism* means a transformation that preserves the structure.

→ `pure` does not change the nature of values and functions so that we can apply the function to the value(s) either before or after being wrapped.

`(pure f) <*> (pure x)` ≡ `pure (f x)`
`apply (pure f) (pure x)` ≡ `pure (f x)`

Applicative laws (3/4)

Law 3 - Interchange

We can provide the wrapped function `Ff` first or the value `x`, wrapped directly or captured in `(▷) x` *(partial application of the* `▷` *operator used as function)*

`Ff <*> (pure x)` ≡ `pure ((▷) x) <*> Ff`

💡 When `Ff` = `pure f`, we can verify this law with the homomorphism law:

```
apply Ff (pure x)           | apply (pure ((▷) x)) Ff
apply (pure f) (pure x)     | apply (pure ((▷) x)) (pure f)
pure (f x)                  | pure (((▷) x) f)
                            | pure (x ▷ f)
                            | pure (f x)
```

Applicative laws (4/4)

Law 4 - Composition

- Cornerstone law: ensures that function composition works as expected within the applicative context.
- Hardest law, involving to wrap the `<<` operator (right-to-left compose)!

`Ff <*> (Fg <*> Fx)` ≡ `(pure (<<) <*> Ff <*> Fg) <*> Fx`

💡 Same verification:

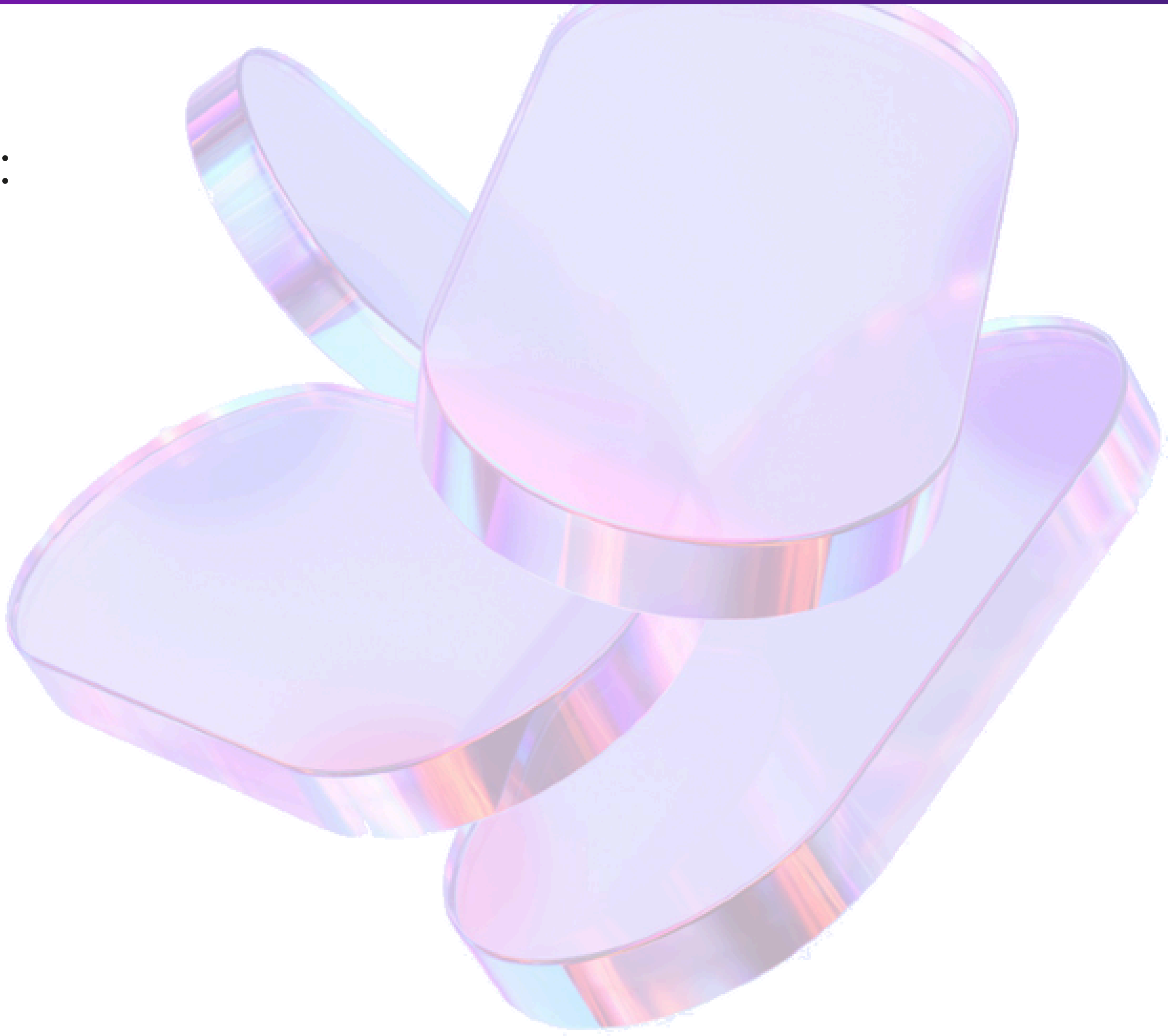
```
(pure f) <*> ((pure g) <*> (pure x))    | (pure (<<) <*> (pure f) <*> (pure g)) <*> (pure x)
(pure f) <*> (pure g x)                 | (pure ((<<) f) <*> (pure g)) <*> (pure x)
pure (f (g x))                          | (pure ((<<) f g)) <*> (pure x)
pure ((f << g) x)                       | (pure (f << g)) <*> (pure x)
                                        | pure ((f << g) x)
```

Applicative *vs* Functor

Every applicative is a functor
→ We can define `map` with `pure` and `apply`:

`map f x` ≡ `apply (pure f) x`

💡 It was implied by the 2 identity laws.



Applicative *vs* Monad

Every monad is also an applicative

- `pure` and `return` are just synonym
- `apply` can be defined using `bind`
  - given `mx` a wrapped value `M<'a>`
  - and `mf` a wrapped function `M<'a -> 'b>`
  - `apply mf mx` ≡ `mf |> bind (fun f -> mx |> bind (fun x -> return (f x)))`

`apply` *vs* `bind` 💡

- Where `apply` unwraps both `f` and `x`, 2 nested `bind`s are required.
- `bind` extra power comes from its ability to let its 1st parameter — the function `'a -> M<'b>` — create a whole new computational path.

Applicative: multi-param curried function

Applicative helps to apply to a function its arguments (e.g. `f: 'x → 'y → 'res`) when they are each wrapped (e.g. in an `Option`).

Let's try by hand:

```
let call f optionalX optionalY =
    match (optionalX, optionalY) with
    | Some x, Some y → Some(f x y)
    | _ → None
```

💡 We can recognize the `Option.map2` function.

🤔 Is there a way to handle any number of parameters?

Applicative: multi-param function (2)

The solution is to use `apply` N times, for each of the N arguments, first wrapping the function using `pure`:

```
// apply and pure for the Option type
let apply optionalF optionalX =
    match (optionalF, optionalX) with
    | Some f, Some x -> Some(f x)
    | _ -> None

let pure x = Some x

// ---

let f x y z = x + y - z
let optionalX = Some 1
let optionalY = Some 2
let optionalZ = Some 3
let res = pure f |> apply optionalX |> apply optionalY |> apply optionalZ
```

Applicative: multi-param function (3)

We can "simplify" the syntax by:

- Replacing the 1st combination of `pure` and `apply` with `map`
- Using the operators for map `<!>` and apply `<*>`

```
// ...
let res = pure f |> apply optionalX |> apply optionalY |> apply optionalZ

let res' = f <!> optionalX <*> optionalY <*> optionalZ
```

Still, it's not ideal!

Applicative - 3 styles

The previous syntax is called **“Style A”** and is not recommended in modern F♯ by Don Syme - see its [Nov. 2020 design note](#).

When we use the `mapN` functions, it's called **“Style B”**.

The **“Style C”** relies on F♯ 5 `let! ... and! ...` in a CE like `option` from `FsToolkit`:

```
let res'' =
    option {
        let! x = optionalX
        and! y = optionalY
        and! z = optionalZ
        return f x y z
    }
```

👉 Avoid style A, prefer style C when a CE is available, otherwise style B.

Applicative *vs* Monadic behaviour

The monadic behaviour is **sequential**:
→ The computation #n+1 is done only after the computation #n.

The applicatives behave in **parallel**:
→ All the computations for the arguments are done before applying them to the wrapped function.

👉 Even if monads can do more things, applicatives can be more performant on what they can do.

Applicative parallel behaviour

The corollary is about the `Result` type and its `bind` function:

```
let bind (f: 'a → Result<'b, _>) result =
    match result with
    | Ok x → f x
    | Error e → Error e
```

→ As soon as the current `result` is an `Error` case, `f` is ignored.
→ On the 1st error, we "unplug".

Applicative parallel behaviour (2)

Given the `Result<'ok, 'error list>` type, the `apply` below can **accumulate errors:**

```
(* 1 *) let apply (rf: Result<'a -> 'b, 'err list>) (result: Result<'a, 'err list>) : Result<'b, 'err
(* 2 *)     match rf, result with
(* 3 *)     | Ok f, Ok x -> Ok(f x)
(* 4 *)     | Error fErrors, Ok _ -> Error fErrors
(* 5 *)     | Ok _, Error xErrors -> Error xErrors
(* 6 *)     | Error fErrors, Error xErrors -> Error(xErrors @ fErrors)
```

☝️ **Notes:**

- Errors are either accumulated *(L6)* or propagated *(L4, L5)*.
- At lines L4, L6, `rf` is no longer a wrapped function but an `Error`. It happens after a first `apply` when there is an `Error` instead of a wrapped value *(L5, L6)*.

💡 Handy for validating inputs and reporting all errors to the user.

🔗 [Validation with F# 5 and FsToolkit](), Compositional IT

5. Wrap up

Functional patterns key points

Pattern	Key words
Monoid	`+` (combine), composite design pattern ++
Functor	`map`, preserve structure
Monad	`bind`, functor, flatten, effects, sequential composition
Applicative	`apply`, functor, multi-params function, parallel composition

Functional patterns in F♯

In F♯, these functional patterns are applied under the hood:

- Monoids with `int`, `string`, `list` and functions
- Monads with `Async`, `List`, `Option`, `Result`...
- All patterns when using computation expressions

☝️ After the beginner level, it's best to know the principles of these patterns, in case we need to write computation expressions.

Functional patterns in F♯ (2)

🤔 Make these patterns more explicit in F♯ codebases

Meaning: what about F♯ codebases full of `monad`, `Reader`, `State`...?

- Generally **not recommmended**, at least by Don Syme
  - Indeed, the F♯ language is not designed that way.
  - Albeit, libraries such as *FSharpPlus* offer such extensions to F♯. 📍
- To be evaluated for each team: idiomatic *vs* consistency ⚖️
  → Examples:
  - **Idiomatic F♯** in .NET teams: using both C♯ and F♯ code
  - **Functional F♯** in FP team: using F♯, Haskell and/or OCaml

Additional resources 🔗

- [*The "Map and Bind and Apply, Oh my!" series*], F# for Fun and Profit
- Applicatives IRL, Jeremie Chassaing
- Functional patterns | F# training GitBook, Romain Deneau

Thanks 🙏

d-edge