Functional Programming in C#

Part 2

Danial Khosravi

Part 1 Summary

- Immutability
- Purity
- Effects, Option

Talk Objectives

- **Either** effect and functional error handling
- **Task** effect and handling asynchronosity funtionally
- **Applicative** and functions in an elevated world

Functional Error Handling With Either

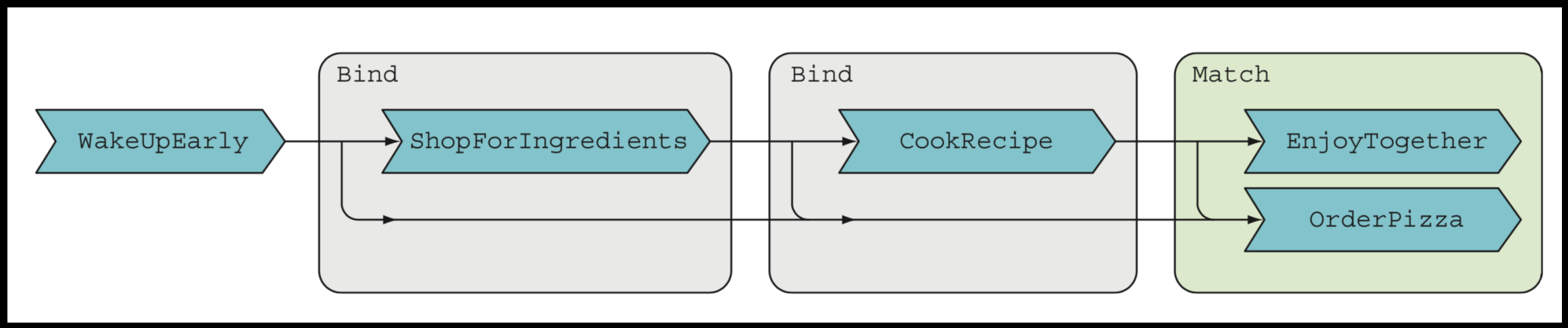
Either<L, R> = Left(L) | Right(R)

- Models the absence of valid value and the reason(s)
- Models things that can go one way or another
- *Right* = All right
- *Left* = something went wrong
- Great choice for dealing with exceptions in a pure functional fashion

Either is a Monad, so it has **Bind** which allows us to chain Either based functions

```
public static Either<Rejection, Candidate> CheckEligibility(Candidate c);
public static Either<Rejection, Candidate> CheckTest(Candidate c);
public static Either<Rejection, Candidate> Interview(Candidate c);

Either<Rejection, Candidate> Recruit (Candidate candidate)
{
  // Right: Utility to lift candidate to the elevated
  // world of Either<Rejection, Candidate>
  return Right(candidate)
    .Bind(CheckEligibility)
    .Bind(CheckTest)
    .Bind(Interview)
}
```

```
   o WakeUpEarly
  / \
 L   R ShopForIngredients
    / \
   L   R CookRecipe
      / \
     L   R EnjoyTogether
```



Example 4

- Using Either to handle errors

C# LINQ Syntax

- Typically used for writing SQL like queries on **IEnumerable** or **IQueryable** data sources

```
from name in namesList select name.ToUpper()
```

- Works on any type given **Select**, **SelectMany** and **Where** are implemented
- Very useful for chaining Effectful/Monadic functions
- *LanguageExt* library provides extension methods for all common effects (e.g Task, Option, Either)

Example 4 - LINQ Syntax

- LINQ Syntax example
- Review example 4 using the LINQ syntax

Try and TryAsync and Exceptional Functions

- Often we already have exceptional functions that want to reuse in our functional code without changing their type/implementation

```
var price = Try(() => int.Parse("1")); // Try<int>
var priceTimesTwo = price.Map(x => x * 2);
priceTimesTwo
    .Match(
        (result) => Console.WriteLine(result),
        (ex) => Console.WriteLine(ex.Message)
    );
```

TryAsync

```
var priceAsync = TryAsync(() => Task.FromResult(int.Parse("1"))); // TryAsync<int>
var priceAsyncTimesTwo = priceAsync.Map(x => x * 2);
await priceAsyncTimesTwo
    .Match(
        (result) => Console.WriteLine(result),
        (ex) => Console.WriteLine(ex.Message)
    );
```

Working with Multi-Argument Effectful Functions

Problem: Multi-Argument Effectful Functions

- **Map** and **Bind** are powerful
- Both take unary functions (functions with 1 argument)
- Real world applications are complex and often composed of functions with many arguments
- Often come from many sources e.g configuration, HTTP APIs and databases
- How to use multi-argument **effectful** functions in an **elevated world**?

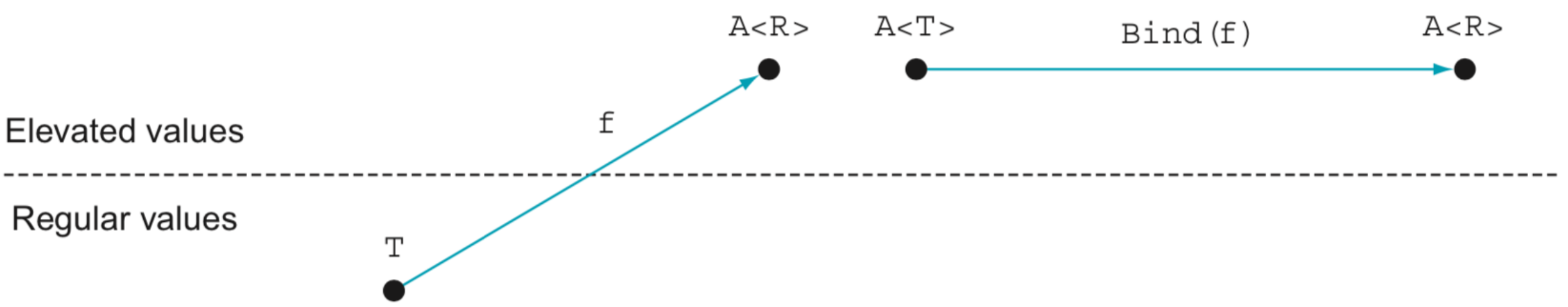
Solutions: **Monadic** or **Applicative** approaches

Monadic Approach

- perfect for sequential application flows
- perfect for multi argument functions where some arguments have dependencies on each other
- Given **userId**, **Transaction**
  => **User** -> **Transaction[]** -> **EnrichTransactions[]** -> **CheckFraud**
- How? by chaining functions using **Bind/flatMap/SelectMany**

Bind/flatMap/SelectMany

- Chain sequential monadic functions using **Bind/flatMap/SelectMany**
- Given **userId**, **Transaction**
  => **User** -> **Transaction[]** -> **EnrichTransactions[]** -> **CheckFraud**



Solutions: **Monadic** or **Applicative** approaches

Applicative Approach

- perfect for parallel application flows
- perfect for multi argument functions with arguments that have no dependency on each other
- Given *userId* =>
  | **User**
  | **Connections**
  | **Photo Albums**
  ----------------> **GenerateProfile**
- How? by providing arguments using the applicatives' **Apply** function

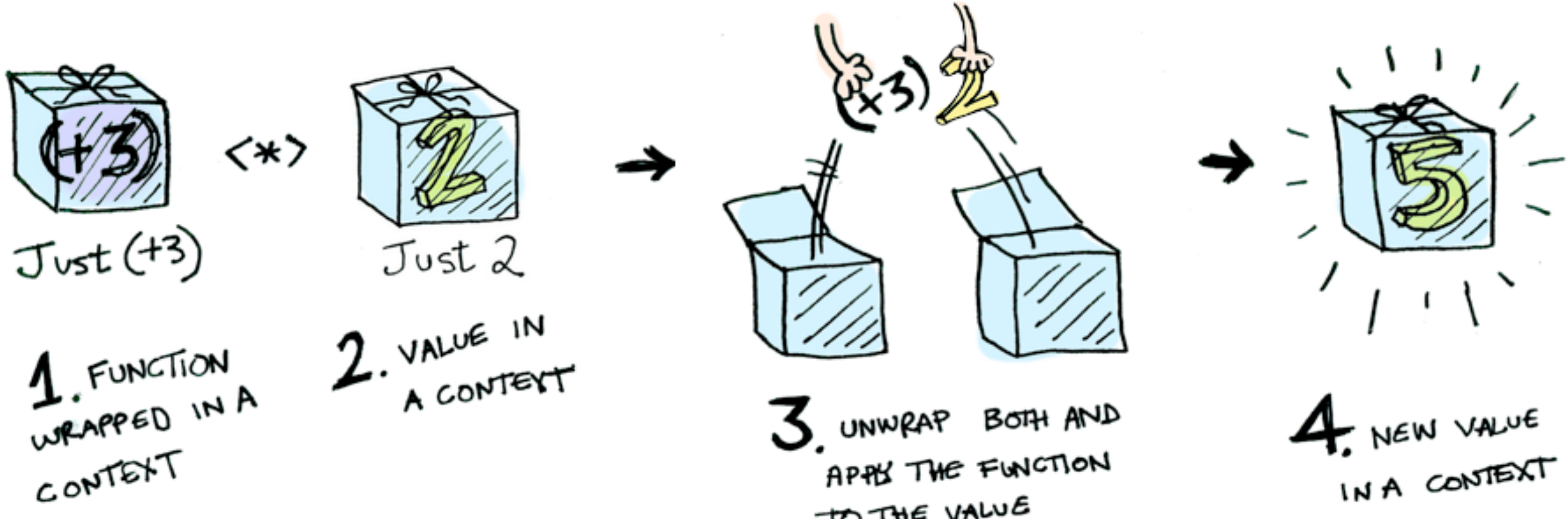
Applicatives

Applicatives

- **Apply** is a common functional HOF like Map and Bind
- It used to provide arguments to **functions** in an **elevated world**
- Functions (*Func<T>*) are just like any other values
- Can be wrapped in a context/effect/box to be lifted in an elevated world
- Useful for applying elevated values to functions that have non elevated arguments
- Idea of a wrapped function is abstract, lets see an example

```
Func<int, int, int> multiply = (x, y) => x * y;

/* Regular world */
multiply(2, 3)          // 6

/* Elevated world */
Some(multiply)          // Some(x => y => x * y))
  .Apply(Some(2))       // Some(y => 2 * y))
  .Apply(Some(3))       // Some(6)
```

```
Func<User, User[], Album[], string> GenerateProfile;

Func<int, Task<User>> FetchUser;
Func<int, Task<User[]>> FetchConnections;
Func<int, Task<Album[]>> FetchAlbums;

Task.FromResult(GenerateProfile)
    .Apply(FetchUser(userId))
    .Apply(FetchConnections(userId))
    .Apply(FetchAlbums(userId))
```


(+3)
<*>
2
Just (+3)
Just 2
1. FUNCTION WRAPPED IN A CONTEXT
2. VALUE IN A CONTEXT
(+3)
2
3. UNWRAP BOTH AND APPLY THE FUNCTION TO THE VALUE
5
4. NEW VALUE IN A CONTEXT

Example 5 and Example 6

- Generate social media profile using the **Monadic** approach
- Generate social media profile using the **Applicative** approach

Traversables

working with lists of elevated values

Transforms MA<MB<T>> to MB<MA<T>>

```
// Sometimes we have list of elevated values
var as = new Option<int>[] { Some(1), Some(2), Some(3) };
var ys = new Either<Exception, int>[] { Right(1), Right(2), Right(3) };
var zs = new Task<int>[] { Task.FromResult(1), Task.FromResult(2), Task.FromResult(3) };

// But we want an elevated list of values
var xs = Some(new int[] {1, 2, 3});
var ys = Right(new int[] {1, 2, 3});
var zs = Task.FromResult(new int[] {1, 2, 3});
```

Example Traversables

Example 7 and Example 8

- Example 7: Generating profile for many users at once
- Example 8: Generating profile for many users at once, while handling failures