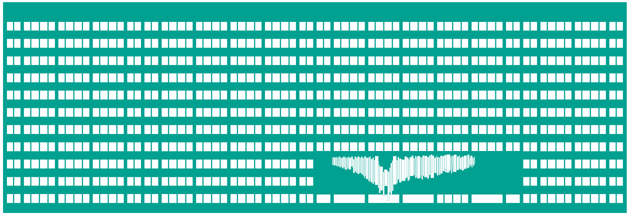
VŠB TECHNICKÁ UNIVERZITA OSTRAVA

VSB TECHNICAL UNIVERSITY OF OSTRAVA



www.vsb.cz

Monads in C#

behalek.cs.vsb.cz/wiki/Practical_Functional_Programming

Marek Běhálek

VSB – Technical University of Ostrava

marek.behalek@vsb.cz

October 21, 2022

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Motivation I

- We are learning functional programming - but what it really means?
- One way how to look at the problem is, we want to learn pure functional programming language Haskell.
  - Nice is some ways: basics principles are exposed, Haskell by itself is a nice (unique) programming language.
  - But do we really want to be Haskell programmers (some does)? → We can be Scala or Clojure programmers (probably more practical). → But fundamentally, the question remains.
  - This in fact means, we need to learn also API, tools,...
- Second option is: Functional programming is in fact programming paradigm → It represents a *style* of programming. Now, the right question is:
  - Can I accommodate the functional style of programming in my preferred programming language?
  - Does it brings something good?
  - Is it superior to other styles of programming?

Motivation II

- So, how do we compare programming styles?
  - No tool is perfect in every situation and a lot depends on personal opinion (preferences) → endless debate with no winner.
- We should probably agree on (something like:-) following conditions:
  - Keep it simple → code, language → keep only what is necessary.
  - Compiler is your friend → let it do as much work as possible.
  - Errors are bad → runtime error are the worst → best is, if there is no possibility to make error.
  - You can add your criteria here.

Functional programming vs OOP (1)

- Today, probably the most popular programming style is Object Oriented Programming.
- Following definition was taken from Wikipedia: What are the key points of OOP?
- Object Oriented Programming - objects and (most often) classes
  - Encapsulation - data are hidden inside and are accessible only trough given interface.
  - *Abstraction* - objects can be black boxes and we can use them even if do not know how they are working inside (works for most programming styles).
  - Composition, inheritance, and delegation - objects can be white boxes and new objects can be created with/based on existing objects.
  - Polymorphism - in OOP, it is usually referring to a situation, when calling code can be agnostic as to which class in the supported hierarchy it is operating on.
- Object-oriented programming makes code understandable by encapsulating moving parts. Functional programming makes code understandable by minimizing moving parts. (M. Feathers)

Functional programming vs OOP (2)

- So, what are the differences?
- Abstraction - mostly personal preference.
- Polymorphism
  - FP: Parametric polymorphism.
  - OOP: Subtyping, subtype polymorphism.
- Composition
  - FP: Composition works fine for *pure* functions (*function composition*, *high order functions*) → Functions with side effects can be recognized at first glance.
  - OOP: Composition works until you need to concurrently work with data (frequent scenario now) → then the problems are:
    - again side effects - but they can be hidden now;
    - *encapsulation* - fundamental OOP principle that hides data in object.
  - Does it really matters? We do not need to use *all* language features.

Examples of *wrong* OOP

- Methods in constructor can not be reordered. We know that only if we *understand* the program.

```
public class Game
{
  List<Item> items;
  Player player;
  World world;

  public Game() {
    LoadItems();
    LoadPlayer();
    CreateWorld();
  }

  private void LoadItems()
  {
    items = new List<Item>();
  }
  private void LoadPlayer()
  {
    player = new Player(this.items);
  }
  private void CreateWorld()
  {
    world = new World(this.items, this.player);
  }
}
```

Examples of *wrong* OOP (2)

```
public struct ValidityDateRange {
  public DateTime Start { get; set; }
  public DateTime End { get; set; }

  public bool IsInEffect(DateTime date) {
    return Start.CompareTo(date) <= 0
          && End.CompareTo(date) >= 0;
  }
  public void Extend(int days) {
    End.AddDays(days);
  }
}
public class Card {
  public string SerialNumber { get; set; }
  public ValidityDateRange Validity { get; set; }
}

public static void Test() {
  var valid = new ValidityDateRange() {
    Start = DateTime.Parse("2021-03-19"),
    End = DateTime.Parse("2021-03-22")
  };
  var card1 = new Card() {
    SerialNumber = "123456",
    Validity = valid
  };
  var card2 = new Card() {
    SerialNumber = "654321",
    Validity = valid
  };
  //What will be written?
  card2.Validity.Extend(7);
  Console.WriteLine($"Card {card1.SerialNumber},
    validity: {card1.Validity.IsInEffect(
      DateTime.Parse("2021-03-24"))}");
  Console.WriteLine($"Card {card2.SerialNumber},
    validity: {card2.Validity.IsInEffect(
      DateTime.Parse("2021-03-24"))}");
}
```

What we really need to apply FP style in C#?

- Today's most popular programming languages are mostly multi-paradigm languages → they support various style of programming.
- What we really need for functional style of programming?
  - Functions - they are there, side effects are mostly optional.
    - Recursion - widely supported in all relevant languages.
    - What if we have a cycle inside in a function, is this a problem?
    - Functions as first class citizens - more of a problem, but most languages covers this.
  - Immutable data types - a choice of a programmer.
  - A strong type system to capture errors.
- Notable items on a nice to have list
  - Algebraic data types - rare, in OOP some solution can be inheritance.
  - Higher-kinded polymorphism - bigger issue, partially can be solved by generic data types.
- In C# we have: delegates, lambda expressions, pattern matching, tuples...

Functions with No Side Effects (1)

- What are side effects, how do i recognise them?

```
public double Add(double a, double b) {
    return a + b;
}
public double Add2(double a, double b) {
    try {
        Console.WriteLine($"a={a}, b={b}");
    } catch (Exception ex) { }
    return a + b;
}
public int Divide(int a, int b) {
    return a / b;
}
```

Functions with No Side Effects (2)

- How can I avoid them?

```
public int? Divide2(int a, int b) {
    if (b == 0)
        return null;
    return a / b;
}

public int Divide3(int a, NonZeroInteger b) {
    return a / b.Number;
}
```

```
public class NonZeroInteger {
    public int Number { get; }

    public NonZeroInteger(int number) {
        Number = number;
        if (number == 0)
            throw new ArgumentException();
    }
}
```

Immutable array

- Sometimes they are called persistent data structures.
- `https://en.wikipedia.org/wiki/Persistent_data_structure`
- `https://en.wikipedia.org/wiki/Persistent_array`

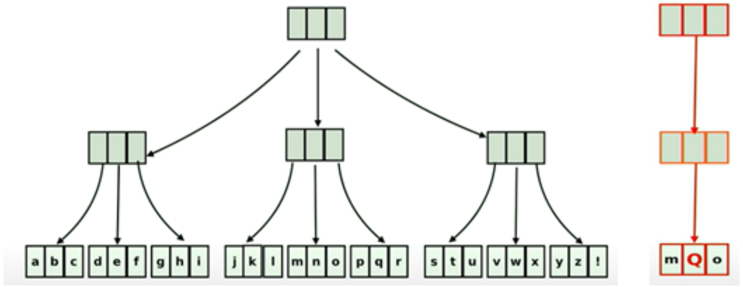


Figure: An idea how to implement immutable array.

Mutable data types

```
public class Stack<T>
{
    private List<T> data;

    public Stack()
    {
        data = new List<T>();
    }

    public void Push(T item)
    {
        data.Add(item);
    }

    public T Pop()
    {
        T item = data[data.Count-1];
        data.RemoveAt(data.Count-1);
        return item;
    }

    static void Main(string[] args)
    {
        Stack<int> stack =
            new Stack<int>();
        stack.Push(1);
        stack.Push(2);
        var x = stack.Pop();
        Console.WriteLine(x);
    }
}
```

Immutable solution in C# (1)

```
public class NewStack<T>
{
    private T Data { get; init; }

    private NewStack<T> Next { get; init; }
        private NewStack()  { }

    static public NewStack<T> Empty() => null;
    static public bool IsEmpty(NewStack<T> stack) => stack == null;

    static public NewStack<T> Push(NewStack<T> stack, T item) =>
    new NewStack<T> { Data = item, Next = stack };

    static public (T Item, NewStack<T> Stack) Pop(NewStack<T> stack) =>
        (IsEmpty(stack)) ? throw new Exception("Empty stack.") : (stack.Data, stack.Next);
}
```

Immutable solution in C# (2)

```
static void Main(string[] args)
{
    NewStack<int> newStack = NewStack<int>.Empty();
    newStack = NewStack<int>.Push(newStack, 1);
    newStack = NewStack<int>.Push(newStack, 2);

    (x,newStack) = NewStack<int>.Pop(newStack);

    Console.WriteLine(x);
}
```

- Immutable data types are studied problem, plenty of possibilities.
- Common in API of many languages (C#: string, DateTime, `https://www.nuget.org/packages/System.Collections.Immutable/`).

Composition of functions

- To solve side effects, we have used monads → Can we implement the same ideas in C#?
- Lets start with something simple, function composition.

```
public static Func<A, C> After<A, B, C>(this Func<B, C> f, Func<A, B> g)
        => value => f(g(value));

public static Func<A,C> Composition<A, B, C>(Func<B, C> f, Func<A, B> g)
        => value => f(g(value));

Func<string, int> parse = int.Parse; // string -> int
Func<int, int> abs = Math.Abs; // int -> int

Func<string, int> composition1 = abs.After(parse);
Func<string, int> composition2 = Composition(abs, parse);
```

Functor (1)

- We have no algebraic data types → we can use sub-typing instead.

```
public abstract class Maybe<A> {}

public class Just<T> : Maybe<T> { public T Value { get; init; } }
public class Nothing<T> : Maybe<T> {}
```

- What we are lacking is Haskell *kind* (`Functor :: (* -> *) -> Constraint`).

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

- We need something like type bellow → it wont compile → we need to cheat it somehow.

```
public interface IFunctor<TFunctor<A>> where TFunctor<>: IFunctor<TFunctor>
{
  static abstract TFunctor<B> fmap<B>(Func<A,B> f,  TFunctor<A> a);
}
```

Functor (2)

- What we had before? How `fmap` was used?

```
*Main> (+1) `fmap` ((*2) `fmap` ((+3) `fmap` (Just 1)))
Just 9
*Main> (+1) `fmap` (*2) `fmap` (+3) `fmap` (Just 1)
Just 9
```

- We use in fact:

```
class Functor (Maybe a) where
    fmap f (Just x) = Just (f x)       instance Functor ((->) r) where
    fmap _ Nothing = Nothing               fmap = (.)
```

- For OOP: `(<&>)::Functor f =>f a->(a -> b)->f b --Data.Functor`

```
ghci> (Just 1) <&> (+1) <&> (*2)
```

Functor (3)

- We do not have the same abilities (no higher-kinded polymorphism) in C# → We can cheat a little. → We will use an interface for functor.

```
public interface IFunctor<A>
{
  IFunctor<B> fmap<B>(Func<A, B> f);
}
```

- Now we need to implement the defined interface in type $Maybe$

```
public abstract class Maybe<A> : IFunctor<A> {
  public abstract IFunctor<B> fmap<B>(Func<A,B> f);
}
public class Just<A> : Maybe<A> {
  public T Value { get; init; }
  public override IFunctor<B> fmap<B>(Func<A, B> f) =>
    new Just<B>() { Value = f(Value) };
}
public class Nothing<A> : Maybe<A> {
  public override IFunctor<B> Bind<B>(Func<A, B> f) => new Nothing<B>();
}
```

Functor (4)

- It can be used as:

```
IFunctor<int> result =
  new Just<int>() { Value = 1 }
  .fmap(x => x.ToString())
  .fmap(x => int.Parse(x));
```

- If we want to preserve original ordering:

```
public static IFunctor<B> fmap<A,B>(this Func<A,B> f, IFunctor<A> a) => a.fmap(f);
public static Func<R,B> fmap<R,A,B>(this Func<A,B> f, Func<R,A> g) => x => f(g(x));
```

```
((Func<int, int>)(x => x + 1))
.fmap((Func<int, int>)(x => x * 2))
.fmap(new Just<int>(){ Value = 1 });
```

```
var result =
  ((Func<int, int>)(x => x + 1))
  .fmap(new Just<int> { Value = 1 });
```

- Note, in Haskell, if we started with `Maybe` result was also `Maybe`. Now it can be different.

Applicative in C# (1)

- We can use the same approach for `Applicative`.
  - Again we use: `(<**>) :: Applicative f => f a -> f (a -> b) -> f b`

```
ghci> (Just 1) <**> (Just (+1)) <**> (Just (*2))
Just 4
ghci> (Just (+)) <*> (Just 1) <*> (Just 2)
Just 3
ghci> (+) <$> (Just 1) <*> (Just 2)
Just 3
```

```
interface IApplicative<A>: IFunctor<A>
{
  IApplicative<A> Wrap(A t);
  IApplicative<B> Apply<B>( IApplicative<Func<A, B>> f);
}
```

Applicative in C# (2)

- Our type $Maybe$ needs to implement new interface IApplicative now.

```
public abstract class Maybe<A> : IApplicative<A> {
  public abstract IApplicative<B> Apply<B>(IApplicative<Func<A, B>> f);
  public IApplicative<A> Wrap(A b) => new Just<A>() { Value = b };
}
public class Just<T> : Maybe<T> {
  public override IApplicative<B> Apply<B>(IApplicative<Func<T, B>> f) => f switch
        { Just<Func<T, B>> justF => new Just<B>() { Value = justF.Value(Value) },
                                _ => new Nothing<B>() };
}
public class Nothing<T> : Maybe<T> {
  public override IApplicative<B> Apply<B>(IApplicative<Func<T, B>> f) => new Nothing<B>(
}
```

- We can use it:

```
new Just<int>() { Value = 1 }
   .Apply(new Just<Func<int, int>>() { Value = x => x + 1 })
   .Apply(new Just<Func<int, int>>() { Value = x *2 });
```

Applicative in C# (3)

- We can use extension method for *Apply*:

```
public static IApplicative<B> Apply<A, B>(
  this IApplicative<Func<A, B>> f, IApplicative<A> a) => a.Apply(f);
```

- Now, we use also the same notation as for <∗>.

```
Func<int, Func<int, int>> plus = x => y => x + y;
result = new Just<Func<int, Func<int, int>>>() { Value = plus }
       .Apply(new Just<int>() { Value = 1})
       .Apply(new Just<int>() { Value = 2 });
```

- Or combine both interfaces.

```
result = ((Maybe<Func<int,int>>) plus
       .fmap(new Just<int>() { Value = 1 }))
       .Apply(new Just<int>() { Value = 2 });
```

Monads in C# (1)

- Finally, we will define: `IMonad`.

```
interface IMonad<A> : IApplicative<A>
{
  IMonad<B> Return<B>(B value);
  IMonad<B> Bind<B>(Func<A, IMonad<B>> f);
}
```

- Again, we can extend our type *Maybe*

```
public abstract class Maybe<A> : IMonad<A> {
  public IMonad<B> Return<B>(B value) => new Just<B>() { Value = value };
  abstract public IMonad<B> Bind<B>(Func<A, IMonad<B>> f);
}
public class Just<T> : Maybe<T> {
  public T Value { get; init; }
  public override IMonad<B> Bind<B>(Func<T, IMonad<B>> f) => f(Value);
}
public class Nothing<T> : Maybe<T> {
  public override IMonad<B> Bind<B>(Func<T, IMonad<B>> f) => new Nothing<B>();
}
```

Monads in C# (2)

- Now, we can bind actions:

```
var result = new Just<int>() { Value = 1 }
             .Bind(x=>new Just<int>{Value = x+1})
             .Bind(x=>new Just<string>(){Value = ""+x});
```

- Moreover, we can observe, that $Bind$ and $Return$ is enough to define $Apply, Wrap$ and $fmap$

```
public IFunctor<B> fmap<B>(Func<A, B> f) => Bind(x => Return(f(x)));

public IApplicative<A> Wrap(A value) => Return(value);

public IApplicative<B> Apply<B>(IApplicative<Func<A, B>> f) =>
  ((IMonad<Func<A, B>>)f).Bind(f=>Bind(x => Return(f(x))));
```

Simplifying our solution (1)

- What if we want to implement monad just for one type $\rightarrow$ we can use extension methods.
  - For clarity, we will skip some details like methods like $Return$ and $Wrap$
  - Moreover, now the types describes *better* what we want.

```
public static Maybe<B> fmap<A, B>(this Maybe<A> x, Func<A, B> f) => x switch
  {
    Nothing<A> => new Nothing<B>(),
    Just<A> v  => new Just<B>() { Value = f(v.Value) },
  };
```

- In the same way, we can implement $Applicative$ and $Bind$.

```
 public static Maybe<B> Apply<A, B>(this Maybe<A> x, Maybe<Func<A, B>> f)=> f switch
  {
    Nothing<Func<A, B>>    => new Nothing<B>(),
    Just<Func<A, B>> justF => x.fmap(justF.Value),
  };
```

Simplifying our solution (2)

- But all we really need is *Bind*.

```
public static Maybe<B> Bind<A, B>(this Maybe<A> x, Func<A, Maybe<B>> f) => x switch
  {
    Nothing<A>     => new Nothing<B>(),
    Just<A> value => f(value.Value),
  };
```

- With these extension methods we can do all examples from previous slides.

```
var result = new Just<int>() { Value = 1 }
           .Bind(x=>new Just<int>{Value = x+1})
           .Bind(x=>new Just<string>(){Value = ""+x});
```

IEnumerable monad (1)

- Important monad is *a list*. In C#, it can be even more general type `IEnumerable`.

```
public static IEnumerable<A> Return<A>(this A a)
{
  yield return a;
}

public static IEnumerable<B> Bind<A,B>(this IEnumerable<A> a, Func<A,IEnumerable<B>> f)
{
    foreach (var itemInA in a)
    {
        foreach (var itemInPartialResult in f(itemInA))
        {
            yield return itemInPartialResult;
        }
    }
}
```

IEnumerable monad (2)

- With $Bind$ for `IEnumerable`, we can use list similarly to Haskell.

```
var list = new List<int> { 1, 2, 3 }
         .Bind2(x => new List<char> { 'a', 'b' }
         .Bind2(ch => new List<(int, char)> { (x, ch) }));
Console.WriteLine(string.Join(", ", list));
```

```
(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)
```

- Now, if you are C# expert, you can say: Wait, there is very similar method to our `Bind` $\rightarrow$ `SelectMany`.

```
public static IEnumerable<TResult> SelectMany<TSource,TCollection,TResult> (
  this IEnumerable<TSource> source,
  Func<TSource, IEnumerable<TCollection>> collectionSelector,
  Func<TSource,TCollection,TResult> resultSelector // returnSelector is optional
);
```

Query syntax (1)

- The method `SelectMany` can be implemented as:

```
public static IEnumerable<C> SelectMany<A, B, C>(
  this IEnumerable<A> a,
  Func<A, IEnumerable<B>> f,
  Func<A, B, C> selector)
{
  foreach (var itemInA in a)
  {
    foreach (var itemInPartialResult in f(itemInA))
    {
      yield return selector(itemInA, itemInPartialResult);
    }
  }
}
```

- We can use it to implement our $Bind$ method.

```
public static IEnumerable<B> Bind<A,B>(this IEnumerable<A> a, Func<A,IEnumerable<B>> f)
  => a.SelectMany(f);
```

Query syntax (2)

- Why it is interesting?
  1. `IEnumerable` is already monad.
  2. Brian Beckman: LINQ is monad. It is very carefully designed by Erik Meijer so that it is monad.
  3. We can use query syntax now.

```
new List<int> { 1, 2, 3 }                      [1,2,3]
 .Bind2(x => new List<char> { 'a', 'b' }        >>= \n -> ['a','b']
 .Bind2(ch=>new List<(int,char)> {(x,ch)}));    >>= \ch -> [(n,ch)]

from x in new List<int> { 1, 2, 3 }            do x <- [1,2,3]
from y in new List<char> { 'a', 'b' }             y <- ['a','b']
select (x, y);                                    return (x, y)
```

- So, we have very similar syntax. Can it be used also for our original type `Maybe`? → Sure, we just need to provide method: `SelectMany`.

Query syntax (3)

- Let's extend our type Maybe:

```
public Maybe<B> SelectMany<B>(Func<A, Maybe<B>> f) => (Maybe<B>)Bind(f);

public Maybe<C> SelectMany<B, C>(Func<A, Maybe<B>> f, Func<A, B, C> resultSelector)
{
  Maybe<B> value = (Maybe<B>)this.Bind(f);
  return value switch
    {
      Just<B> result when this is Just<A> =>
        new Just<C> { Value = resultSelector(((Just<A>)this).Value, result.Value) },
      _ => new Nothing<C>()
    };
}
```

- Or alternatively:

```
 public Maybe<C> SelectMany<B, C>(Func<A, Maybe<B>> f, Func<A, B, C> resultSelector)
   => (Maybe<C>)Bind(x => f(x).Bind(y => Return(resultSelector(x, y))));
```

Query syntax (4)

- Defining `SelectMany` allows to use query syntax for this type.

```
var test = from x in new Just<int> { Value = 1 }
           from y in new Just<int> { Value = 1 }
           let z = x + y;
           select z;
```

- We can simplify the whole problem to: If we want monadic behaviour, all we need is to provide method $SelectMany$.

Func<A> monad (1)

- First, lets define the *bind* for a simple function: `Func<A>`.

```
public static Func<A> Return<A>(A value) => () => value;
public static Func<C> SelectMany<A, B, C>(
  this Func<A> source,
  Func<A, Func<B>> selector,
  Func<A, B, C> resultSelector) => () =>
    {
      A value = source();
      return resultSelector(value, selector(value)());
    };
public static Func<B> Select<A, B>(
  this Func<A> source,
  Func<A, Func<B>> selector) =>
    source.SelectMany(
      selector,
      (_, result) => result);
```

Func<A> monad (2)

- A test for our newly created monad:

```
Func<string> query = from name in (Func<string>)Console.ReadLine
                     from fileContent in (Func<string>)(()=>File.ReadAllText(name))
                     select fileContent;
Console.WriteLine("Query created.");
string result = query(); // Execute query.
Console.WriteLine(result);
```

- $Func$ is just a delegate:

```
public delegate TResult Func<in T,out TResult>(T arg);
```

- The same approach can be used for $state$ monad (and similar monads):

```
public delegate (TState State, T Value) State<TState, T>(TState state);
```

Task<A> monad

- Our monadic type does not need to be a *function*, we can use for example class `Task`.

```
public static async Task<C> SelectMany<A, B, C>(
  this Task<A> source,
  Func<A, Task<B>> selector,
  Func<A, B, C> resultSelector) =>
    resultSelector(await source, await selector(await source));
```

- Usage:

```
Task<string> query =
  from response in new HttpClient().GetAsync(@"https://idnes.cz/")
  from stream in response.Content.ReadAsStreamAsync()
  from text in new StreamReader(stream).ReadToEndAsync()
  select text; // Define and execute query.
Console.WriteLine("Query created");
string result = await query; // Query result.
Console.WriteLine(result);
```

Conclusion

- Monad are already embedded into the language design. $\rightarrow$ We can start using them to $bind\ actions$.
- We do not have the same language capabilities as in Haskell, but most issues can be overcome and C# like a language provides pretty strong support for functional programming.

Thank you for your attention

Marek Běhálek

VSB – Technical University of Ostrava

marek.behalek@vsb.cz

October 21, 2022

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE