FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Harun Ćerim

Extending C# with a Library of Functional Programming Concepts

Department of Distributed and Dependable Systems

Supervisor of the master thesis: doc. RNDr. Pavel Parízek, Ph.D.

Study programme: Computer Science

Specialization: Software and Data Engineering

Prague 2020

This thesis is a result of the hard work and support of many people that have been by my side throughout this journey.

Living in Prague and studying at Charles University has been a wonderful experience where I met a lot of smart, interesting, and good people that changed my perspective for better on many things related to both life and school.

I would like to thank my family for their love and support. I would also like to thank my friends for being by my side throughout this journey.

Lastly, I would like to thank my supervisor doc. RNDr. Pavel Parízek, Ph.D., and express my gratitude for his guidance and patience over the last few months.

This thesis is dedicated to my brother, Faris.

Title: Extending C# with a Library of Functional Programming Concepts

Author: Harun Ćerim

Department: Department of Distributed and Dependable Systems

Supervisor of the master thesis: doc. RNDr. Pavel Parízek, Ph.D., Department of Distributed and Dependable Systems

Abstract: The main goal of this thesis was to implement a functional programming (FP) library named Funk that extends C# with support for concepts present in functional programming languages, such as F# and Scala. Funk utilizes many functional programming concepts, including immutability, pattern matching, and various types of monads, together with stronger typing. Introduction of these concepts into C# helps in avoiding many runtime errors and boilerplate code, and it also lets developers write C# code in a declarative rather than in an imperative way, making the day-to-day software development easier and less error-prone. Additionally, the thesis analyzes and compares Funk with existing functional programming libraries such as Language-ext and FuncSharp. Finally, it analyzes the new features of C# 8, which include nullable reference types and pattern matching and compares them with the functionalities of the Funk library.

Keywords: csharp, functional programming, category theory, monads, asynchronous programming

Table of Contents

1. Introduction

The goal of this project and the main motivation behind Funk is to make C# less error-prone and more powerful programming language to work with. As C# is primarily an object-oriented programming (OOP) language, many concepts from that paradigm make it difficult for developers to trust their codebase which leads to defensive programming and focusing on avoiding unexpected errors rather than on concrete problems. Developers tend to write code either expecting that everything will go as planned or that everything can go wrong. The latter is not incorrect but the way it is solved is tedious and repetitive. Repetitive is boring and error-prone and anything repetitive can be abstracted away which is the main goal of any modular (component-oriented) programming language.

```
/// <summary>
/// Returns latest stock price.
/// </summary>
/// <exception cref="ArgumentException"></exception>
/// <exception cref="InvalidOperationException"></exception>
public async Task<StockPrice> GetLatest(string companySymbol)
{
    if (string.IsNullOrWhiteSpace(companySymbol))
        throw new ArgumentException("Company symbol not provided!");
    if (Symbols.All(s => s != companySymbol))
        throw new InvalidOperationException("Company symbol not supported!");
    try
    {
        var response = await Client.GetAsync(url + companySymbol );
        var content = await response.Content.ReadAsStringAsync();
        return JsonConvert.DeserializeObject<StockPrice>(content);
    }
    catch (ArgumentNullException e)
    {
        throw new InvalidOperationException("Url not provided.");
    }
    catch (HttpRequestException e)
    {
        throw new InvalidOperationException("Request failed.");
    }
    catch (JsonException e)
    {
        throw new InvalidOperationException("Response not in a correct format.");
    }
}
```

Snippet 1

From Snippet 1 we can see the usual way of writing *safe* code in C#. This function performs a simple operation to get the response from the external service regarding the latest stock price for the specified company provided as an argument. It only takes 3 lines of code including the deserialization process to express that operation. However, this function spreads to more than 20 lines because of possible failures that may occur. Some of those failures are business-related and others are related to the exceptions that may occur in other functions that this function calls. This is tedious and also quite possibly wrong as not all possible failures might be covered. Additionally, it completely drives us from the problem we are trying to solve to dealing with possible failures and finding the best possible way to handle them. Moreover, the caller of this function will also need to look into the implementation of the function or follow the instructions in the comments to understand how to handle possible failures. Since none of this is forced by the compiler, usually bad things (unexpected bugs) occur.

The major contribution of this thesis is the exploration of functional programming (FP) concepts and patterns in the context of their implementation. The thesis contains an analysis of the most useful concepts including various types of monads and different types of pattern-matching techniques that are not fully present in C#. The results of the analysis show the advantages of thinking and writing code functionally and point out the flaws that object-oriented (OO) thinking brought to C#. Funk is a library that simplifies the developer's work and increases confidence in the codebase. Additionally, we have analyzed the previous work done by the community to address the kinds of issues that functional programming is trying to solve and compare it to the features implemented in Funk. Finally, through demo programs that use Funk, we show how easily and fluently one can write code with increased expressiveness and safety regarding unexpected bugs due to the incorrect data representation.

The structure of this thesis is as follows.

The second chapter opens with the introduction of functional programming. It talks about the concepts and patterns, pointing out the advantages of the paradigm and disadvantages that can be addressed.

The third chapter talks about C# as a multi-paradigm programming language. It points out the functional programming-related capabilities of the language and how they can be improved. It concludes with the problems that C# and the .NET framework teams introduced into C# and .NET over the years.

The fourth chapter describes Funk and the concepts that are implemented. It talks about available monads and monadic structures, functional prelude, extension methods for different types, and exception types.

The fifth chapter describes the tools used to develop the Funk library as well as the technical challenges that we had to face during the development.

The sixth chapter describes the case study and the evaluation of Funk.

The seventh chapter analyses other functional programming libraries and compares them with Funk. It also analyzes improved functional programming concepts in C# 8.

The last chapter mentions the current adoption of Funk in the .NET community, features planned for the future, and concludes this thesis with final remarks.

2. Functional Programming

In this chapter, we will introduce the functional programming (FP) paradigm together with the most important concepts and patterns and the benefits that it brings.

2.1. Overview

Functional programming is a declarative programming paradigm where functions are first-class values. Those functions are usually *pure*, which means that besides always returning the same result for the same input, they never have any side effects. Those side effects come in a form of state mutation of non-local variables or any other mutation of the global state.

Functional programming insists on *honest functions*. Honest functions represent functions that tell you by their interface what kind of computation do they perform, which parameters do they accept, and what is their return type. Honest functions never have unexpected behaviors (the ones not stated in the interface) that can be caused by inner or outside factors.

Functional programming gives the developer *power*, *safety*, and *clarity*. Every line of code is an asset but a liability as well. Functional programming helps you reduce the code size which results in keeping the assets and getting rid of the liability (Bounanno, 2018). Stronger typing with guaranteed immutability ensures that the program will perform without breaking or having unexpected results in a concurrent environment. Additionally, having less code that is safe ensures easier maintainability and better readability of the code.

In mathematics, a function is simply a map between elements from one set to elements of another set. We call these sets a *domain* and a *codomain*.

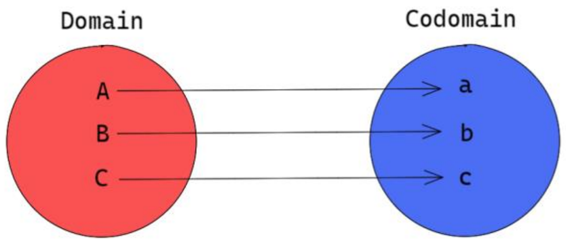


Figure 1: Function

The types for the domain and codomain represent the function signature or interface. That function represented as an arrow in Figure 1 given an element from the domain returns the element from the codomain.

2.2. Function purity

Pure functional programming languages don't allow state mutation to ensure one of the principles of pure functions. The other principle, *referential transparency*, stating that the result of the computation should always be the same for the same input, can be ensured by any language as it is more of a logical decision. Referential transparency is not widely adopted even in some functional programming languages. However, by the definition, for the function to be pure, it must satisfy both of these rules.

In object-oriented programming, function purity is not that common as it is in functional programming. Usually, object-oriented programming languages are used in commercial and enterprise solutions where communication with the *outside world* is a must for the program to perform some useful operation. The outside world represents an external resource such as file system, database, external Application Programming Interface (API), etc. Communication with these resources means working with the data that might change (mutate) either by that operation or during some other computation. One simple example is a function in Snippet 2 that returns the current time.

```
public static DateTime GetCurrentTime()
{
    // return current time.
}
```

Snippet 2

No matter how simple the `GetCurrentTime` function can be, it is still not pure as at any given time it is going to return a different result.

The more functional approach in object-oriented programming is to have the *impure-pure-impure* function relationship. It is also called an impure-pure-impure *sandwich*. The motivation behind that concept is to reduce the footprint of impure functions and the amount of impure code in the codebase. The communication with the external resource is impure but any computation based on results can be extracted into a separate function that can be pure. The impure function would call the pure function to perform that specific computation and continue with the rest of the process which is again most probably impure.

```
public async Task<UserViewModel> ProcessUsersRequest(UpdateUserRequest request)
{
    var user = await db.Users.Get(request.Id);
    var newUser = user.Update(request);
    var response = await db.Users.Update(newUser);
    return response.ToViewModel();
}
```

Snippet 3

From Snippet 3 we can see a function that calls four other functions. The first three functions represent an impure-pure-impure sandwich. The first function is impure as it performs an I/O operation and tries to find a user with the specified id. The second function is pure as it only creates an updated user that is going to be stored in the database by calling again the impure function. By separating pure from impure operations we are increasing the amount of pure code where we make smaller units more testable. An important thing to note is that any function that calls impure function is itself impure. So overall `ProcessUserRequest` is an impure function but at least we can be certain for some parts of it to be completely pure.

Function purity can significantly improve the codebase to minimize the number of unexpected behaviors, especially in concurrent environments.

2.3. Function signature and honesty

Similarly, as function purity, function honesty can significantly improve code readability and reduce the number of unexpected behaviors.

The function interface or signature is responsible for telling the caller what the function does, what parameters it accepts, and what it is going to return. Its name is also a part of the interface and it helps describe the behavior without looking into the implementation.

In object-oriented programming, developers are used to writing dishonest functions where in some cases the function either returns the null value or it does not return (throws an exception). We can have a simple function that returns the user from the database based on the user id.

```
public User GetUser(Guid id)
{
     // communication with the db.
     // return result
}
```

Snippet 4

Now, this function in Snippet 4 is obviously not pure, but it is also not honest, and we should always tend to write honest code that tells you exactly what is going to happen. We can see that the function accepts the `Guid` type of id and based on the value it retrieves the `User` object from the database. The problem with this function is that if there is no user for the given id, the function will fail to retrieve the user. The return value will be either null or the exception will be thrown. Both are incorrect solutions and introduce confusion. It is forcing the caller to look into the implementation and to handle the call properly. This way of writing code is introducing something called *defensive programming*. In defensive programming, we usually have the case where every function is either wrapped in a *try-catch* block where the exception needs to be handled or there are null checks in front of every computation to avoid *Null-Reference Exception* (NRE).

```
public static string GetUserName(Guid id)
{
    try
    {
        var user = GetUser(id);
        return user == null ? "User not found" : user.Name;
    }
    catch (UserNotFoundException e)
    {
        return "Exception thrown";
    }
}
```

Snippet 5

The example function in Snippet 5 performs a very simple operation and yet it requires more than 10 lines of code. Even if we ignore the fact that this program could be written in one line there are still many things wrong here including the fact that unless you look into the implementation you are not aware from the signature of the method that the method might not return the valid username. In case the user is not found, or the exception is thrown the primitive return type that was meant to indicate success now contains a reason for failure in the same format. The response could, however, be a tuple of success and failure where in case of success, the failure would be null. However, that approach can also introduce bugs because of incorrect handling as the caller would not be forced by the compiler to handle the response properly. The program usually has hundreds of functions like this one which means a lot of repetitive work regarding null checking and exception handling. This increases the codebase and makes the code less readable and completely vulnerable to bugs. Additionally, it significantly lowers the level of abstraction the object-oriented programming should bring in the first place.

2.4. Concepts

To address the kinds of issues that developers face in the object-oriented world, we can look at the concepts and patterns that functional programming introduces to support better expressiveness and declarative style of writing code.

Functional programming is all about avoiding state mutation, so as mentioned earlier in purely functional programming languages everything defined (objects, variables, fields, etc.) is immutable. In most OOP languages everything is mutable by default. To make a field immutable in C# it must be marked as *readonly*. Variables cannot be read-only.

Functional programming also introduces a concept called a *monad*. The concept of a monad is better known from a Category Theory as a *monoid in the category of endofunctors* (Lane, 1971). However, the definition is more understandable for someone who has a strong mathematical background and not suitable for developers who are trying to understand the concept. The simplest way to explain a monad is to say that it amplifies the specific type by additional features specific for that monad. So, we can say that monads are type amplifiers.

```
public interface IMonad<T>
{
    IMonad<T> Return(T item);
    IMonad<R> Bind<R>(Func<T, IMonad<R>> selector);
}
```

Snippet 6

From Snippet 6 we can see two methods defined for the `IMonad` interface. The type that is going to implement it and to be generic as the interface will represent a *type* *constructor*. It means that it is not a concrete type until a specific type is provided during its construction. In pure functional programming languages like Haskell, monadic structures are a natural way of representing data. For a type to be considered a monad it needs to satisfy two things as described in the interface. The first one is the *return* function which can be also called wrapping or lifting function. It is responsible for lifting any type to a given monad. The second one is a *flattening* function that results in having one type of a given monad transformed into another type of that same monad based on its value. It can be also called a binding function. These two requirements need to be fulfilled to call a specific structure a monadic structure. Additionally, some languages like Scala require also *mapping* function that works similarly as flattening function but instead of having already monadic structure as a result of the transformation specified by the selector, mapping function lifts the result type automatically. The mapping function usually uses the flattening function internally. Unwrapping function is not a requirement of a monad.

Pattern-matching is also a concept present in functional programming. It is a very useful feature where a specific value is matched on a sequence of given values. In C#, *switch* statements can be considered a form of pattern matching but they are not nearly as powerful as the ones available in languages like Scala or F#. C# 8 has switch statements improved and now they resemble the ones in other functional programming languages. However, it is the newest release fully compatible only with the latest .NET (Core 3.0, Standard 2.1) and cannot be used in many situations (existing applications on older frameworks or legacy systems).

Besides the concepts mentioned, functional programming offers many more interesting and useful patterns that can be used to simplify and make the code more robust and powerful. These patterns include *functors, applicatives, partial application*, the concept of *currying*, etc. They will be described in detail in the following chapters.

2.5. Benefits

As mentioned earlier, functional programming gives the developer power, safety, and clarity. Functional programming allows the developer to replace statements with expressions. It means switching from imperative style to declarative style of writing code as illustrated in Figure 2.

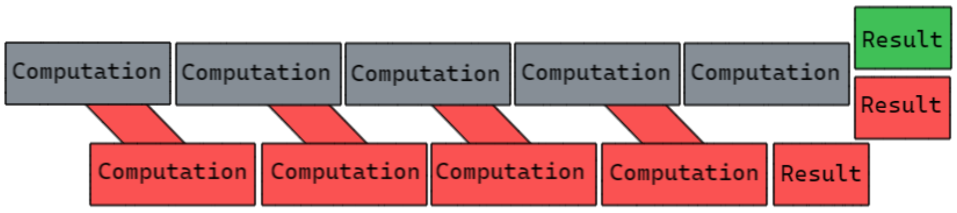


Figure 2: Railway-oriented programming

With monadic structures and pattern-matching that were mentioned earlier, it brings *railway-oriented programming* (ROP) (Wlaschin, 2014) style into the code. It helps developers more naturally express themselves when trying to avoid a *happy path* without lots of boilerplate code and defensive style. The railway analogy is correct as it properly describes the way expressions should work. As shown in Figure 2, the initial computation might indicate success or failure. To perform the second computation, the initial computation must be in a successful state or the *right path*. If it is in the *wrong path* the next computation will not be executed and the whole computation will end up as a failure. Only if all the computations succeed, the result will be in a success state. As it is visible from the figure, once the wrong path is entered, there is no going back to the right path.

With immutability, it ensures that the data is not going to mutate once created which builds trust, especially when working with sequences and abstracted code. Functional programming brings more benefits than drawbacks, but it is worth mentioning them as well. Drawbacks are usually linked to performance due to immutability. Instead of updates, new objects are created that might impact the memory consumption. However, when designed properly, functional programs can perform as well as object-oriented programs and sometimes even better. *Functional data structures* have been invented to provide a way of working with data safely ensuring immutability but taking into account performance by making immutable objects reusable without the need of constructing new objects every time they are required. In times where hardware is not an issue, functional programming concepts are worth implementing even when there is a small performance impact. It is better to have a more reliable program than the best performing program that often breaks.

3. Functional C#

This chapter talks about the functional programming features of the C# that make it possible to extend with additional features and concepts that are currently missing from C# and the .NET.

3.1. Functions as first-class values

C# as a multi-paradigm but primarily an object-oriented programming language has properties of a functional programming language as well. It had functional programming features from the earliest versions when the `Delegate` type was introduced. It is a type that makes functions first-class values in the language and makes everything else that is functional, including *Language Integrated Query* (LINQ), possible. In later versions of C#, working with the raw `Delegate` type was replaced with `Func` and `Action` delegates that represent a function that returns and a void function respectively. *Lambda expressions* were also introduced which made syntax even more fluent and enabled easier delegate representation.

```
static Func<int, int, int> Add1 => delegate(int first, int second)
{
    return first + second;
};
static Func<int, int, int> Add2 => (first, second) => first + second;
```

Snippet 7

As it is visible from Snippet 7 the first `Add` function uses the older way of representing delegates by using the keyword `delegate` along with the *anonymous method*. Anonymous methods are just inline statements used in place of delegates to create an anonymous function. They can be completely omitted using lambda expressions as lambdas also create anonymous functions. The second `Add` function is using lambda expression which makes it much clearer and straightforward.

In C#, there are different types of functions: *methods*, *delegates*, *lambda expressions*, *properties*, and *events*. Methods are types of functions that are common in the object-oriented programming world. They can implement an interface, can be overloaded, etc.

```
public interface ICalculator
{
	int Add(int first, int second);
}

public class Calculator : ICalculator
{
	public int Add(int first, int second)
	{
		return first + second;
	}
}
```

Snippet 8

Delegates represent function objects or more accurately *type-safe function pointers*. Snippet 8 shows the usual way of defining an interface with its implementation. Delegate, as viewed from the object-oriented programming standpoint, is similar to an interface and its implementation resembles the method that implements that interface.

```
public delegate int Add(int first, int second);

Add add = (first, second) => first + second;
```

Snippet 9

In Snippet 9, we can see how the delegate is defined similarly as the interface in Snippet 8. The contract is therefore defined in the interface and the delegate, and the responsibility of the class in Snippet 8 and the function object in Snippet 9 is to implement it.

However, delegates in this form are rarely used today and are replaced with a much more straight-forward types: `Func` and `Action` with various arities.

```
Func<int, int, int> add = (first, second) => first + second;
```

Snippet 10

The code from Snippet 9 can be replaced with the one-liner from Snippet 10 in C# 3.0 and higher. It is much more straightforward and readable as shown in the comparison in Snippet 7, and therefore it is the preferred way of defining delegates.

Lambda expressions are represented with an arrow (=>) sign. It is sometimes called just *lambda* and it represents an inline function as already explained. However, it is also possible to define a body with the lambda expression using curly braces (**{}**). With each lambda expression, a *closure* is captured and it represents that lambda along with the context in which it is declared.

```
Func<User, string> GetGreeting = user =>
{
    var greet = GetGreeting(user.Localization);
    return $"{greet} {user.Name} {user.Surname}";
};
```

Snippet 11

As shown in Snippet 11, we can define variables inside of the body of the lambda expression. Closure represents this delegate with its enclosing environment including the User argument when invoking this function or any other passing variable along with everything that is defined in the body.

3.2. Thinking functionally in C#

Even though C# is a functional programming language to some degree, many developers don't use those features that it offers and are more comfortable using it in a plain object-oriented way. C# was created as Microsoft's response to Java and many developers that use C# view it as a powerful but still just an object-oriented programming language. Java doesn't have features that C# offers from a functional programming standpoint and therefore Scala was created to address those missing features. C# is not nearly as powerful functional programming language as Scala but as the programming language with the great support of the community and the .NET ecosystem, it is usually a language of choice for many organizations. Overall, thinking functionally in C# is possible and worth discovering for the benefit of any organization.

As described in Section 3.1., C# has functions as first-class values. It means one can create an object that points to a specific function. It is a very powerful feature that enables developers to think of functions as objects and not just the functions as class members. Another great feature of the language is the ability to create *extension methods*. Those are functions available as `static` methods that are accessed as class functions rather than instance functions but make an impression that are available in the object directly. It is achieved by the keyword `this`.

```
public static class IntExtensions
{
	public static int Add(this int first, int second) => first + second;
}

var result = 1.Add(2); // 3
```

Snippet 12

As shown in Snippet 12, the second line shows that the extension method `Add` can be accessed directly without the need for further qualification. However, it is also possible to call this method with class qualification as shown in Snippet 13.

```
var result = IntExtensions.Add(1, 2); // 3
```

Snippet 13

By comparing the two function calls, we can see that the extension methods are powerful because they can be called without class qualification which adds to the code writing fluency. This fluency is one of the features of functional programming languages and is what makes *Fluent API* possible in C#.

```
public static class ObjectExt
{
    public static R Match<T, R>(
      this T obj,
      T case1, Func<T, R> selector1,
      T case2, Func<T, R> selector2,
      T case3, Func<T, R> selector3
    )
    {
      if (obj.Equals(case1))
      {
        return selector1(obj);
      }
      if (obj.Equals(case2))
      {
        return selector2(obj);
      }
      return selector3(obj);
    }
}
```

Snippet 14

The extension method from Snippet 14 is a naive pattern-matching function for any type. It matches the value of the provided object with the ones specified in expressions and returns the result of the selector that is related to the correct case.

```
var dummyResponse = Console.ReadLine().Match(
    "a", a => $"{a} chosen",
    "b", b => $"{b} chosen",
    "c", c => $"{c} chosen"
);
```

Snippet 15

We can call `Match` function with any type and in Snippet 15 the `String` is used. It matches the value of the input and returns a response specified by the selector. The input might differ from the three cases provided as the `String` is an infinite set of values. This was an example to demonstrate how extension methods can amplify the type and enable developers to write expressions fluently without the need for creating intermediate statements or variables. This enables developers to write powerful reusable components for specific types. C# developers use extensions provided by *Language Integrated Query* (LINQ) for the `IEnumerable` interface all the time. LINQ, as one of the most important components (collection of types and extension methods) in the whole .NET, will be described in the following section.

3.3. LINQ

Language Integrated Query is a .NET Framework component that adds native querying capabilities to .NET languages. It is also a *grammar*, which is added directly into C# in the form of the *query syntax*. It is a replacement for older technologies like ADO.NET that adds a level of abstraction over data manipulation (database integrations, working with sequences, etc). Instead of writing raw *Structured Query Language* (SQL) queries, developers can express their queries more natively with compile-time checks and IntelliSense support. LINQ supports querying objects in memory (LINQ to Objects), SQL data (LINQ to SQL), XML data (LINQ to XML), and ADO.NET datasets.

Before LINQ and Entity Framework, .NET developers were writing raw queries. Sometimes it is desirable when it comes down to the performance when a complex query needs to be executed as the LINQ query is internally converted to SQL. However, in most cases, queries are rather simple, and having an integrated query technology is a big advantage.

```
SELECT UPPER(Name) FROM
(
  SELECT *, RN = row_number()
  OVER (ORDER BY Name)
  FROM Customer
  WHERE Name LIKE 'A%'
) A
WHERE RN BETWEEN 21 AND 30
ORDER BY Name
```

Snippet 16

The query in Snippet 16 is quite big for the amount of work it does. It returns the top 10 customers while skipping the first 20. This query can be written using LINQ syntax as shown in Snippet 17 which internally calls LINQ extension methods.

```
var query =
    from c in db.Customers
    where c.Name.StartsWith ("A")
    orderby c.Name
    select c.Name.ToUpper();
var thirdPage = query.Skip(20).Take(10);
```

Snippet 17

The comprehensive list of differences and the advantages of using LINQ over raw SQL that includes the examples shown in Snippet 16 and Snippet 17 is explained in detail by Joseph Albahari in his article „Why LINQ beats SQL“. In this thesis, we are going to focus only on the functional features of LINQ.

Many developers are not aware that LINQ is a functional library. The idea behind fluent API and `IEnumerable` interface with its corresponding implementations was to create a facility for working with sequences in a functional style as it is done in languages like Haskell and Scala. `IEnumerable` represents the base interface that all sequences compatible with LINQ are implementing and it satisfies both conditions of a monadic structure. There are lifting functions to create a sequence of any implementation class that can be represented as `IEnumerable<T>` where T can be of any type. There are also `SelectMany` functions that act as binding (flattening) functions along with `Select` functions that represent mapping functions from languages like Scala.

```
var result = Enumerable.Range(0, 100)
     .Where(i => i % 2 == 0)
     .OrderByDescending(i => i)
     .Select(i => $"{i}")
     .Aggregate((i, j) => $"{i}, {j}");

// 98, 96, 94, 92, … , 4, 2, 0
```

Snippet 18

From Snippet 18, fluent API syntax is visible where functions `Where`, `OrderByDescending`, `Select`, and `Aggregate` represent *higher-order functions* (HoF). A higher-order function is a function that either accepts or returns a function or does both and it is a concept from functional programming. Functions from the example in Snippet 18 are extension methods that accept a *predicate* as an argument and all of them except `Aggregate` return `IEnumerable` which means they can be chained. Because of this, we can express operations fluently. Therefore, LINQ extension methods are sometimes referred to as the fluent API.

C# also has a LINQ grammar where expressions can be written using query syntax as syntactic sugar that internally calls LINQ extension methods as already shown in Snippet 17.

```
var result = from i in Enumerable.Range(0, 100)
             where i % 2 == 0
             orderby i descending
             select $"{i}";
```

Snippet 19

However, not every extension method has its query syntax sibling. As it is visible from Snippet 19, the `Aggregate` method is missing as there is no way of expressing the aggregation using query syntax. It needs to be handled either by calling the extension method or manually by using a loop, which is not a functional approach. The advice from Microsoft is to use the fluent API as much as possible, as it is the most correct way of working with LINQ.

Developers use LINQ and generally work with sequences correctly and in a functional way, but with the rest of the types, developers work in an object-oriented (imperative) way which leads to unexpected behaviors. These unexpected behaviors are caused because of the problems that C# and .NET framework teams introduced into the language and the framework over the years.

3.4. Problems

One of the biggest mistakes in computer science was the invention of the *null reference*. It is referred to by its inventor as the billion-dollar mistake (Hoare, 2009). The concept of null became present in object-oriented programming languages and it found its way to C# as well. The null reference does not represent any object and it means there is nothing there. It is the default value of reference type variables. In C#, there are also nullable value types. They are represented as `Nullable<T>` where T is a value type. C# introduced a syntactic sugar where nullable value types can be defined as T?. In C# 8, the language team introduced nullable reference types that can be marked as T? as well. However, there is no framework support and therefore no `Nullable<T>` for reference types. The language team from the start introduced the problem that is now deeply integrated into the language and the runtime and will probably never change. Attempts to fix the issue are visible through partial solutions that we saw gradually taking place in C# and the .NET.

Another troubling issue in C# is the default mutability. Whenever you define a variable or a field that is not marked as readonly is mutable by default. Also, many framework types and their properties are mutable as well. There are a few immutable types like `String` and `DateTime` as well as immutable sequence types from the `System.Collections.Immutable` framework namespace. The idea behind immutability is that once the object is created it cannot be changed anymore. To make changes, a new object needs to be constructed. This ensures that unwanted mutations can never occur. Additionally, it ensures that in a concurrent environment, the objects are thread-safe and immune to race conditions. C# 6.0 had an improvement in this direction when the team introduced *getter-only auto-properties* that can be set only in the class constructor as shown in Snippet 21. This enabled defining properties easier if developers want to make sure the property is truly immutable.

```
public class User
{
    private readonly string name;
    public string Name { get { return name; } }
}
```

Snippet 20

Defining an immutable property was not straightforward as visible from Snippet 20. Developers could not take advantage of automatically defined backing fields for auto-properties as they needed to explicitly state that the field once set in the constructor cannot be changed later on (even within a class).

```
public class User
{
    public string Name { get; }
}
```

Snippet 21

From Snippet 21, we can see that with getter-only auto-properties it is easier to create immutable properties which shows a clear intention of the language team to make C# follow more functional programming principles than it used to. However, no matter if the property is get-only or not, the value of the object that is carried can be changed if it is not immutable.

```
public class User
{
    public List<int> Accounts { get; }

    public User()
    {
        Accounts = new List<int> { 123, 234, 345 };
    }
}

public static class Mutator
{
    public static void Mutate(this List <int> sequence) => sequence.Add(456);
}
```

Snippet 22

In Snippet 22, the `String` is replaced with `List` which is an implementation of the `IEnumerable` interface. `List` is a mutable sequence type, and therefore if the `Mutate` function would be called on the `Accounts` property, it would be able to change its value (add another item to the list). This leads to unexpected behaviors especially in larger programs where developers don't expect some function defined elsewhere to mutate the object that they are working with. For that reason, immutable collections are created in the .NET framework but are not widely used.

C# team has recently improved working with tuples. In the older versions, working with tuples was not easy and they were quite heavy on the memory. It was a reference type called `Tuple` with various arities. From the sixth version of C# onwards the team made a significant improvement creating a new type called `ValueTuple`. Instead of being a reference type, it is now a value type and the language even has syntactic sugar for defining them with parentheses (**()**). Also, from the seventh version, developers can assign meaningful names to tuple elements.

```
public (string Name, int Age) GetUser()
{
     return ($"John", 40);
}

var user = GetUser();
user.Age = 40;
```

Snippet 23

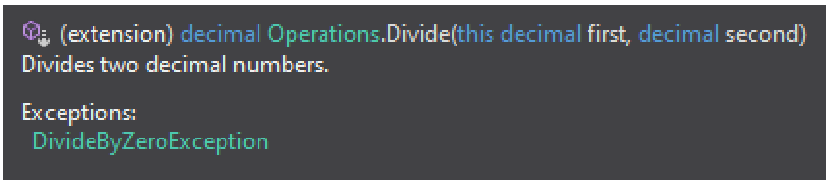
However, as it is visible from Snippet 23, `ValueTuple` is a mutable type and therefore cannot replace the missing feature that is planned for the future versions of the language referred to as `Record`.

Working with exceptions in C# is also a troubling task. Usually, exceptions are not handled properly and method signatures cannot indicate whether a certain method will throw an exception under a certain condition. Therefore, developers tend to write comments above methods to warn callers what type of exceptions can occur. This is a common practice throughout the whole .NET ecosystem including the framework. It is a bad practice because technical details should be visible in code and only business-related information should be commented on.

```
/// <summary>
/// Divides two decimal numbers.
/// </summary>
/// <exception cref="DivideByZeroException"></exception>
public static decimal Divide(this decimal first, decimal second)
{
     return first / second;
}
```

Snippet 24

From Snippet 24 it is visible how this function's signature is not telling the caller what to expect if the second argument passed is 0. As it was described in the second chapter, this is what makes this function dishonest. Therefore, developers write comments to warn the caller of the possible exception and make them handle exceptions for themselves.



```
(extension) decimal Operations.Divide(this decimal first, decimal second)
Divides two decimal numbers.

Exceptions:
  DivideByZeroException
```

Figure 3: Visual Studio tooltip function description

In Figure 3, we can see how Visual Studio shows the description of the function with the exceptions that could be thrown. This is a very weak approach to warning about possible errors and usually leads to program crashes as developers are not forced by the compiler to handle them.

The following chapter will describe in detail how to address these issues in a functional programming style and describe other functional programming concepts implemented in Funk.

4. The Funk Library

This thesis addresses the issues that mostly come from C# and not the .NET framework. The thesis does not address or discuss architectural decisions of making programs. Funk addresses the problems that developers face on day-to-day programming tasks using C# and focuses on solving code-related problems rather than on the architectural challenges of making robust programs. Concepts and solutions to those problems can be applied to other object-oriented programming languages as well.

In this chapter, the problems that are described in Section 3.4 will be addressed using concepts from functional programming that are inspired mostly from the *Category Theory*. Specific types created in Funk will be described in detail together with the functional `Prelude` and extension methods that can significantly improve and make day-to-day development easier and less error-prone. The chapter will conclude with the current state of Funk in the .NET community.

4.1. Category Theory

This section can be started with a quote: „Abstraction is the elimination of the irrelevant and the amplification of the essential“ (Martin, 2006). Abstractions help us understand things around us while letting us initially ignore the details that may distract us. Abstractions are very important in programming and many computer scientists and programmers came up with various kinds of abstractions to make complex and repeatable problems easier to tackle. We can see these abstractions from programming languages evolution through design patterns to new programming paradigms. Most of those abstractions have mathematics as the source of inspiration and more specifically the Category Theory.

A *category* is in the simplest sense a collection of objects along with the arrows that go between them. We call these arrows *morphisms*. Properties of the category are the associative composability of those arrows and the identity arrow for every object that serves as the unit under composition (Milewski, 2019). Category Theory is a branch of mathematics that is trying to unify all mathematics in terms of categories ignoring the details of what those objects and arrows represent.

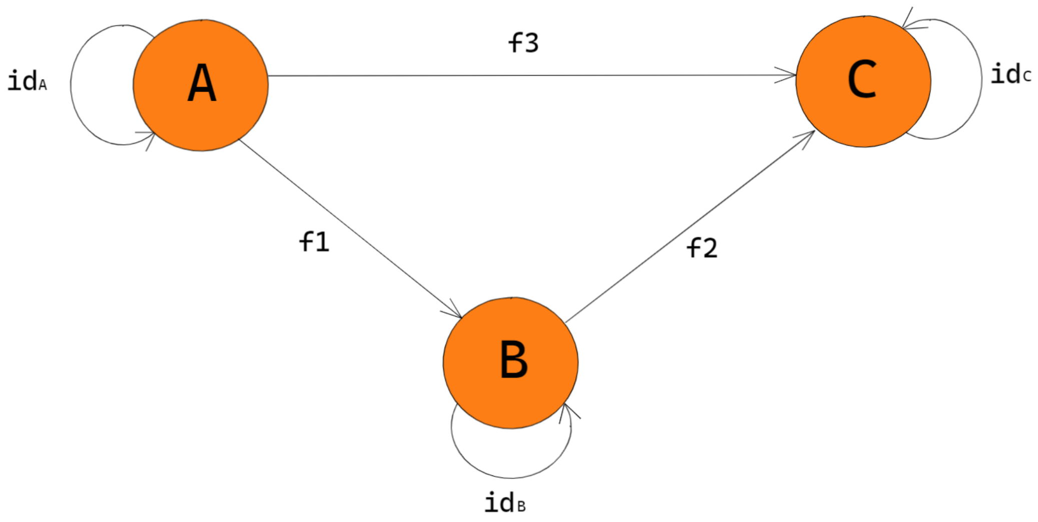


Figure 4: Category

In programming, we can look at these morphisms as functions where objects that are connected with them represent a function signature. In Figure 4 we have a function f1 that takes an object of type A and returns an object of type B. We also have a function f2 that takes an object of type B and returns an object of type C. Therefore, with the composability property, we have a function f3 ($f2 \circ f1$) that takes an object of type A and returns an object of type C.

The idea behind the Category Theory is that objects are abstractions and the only thing we care about is how they relate to other objects via morphisms. In object-oriented programming, we have interfaces that represent these abstractions and classes that are the concrete implementations. Functions are those morphisms and if we are forced to look into the implementation rather than just looking at the interface to understand the concrete object, we lose all the benefits of the paradigm.

Composability as the main property of the category is the essence of programming as well. Developers are trying to break down complex problems into smaller ones and solve them separately while being able to assemble the results back thus solving complex problems much easier. Types and a strong type system can help a lot with composability. The following section explains how Funk took concepts from the Category Theory and implemented them to make data representations more accurate and to help developers abstract away the repetitive and tedious work when dealing with the problems stated in Section 3.4.

4.2. Types

As already mentioned, Category Theory can be applied to all mathematics but in programming, we can look at the Category Theory as the *category of sets*. A set as a collection of distinct objects can be finite and infinite. A type in simple terms represents a set. For example, `Boolean` is a finite set of 2 values: true and false. `String` on the other hand is an infinite set of values as the string is a sequence of characters and there can be infinitely many distinct character sequences.

C# has a stronger and more powerful type system when compared to dynamic programming languages like JavaScript and Python. However, C#'s generic type system does not support *type classes* or *higher-kinded types* as opposed to what Haskell and some other functional programming languages support. From C# 8, we can say that the language now lays in the middle of object-oriented programming and functional programming. For a widely accepted industry language with a lot of functional programming features, this is enough to get started. Funk takes the language one step further into the functional programming direction and provides some of the missing types and features that are proven to be useful not only in functional programming but in programming in general. The types that Funk provides are `Unit`, `Record`, `OneOf`, `Exc`, `Maybe`, `Pattern`, `AsyncPattern`, `TypePattern`, and `AsyncTypePattern`.

4.2.1. Unit

There is a *Void* type in functional programming which is also familiar to developers from the object-oriented programming paradigm. In .NET, `Void` is a type (`struct`) and it is also part of the language through a keyword `void`. However, `Void` as a type cannot be used directly in C# nor an instance of it can be created. Therefore the following signature in Snippet 25 and the intended implementation of the method are not correct.

```
public static Void Do()
{
    // return Void instance.
}
```

Snippet 25

The `void` return type is used for functions that never return. In some sense, it is understandable to have functions that perform certain operations and don't return any result because there is simply nothing to return. From the functional programming perspective, the `void` is acceptable if the function does something consistently. However, when we have a void return type, from the signature, we can never be sure if the operation may fail. Since C# is not a pure language and usually the code that is written is not pure, there are possibilities that the void-returning function is impure and also dishonest. From Snippet 25 we could call the `Do` method without wrapping it in a try-catch block expecting that the operation will succeed and cause a program failure if it doesn't. A partial solution to this problem can be solved with the concept called the *Unit*. In functional programming, it is a well-known concept. It is represented with closed parentheses (**()**) and in some sense, we have it in many programming languages. Any function that takes no parameters takes the value of some *hidden type*. It is the only instance of that type and therefore unnecessary to explicitly define. We can also see it in C# in defining `Func` or `Action` variables after the equality sign and before the lambda expression as shown in Snippet 26.

```
Func<string> greeting = () => "Hello World!";
```

Snippet 26

Since we cannot use this representation of a single value set outside of these boundaries, we cannot take full advantage of the concept. Therefore, Funk implements a custom type called `Unit` which provides a representation that can be used in these and many other situations where we need to represent the data absence.

The `Unit` in Funk represents the absence of data (an empty value) and it only has one possible value, itself. From a mathematical standpoint which is expressed in *Russell's paradox* and later in the modern *Set Theory*, there is no such set that contains itself. However, since this is not a thesis that focuses on the Set Theory nor does it try to prove that there is such set we can say that the `Unit` is the one value set where that value is the `Unit`.

The value is not accessed from the instance level but from the `struct` level (`Unit` is a `struct`). The instance of the `Unit` contains no information. However, as already explained in the Category Theory section, the important thing about the object are the morphisms (functions) that go from or to it. The `Unit` is used heavily throughout the whole library and every other type depends on it.

```
var unit = new Unit();

var anotherUnit = Unit.Value;
```

Snippet 27

From Snippet 27, we can see two ways of creating an instance of the `Unit` type excluding the functional `Prelude` that is going to be described in the following sections. The first way is simply calling the `Unit` constructor and the second way is calling the `Value` property that returns the instance of the `Unit` by calling the underlying constructor.

Creating an instance of the `Unit` type while using this library will rarely be needed as any type provides it already through any expression. In the end, the `Unit` is there to denote the absence of data.

```
public static async Task<Unit> DoAsync()
{
	// implementation.
	return Unit.Value;
}
```

Snippet 28

However, when designing functions that return *nothing*, developers could instead use the `Unit` to indicate the operation completion as shown in Snippet 28.

```
Func<string, Unit> writer = message =>
{
	Console.WriteLine(message);
	return Unit.Value;
};
```

Snippet 29

It is also better to replace `Action` to `Func` functions that instead of being `void` return the `Unit` as shown in Snippet 29. By doing that, we are also able to unify the implementations where instead of having different methods that accept either `Func` or `Action`, reduce the code size and depend only on the `Func`.

The `Unit` type provides two `Match` functions following other types in the library. One returns the result specified by the selector passed as an argument during the call and the other is void-returning function as it accepts `Action`. These operations in normal situations would be rarely used but in case of completely avoiding statements, they help with additional expressiveness.

```
var unit = Unit.Value;

unit.Match(_ => Console.WriteLine(_.Match(__ => "John")));
```

Snippet 30

In Snippet 30, the outer `Match` function accepts the `Action<Unit>` parameter. The `Unit` type in the expression is denoted with the underscore (_) by the Funk convention. The inner `Match` accepts the `Func<Unit, T>` parameter where T is a result of the selector. Therefore, this expression prints *„John"* to the console.

Calling a `ToString` method on the `Unit` object results in the string *„empty"* indicating that the `Unit` contains no value. Calling the `GetHashCode` method which is used during equality check results in 0. It is also possible to check for equality using equal (==) and not equal (=!) operators besides `Equals` functions. Two `Unit` objects are always equal.

The `Unit` type object cannot be null as it is the value type!

4.2.2. Record

In C# there is a type that replaced the old reference type `Tuple` and it is called `ValueTuple`. It is a value type and it is mutable. The fact that it is mutable is enough motivation to create an immutable substitute. Value types in C# are mutable no matter what, but we can at least make members of that type immutable.

`Record` the same as the `ValueTuple` represents a concept from the Category Theory called a *product*. It is more specifically a *Cartesian product*. For a product of 2 sets, we have a set of pairs.

`Record` is a value type as well and in Funk, there are currently `Record` types from arity of 1 to the arity of 5 (`Record<T1,..,T5>`). Of course, implementing more would follow the same logic but as a proof of concept, Funk in most cases stops with the number 5 regarding generic type arguments.

```
var tuple = ValueTuple.Create("John Doe", 40);

var record = Record.Create("John Doe", 40);

var recordFromTuple = Record.Create(tuple);

var toRecord = tuple.ToRecord();

var single = "John Doe".ToRecord();
```

Snippet 31

From Snippet 31 it is visible that the creation of `Record` follows the `ValueTuple` API. It is not possible to use parentheses as C# provides syntactic sugar only for native types. However, there are extension methods that provide more fluent syntax as shown in lines 4 and 5. It is possible to create a `Record` object from the tuple and a single element as well. A functional `Prelude` way of creating the `Record` object will be described in the following sections.

`Record` type follows the same naming of its members for an easy replacement of `ValueTuple` in the code. It provides much more powerful handling and transformations. The `Record` type is also a monad because of the flattening functions it provides as shown in Snippet 33 and Snippet 34.

There are two `Match` functions that the `Record` type provides. The first one accepts `Func` and the second one `Action` similar to the `Unit`. By using pattern-matching functions, we can access the underlying members and perform desired operations without the need for having an imperative style of code.

```
public static Record<string, int> GetUser()
{
    return ("John Doe", 40);
}

GetUser().Match((name, age) =>
    $"User’s name: {name} and age: {age}."
).ToRecord().Match(Console.WriteLine);
```

Snippet 32

As it is visible from Snippet 32, the first `Match` function returns the result of the specified selector. In this case, since the `GetUser` function returns the `Record<string, int>` type object, two underlying arguments are projected to the resulting string. That string is lifted to the `Record<string>` again where the second `Match` function is the `void` one and the projected underlying value is printed to the console (*method group* used instead of lambda). Another interesting detail to notice is in the return statement of the `GetUser` function. The return value is of type `ValueTuple` and the compiler does not complain since there is an *implicit conversion* between the `Record` type objects and `ValueTuple` type objects. This is to enable even more natural feel when working with the `Record` type.

What makes the `Record` type a monad is the ability to lift any type to the `Record` type object as well as the flattening ability that is available through mapping functions. There are four mapping functions: `Map`, `FlatMap`, `MapAsync`, and `FlatMapAsync`. The first two are synchronous and the second two are for the *async* environment.

```
var newUser = GetUser().Map((name, age) =>
    (new { Name = name }, new { Age = age })
);
```

Snippet 33

Now `Map` function is not enough to make the `Record` type a monad but it is still useful in case when we don’t want to create or don’t rely on the `Record` type directly in the expression. As visible from Snippet 33, `Map` projects underlying values to the result of the selector lifting the return value to the same structure as it was before. In this case, it was the `ValueTuple` type. The `Map` function is a structure-preserving mapping function where the type of the underlying arguments is not relevant but the overall structure of the `Record` is. If the `Record` was a product of 2 sets, it is required to stay like that after the mapping. In this case, the `Record<string, int>` after the mapping function becomes the `Record<‘a, ‘b>` where primitives are now anonymous types but the `Record` object still has two underlying values.

```
public class User
{
    public string Name { get; private set; }

    public int SpouseId { get; private set; }
}

var john = await GetUserAsync(123);

var jane = await john.FlatMapAsync(j => GetUserAsync(j.SpouseId));
```

Snippet 34

In the concurrent environment having types compatible with asynchronous operations is crucial. In the example from Snippet 34, `FlatMapAsync` returns the `Task<Record<..>>` that can later be awaited to get the underlying `Record`. The `FlatMap` function makes the `Record` a monad since it preserves the structure and instead of returning `Record<Record<..>>`, it flattens it to the single `Record` object. `Map` functions use `FlatMap` functions under the hood.

Calling a `ToString` method on the `Record` object returns the result that `ValueTuple‘s ToString` method returns. `GetHashCode` method as well uses the underlying `ValueTuple‘s` method. It is also possible to check for equality using equal (==) and not equal (=!) operators besides `Equals` functions. Equality is checked using the underlying `Equals` method that `ValueTuple` provides. Because of the implicit conversion comparing the `Record` object and the `ValueTuple` object of the same arity is perfectly legal.

The `Record` type object cannot be null as it is the value type!

4.2.3. Maybe

As it was already mentioned, C# embraced null reference and now reference type variables' default value is null. That causes the notorious `NullReferenceException` when the object reference is not set to an instance of the object which is the most common exception that occurs in the .NET. From C# 8, C# has nullable reference types which in some ways helps in warning the developer that a certain reference type could be null. However, since it is not enforced by the compiler to be handled in any way, the feature appears to be a helper for Integrated Development Environments (IDEs) to warn about the possible `NullReferenceException` rather than providing a concrete solution. Additionally, since this feature is available only in C# 8, all previous versions cannot take advantage of it.

`Maybe` type is the monad that exactly addresses this issue. It is a type constructor for both value and reference types. In some of the other functional programming languages, it is referred to as the *Option*. It is a concept from Category Theory called a *coproduct* or a *disjoint union*. Coproduct is the opposite category compared to the product. It means that a coproduct of two sets can be a value from either one of these sets but not all. It is going to be described in more detail in Section 4.2.4. `Maybe` type is a *coproduct* of two sets: specified type used in the type constructor (T) and `Unit`. In Figure 5 we can see that `Maybe<T>` is a disjoint union of two sets: T and `Unit`. As already explained, the `Unit` type represents the absence of data. If the object wrapped in a `Maybe` type object is null, then the value of the `Maybe` will be empty (`Unit`). Otherwise, it will be the value of T.

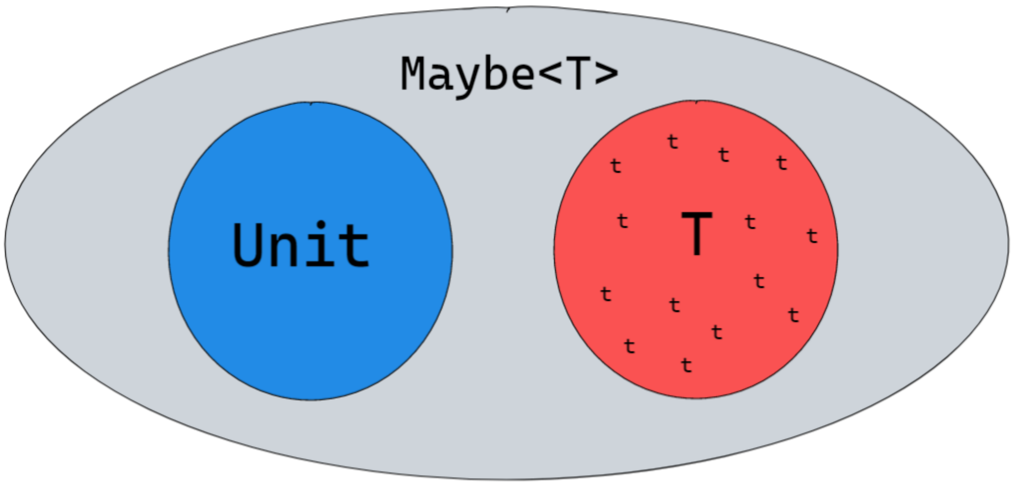


Figure 5: Disjoint union of 2

What makes the `Maybe` type a monad are the lifting and binding functions that it provides.

```
object empty = null;
int? nullable = 2;

var firstMaybe = Maybe.Create(empty); // Maybe<object>
var secondMaybe = nullable.AsMaybe(); // Maybe<int>
```

Snippet 35

From Snippet 35 it is visible that there are two different lifting functions (there is also `Maybe.Empty<T>` function to create an empty `Maybe`object). One is a factory method and the other is an extension method. Both of them perform the same operation and that is lifting the object to the higher construction of `Maybe`. The first variable, `empty`, is null, and therefore when wrapped in the `Maybe` type will result in an empty `Maybe` object. The second variable, `nullable`, is a nullable value type but it is not null, and therefore when wrapped in the `Maybe` type will result in a non-empty `Maybe`. The important thing to notice is the underlying type of `Maybe` when constructed with the nullable value type. It stops being a nullable value type as the value is resolved during the construction. Therefore T? when lifted becomes `Maybe<T>`. `Maybe` type is not only a more powerful replacement for manual handling of the null references but also for the `Nullable` type provided by the framework.

```
public static Maybe<User> GetUser(int id)
{
     return context.Users.FirstOrDefault(id);
}


var john = GetUser(123);

var jane = john.FlatMap(j => GetUser(j.SpouseId));
```

Snippet 36

This example from Snippet 36 is similar to the examples for the `Record` type. It is because monads follow the same principle when working with the underlying data. In this example, we can see the `GetUser` method is now, comparing to the dishonest functions mentioned in the earlier sections, letting the caller explicitly know that it may not return the user for a specified id as the return type is now wrapped in the `Maybe` type object. Now the caller is forced by the compiler to handle it to perform the desired operation. In this case, the binding function is called where the underlying value (if present) is used in the expression to retrieve another `User` object which may also not be found. If the value of the `john` object is empty, the function passed as an argument in the `FlatMap` method will not be executed. Using this approach, we can completely avoid statements and imperative style by being able to express the desired operation safely without the need to worry about the possible `NullReferenceException`. Another thing to notice is the implicit conversion between any object and the `Maybe` type object in the `GetUser` function. Because of that, we are not forced to call some of the lifting functions to turn the result into `Maybe` type of that result as it is done automatically.

```
var name = jane.Match(
    ifEmpty: _ => "No user found!",
    ifNotEmpty: j => j.Name
);
```

Snippet 37

In Snippet 37 we can see one way of retrieving the underlying value or its properties. `Match` is a pattern-matching function where the first parameter is the expression in case the `Maybe` type object is empty and the second one is the expression if the `Maybe` type object is not empty. Specifying the name of the arguments is not necessary but is used in this example for an easier distinguishment. In the first argument expression, underscore is used as the type is the `Unit` type object which represents empty value. In the second argument expression, `j` represents the `User` object. This way we can write expressions instead of statements and completely omit null checks, reduce the code size, and introduce additional safety. There are also two other `Match` functions. One accepts `Action` parameters for `void` operations and the other is used in the case when we want to throw the exception if the `Maybe` type object is empty.

```
var name = jane.Match(
    j => j.Name,
    _ => new NotFoundException()
);
```

Snippet 38

In Snippet 38, the second expression, where the `NotFoundException` is returned, is optional and if we don’t supply it and the `Maybe` type object is empty, `Match` function will throw the `EmptyValueException` which is a custom exception from Funk to indicate that the `Maybe` type object was empty.

```
var jane = john.Map(j => GetUser(j.SpouseId).UnsafeGet());
```

Snippet 39

There is also a `Map` function similar to the one in the `Record` type. Instead of flattening the result, it lifts the result of the expression to the `Maybe` of the result. As visible from Snippet 39, it is used since the result of the expression is the User and not `Maybe<User>`. This is because the `UnsafeGet` method was called indicating that instead of using the functional way of getting the value, we are using the imperative style. `UnsafeGet` function should be avoided as it throws `EmptyValueException` if the `Maybe` type object is empty and is not a functional way of resolving the underlying value. There are also asynchronous mapping functions available in the `Maybe` type. Other functions for the `Maybe` type related to its transformations will be described in the following sections.

Calling a `ToString` method on the `Maybe` object returns the result the underlying types‘ `ToString` method returns. `GetHashCode` method as well uses the underlying types‘ method. It is also possible to check for equality using equal (==) and not equal (=!) operators besides `Equals` functions. Equality is checked using the `Equals` method that the underlying type provides.

The `Maybe` type object cannot be null as it is the value type!

4.2.4. OneOf

In Category Theory, there is always a dual of some concept. As we already mentioned, a `Record` type represents a product of sets and a dual of a product is a *coproduct*. `Maybe` type is also a coproduct as it was already explained in the previous section. `OneOf` type represents the opposite construction of the `Record` type. The `Record` type is a product of values and `OneOf` type is a coproduct of values. A coproduct is a *disjoint union* sometimes called *discriminated* or *tagged union*. If a coproduct is a construction of two sets, then the value is an element of one of those two sets. In functional programming languages like Haskell, there is a construction called *Either* which represents a coproduct of two sets. `OneOf` type, on the other hand, is a more suitable name for this construction as it supports more than 2 sets (up to 5) and an additional set that represents an empty value. It is a type that handles the possible absence of data (null reference). It handles null reference similarly as the `Maybe` type does, but `OneOf` type is not a construct that should be used for that purpose. `OneOf` type is a reference type and therefore can be used as a base class for the custom implementation and modeling that will be described in the following examples. In Funk, `OneOf<T1, T2>` is used as a base class for the exceptional monad that is going to be described in the following section. Figure 6 shows the representation of the disjoint union of 3 sets that `OneOf<T1, T2>` represents.

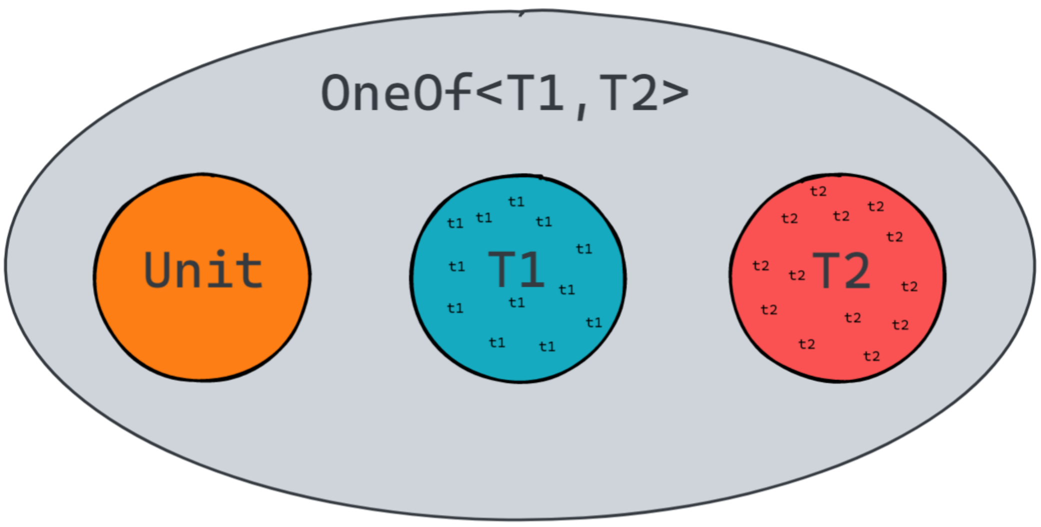


Figure 6: Disjoint union of 3

Even though we define `OneOf` type as the construct of two sets there is an additional set added implicitly because of the null reference handling and that is why `OneOf<T1, T2>` is a coproduct of three sets.

The `OneOf` type is not a monadic structure as it does not provide binding operations. However, it is still a very useful construct that can serve as a base class for custom types representing your data models.

For example, we can imagine an *event-sourced system* and how the `OneOf` type is a natural fit for designing data models. The core entity in event-sourced systems is an *event*. Events represent what happened in the system and are the transition footprint of states. In event-sourced systems, states are constructed from events and are usually not stored in the database unless they need to be cached for performance purposes. Each event can be constructed from the command received from the caller (user, client application, etc.), and together with the previous state which is constructed from the past events, using *state transitions*, produce a new state. State transitions are normal binary functions that take the current state and a new event (or command) and produce a new state and an event (in case the command was used). After that, the new state (or a state view model) is returned to the caller, the event is stored in the database and is published (to client applications or other interested parties).

The goal of the event-sourced architecture is to, because of constant changes, minimize the amount of transferred data, enable real-time data streaming, and store everything that is happening in the system. In an event-sourced system, data is never overwritten, only new data is stored. Therefore, an event-sourced system has all the necessary infrastructure for data analytics. Because of this type of architecture, usually a single entity like a User on some E-Commerce platform has many commands and events related to it. Soon, it can become quite overwhelming to work with hundreds of different commands and events in the system. OneOf type is a construct that can help minimize the complexity and abstract away some tedious and repetitive work.

We can construct a UserEvent class, as shown in Snippet 40, that implements `OneOf<T1, T2, T3>` type to work with user-related events using expressions with pattern-matching without the noise of statements.

```
public class UserEvent : OneOf<AccountCreated, InfoUpdated, AccountClosed>
{
    public UserEvent()
    {
    }

    public UserEvent(AccountCreated first)
      : base(first)
    {
    }

    public UserEvent(InfoUpdated second)
      : base(second)
    {
    }

    public UserEvent(AccountClosed third)
      : base(third)
    {
    }
}
```

Snippet 40

`UserEvent` class represents a simplified collection of possible events related to the User in some event-sourced system. As only one event can occur at a time, the `OneOf` type is a suitable container for these events. The first constructor is a default one but to work with `OneOf` type correctly all the overloaded constructors should be properly defined as shown in Snippet 40. The default constructor can be called in case the resulting `OneOf` object needs to be in an empty state without the need for calling one of the overloaded constructors passing a null argument as shown in Snippet 41.

Now, we can have the same representation for commands as well and the following snippet shows how we can work with the `OneOf` type using `Match` functions.

```
public Record<UserEvent, User> GetNew(UserCommand cmd, User user = null)
{
      return cmd.Match(
            _ => (new UserEvent(), user),
            f =>
            {
                var (newUser, created) = f.Create();
                return (new UserEvent(created), newUser);
            },
            s =>
            {
                var (newUser, updated) = user.Update(s);
                return (new UserEvent(updated), newUser);
            },
            t =>
            {
                var (_, closed) = user.Close(t);
                return (new UserEvent(closed), _);
            }
      );
}
```

Snippet 41

The `GetNew` function in Snippet 41 is responsible for creating a new `UserEvent` object together with the new state of the `User` object based on the `UserCommand` object and the current state of the `User` object passed as arguments. The `UserCommand` object is pattern-matched on its possible values and for each case, a different operation is performed. In the case the `UserCommand` object was empty, we are returning the empty `UserEvent` object together with the `User` object passed as an argument (if we omit the empty case and if the object is empty, the `Match` function will throw the `EmptyValueException`). In case the `UserCommand` object contained `CreateAccount` command object, the first case would be executed where `Create` function returns a `ValueTuple` object of the newly created `User` object and Account`Created` event object. The `ValueTuple` object of `UserEvent` object and the `User` object is returned which is implicitly converted to the Record type object. The other operations specified in the `Match` function for other possible cases follow the same logic. This example shows us how we can write a computation where we depend on different types without the need for statements and a lot of code. The example from Snippet 41 could also perform checking for possible null reference of the `User` object and return a `Maybe` type object of the result but the focus was on showing how pattern-matching works for OneOf type.

The OneOf type can be also used in the combination with the Maybe type as it exposes properties for its underlying members wrapped in the Maybe type object.

```
private void Log(UserEvent @event)
{
    @event.Third.Map(c => $"Account {c.Id} closed").Match(
        ifNotEmpty: m => _logger.Log(m)
    );
}
```

Snippet 42

The function Log from Snippet 42 is interested in logging only if the UserEvent object is AccountClosed object. Therefore, Third property is of type Maybe<AccountClosed> and in case it is not empty the Map function will return Maybe<string> which is then pattern-matched using Match function. In Match function, we are interested only in the case when the Maybe type object is not empty. We pattern-match on the message and in case it is not empty, _logger.Log function will log the message. If the object is empty on the other hand, the function does nothing.

Calling a ToString method on OneOf object returns the result the underlying types‘ ToString method returns. GetHashCode method as well uses the underlying types‘ method. It is also possible to check for equality using equal (==) and not equal (=!) operators besides Equals functions. Equality is checked using the Equals method that the underlying type provides. Underlying type is the non-empty object or the default value which is the Unit object.

OneOf type is not an abstract class and therefore can be used independently without the need to implement it in a custom type. Since OneOf type is a class, it is a reference type and therefore can be null!

4.2.5. Exc

Exception handling in C# is tedious work. Try-catch blocks are what makes code less readable and increases the overall codebase. Additionally, handling exceptions with a try-catch technique forces the caller to look into the implementation of a specified function to figure out the logic and to properly handle other exceptions that may occur or that may be thrown in that function or other related functions.

`Exc` is a type specifically created to solve this problem. As with previous types introduced by Funk, we are trying to abstract away repetitive and tedious work. `Exc` type is an exceptional monad sometimes referred to as a *Try* or a *Result* monad in other functional programming languages like Haskell and Scala. `Exc` is a type that is responsible for handling the possible failure of a specified operation. Like `Maybe` type is a container of a value where the value might not be present, `Exc` type is a container of a result of the operation that may have failed during the execution. `Exc` type implements `OneOf<T1, EnumerableException<T2>>` where T2 needs to be any type of `Exception`. Since `Exc` type implements `OneOf` type, it handles the possible absence of data as well. `EnumerableException` is a type of `Exception` provided by Funk that is a more powerful replacement for Aggregate`Exception`. It can contain one or more failures that happened during the computation depending on whether the operation was related to validation or a normal railway-oriented process. `EnumerableException` is a type that can be used outside of `Exc` type as well. Since `EnumerableException` exception type implements an `IImmutableList` interface, it is immutable.

In the rest of this section, we illustrate the usage of the `Exc` type on realistic examples. We can have an extension method that is responsible for deserializing a string into a specified type using the `Newtonsoft.Json` library. Traditionally, we would write this function as the following function shown in Snippet 43.

```
public static T SafeDeserialize<T>(this string content)
{
    try
    {
        return JsonConvert.DeserializeObject<T>(content);
    }
    catch (JsonException e)
    {
        return default;
    }
}
```

Snippet 43

The first problem with this approach shown in Snippet 43 is that we are forced to manually handle the possible failure of the `DeserializeObject` function. The second and even bigger problem is that, in case of an exception, we are returning the default value of T. Not only that it doesn't make sense since the operation failed, but we are also forcing the caller to look into the implementation to understand why the function `SafeDeserialize` returned null (in case the type T was a reference type). This function could be modified to return a `ValueTuple` object of T and an `Exception`. In this case, the caller would not be aware of whether the operation failed or not and would be forced to check if the `Exception` is null or not which again forces the caller to handle everything manually and could potentially introduce additional bugs into the code. The third option would be to throw a custom `Exception` and as shown in previous chapters, add a section in comments which states that the `SafeDeserialize` function throws some `Exception` which would make it unsafe immediately and the name of the function would not make any sense anymore.

`SafeDeserialize` function could be rewritten in a way that instead of returning T it returns an `Exc` type object of T and a `JsonException` as shown in Snippet 44.

```
public static Exc<T, JsonException> SafeDeserialize<T>(this string content)
{
    return Exc.Create<T, JsonException>(_ => JsonConvert.DeserializeObject<T>(content));
}
```

Snippet 44

As we can see from Snippet 44, `SafeDeserialize` is now an honest function. It tells the caller that the operation may fail and it can be caused by possible `JsonException` or any other derived `Exception` type. Additionally, we reduced the code size, made the function much more readable and we did not have to wrap anything in a try-catch block as it is done for us when we call the `Exc.Create` method. The `Exc.Create` method accepts a `Func<Unit, T>` argument where the provided function is internally wrapped in a try-catch block. Omitting the explicit `Exception` type definition during the call returns `Exc<T, Exception>` type object. Even though it makes much less noise, it is not advised to create it without the explicit exception definition as not all exceptions should be handled. Only the expected ones should be handled as we don't want to handle possible developer errors. There is also the asynchronous version of `Create` method called `CreateAsync` which accepts `Func<Unit, Task<T>>` argument and returns a `Task<Exc<T, E>>`. If the provided function during the invocation of the `Create` method returns null, the `Exc` type object will be empty.

`Exc` type is a monad because of the binding functions it provides. There are many different binding functions for the `Exc` type. All of them have the same signature but perform a different operation. Some binding functions represent a continuation in case the result is in a successful state. Some represent a fallback in case the result is in an empty `Exc` object or if the result is in the failure state. Most of them will be described in the following sections related to extension methods. In this section, we will describe binding functions that perform a continuation.

```
public static async Task<Exc<string, Error>> GetContent(this HttpResponseMessage
message)
{
    var response = await message.Content.ReadAsStringAsync();
    return message.StatusCode.Match(
        HttpStatusCode.Unauthorized, _ =>
            Exc.Failure<string, Error>(new UnauthorizedError(response)),
        HttpStatusCode.Forbidden, _ =>
            Exc.Failure<string, Error>(new ForbiddenError(response)),
        HttpStatusCode.BadRequest, _ =>
            Exc.Failure<string, Error>(new InvalidRequestError(response)),
        HttpStatusCode.OK, _ =>
            Exc.Success<string, Error>(response),
        _ =>
            Exc.Empty<string, Error>()
    );
}
```

Snippet 45

We can have a function like the one shown in Snippet 45 that reads the content of the `HttpResponseMessage` and returns the appropriate `Exc` result based on the content. The function first reads the content of the `HttpResponseMessage` object. Then the function pattern-matches on the `StatusCode` value of the message. This `Match` function is the extension method provided by Funk that will be described in detail in the following sections. It is similar to the naive `Match` function shown earlier in Section 3.2 but it handles more cases and has a default case and fallback capabilities as well. `Exc.Failure`, `Exc.Success` and `Exc.Empty` functions are factory methods to directly create an `Exc` type object in a specified state when there is no operation to perform as in the case when the `Exc.Create` function is used. The `Match` function returns the result of the selector that is specified under the correct case. It returns the result in the form of an Exc<string, Error> type object. If the `HttpStatusCode` value was `OK`, the result will indicate success. Otherwise, it is going to be either failure or, in case none of the specified cases were matched, the result will be an empty `Exc` object as specified in the default case.

Now, with functions `GetContent` and `SafeDeserialize` we can get the deserialized object based on the `HttpResponseMessage` using binding functions that perform continuation as shown in Snippet 46.

```
var response = await Http.SendAsync(request);

var obj = response.GetContent().FlatMapAsync(r =>
        Task.FromResult(r.SafeDeserialize<User>().MapFailure(e =>
              new Error(e.Root.Map(ex => ex.Message).GetOr(_ =>
                    "Unable to properly deserialize response returned by the server."
              ))
        ))
);
```

Snippet 46

As we can see from Snippet 46, once we get the `HttpResponseMessage` object we call the `GetContent` function from Snippet 45 and as a continuation in the process of getting the deserialized `User` object, we call the asynchronous `FlatMapAsync` binding function. Inside of the `FlatMapAsync` function, we pass the continuation argument of type `Func<string, Task<Exc<User, Error>>>` that will be evaluated only if the `GetContent` returns the successful result. Inside the continuation, we are calling `SafeDeserialize` function from Snippet 44 that will perform deserialization of the string to the specified object type. The problem here is that since `GetContent` function returns an `Exc` object of type `Exc<string, Error>` and `SafeDeserialize` function returns an `Exc` object of type `Exc<User, Error>` the compiler cannot infer the resulting type as the `Exception` types differ. The rule of binding and mapping `Exc` type objects is that the `Exception` type stays the same as only semantically similar operations should be chained. However, in this case, we have a `SafeDeserialize` function that is a generic function that can be used anywhere, but since it handles `JsonException` which is an `Exception` type from `Newtonsoft.Json` library, we can use `MapFailure` function to map one `Exception` type to another. Therefore we are mapping `JsonException` to `Error` type and the compiler is now able to infer the return type. Inside of the MapFailure, we are provided with the `EnumerableException` of the JsonException type that we can use to create a new message. EnumerableException has Root and Nested properties that represent the root and nested failures of an Exc type object in a failed state. Both of these properties are wrapped in a Maybe type object as they can be empty. Therefore, to get the message of the root exception inside of the EnumerableException, we need to use the Map function. The Map function, as already explained, returns the Maybe type object of the specified selectors return value. Therefore, we get a Maybe<string> type object. However, the constructor of the Error exception accepts a string as an argument. To avoid calling the Match function, we can use GetOr which is one of the extension methods for the Maybe type that will be described in the following section. The GetOr function either returns the value of the non-empty Maybe type object or returns the result of the specified selector. The whole result inside of the FlatMapAsync function is wrapped in a Task object as it the asynchronous binding operation which is used because GetContent function returns a Task of the result as well. The whole process can be expressed in the same way in a synchronous environment as well. This quite complicated operation in terms of possible failures, type incompatibility, and possible absences of data is solved in a single line of code. We completely avoided using statements and relied only on expressions and the fluent API Funk library provides.

Exc type provides Map functions as well that perform semantically the same operation as what Map functions perform for the Maybe monad. They, instead of flattening the result, lift the result of the selector to the higher construction of Exc by wrapping a selector in the Exc.Create function.

```
public Exc<User, NotFoundException> Get(int id)
{
    // implementation
}

var name = Get(123).Map(u => u.Name); // Exc<string, NotFoundException>
```

Snippet 47

We can have a function Get from Snippet 47 that returns an Exc type object of the User and the NotFoundException in case the User object is not found for a specified id. To get the name, we can call the Map function providing a selector that returns a Name value of the User object. The result of the operation is of type Exc<string, NotFoundException> as the Get function might not return the User object.

As we can see from both binding and mapping functions, the value always stays wrapped in the `Exc` type object. Actually, for every monad, the result always stays in the elevated value of that monad.

Since `Exc` type inherits from the `OneOf` type, it provides the same `Match` functions. Comparing two `Exc` type objects is also based on the implementations that `OneOf` type provides.

4.2.6. Patterns

Pattern-matching is one of the most important features of any functional programming language. It is so powerful because it simplifies the code by enabling developers to use expressions instead of statements. Pattern-matching is a process of checking the value of a given object by a given number of possible cases where, for the expression to be evaluated, the value of the object needs to exactly match the value of the specified case.

We saw an example of pattern-matching in Snippet 45 where we had to match the value of an `enum` to several specified cases and corresponding actions. The `Match` function from Snippet 45 is generic and works for any type. Additionally, all of the `Match` functions that available directly in types provided by Funk are considered pattern-matching functions.

The matching functions covered so far are useful in situations when we want to get the result immediately. For example, we are matching on an `Int` type object and based on the value, we are returning a corresponding string. Or we are matching on the `Maybe<HttpResponseMessage>` type object and based on its state, we are returning a `Task` of a string. However, sometimes we don't want to execute the operation immediately but would rather have the whole computation as a *lazy* object. `Lazy` is also a concept from functional programming and a native type in .NET. and is an object that contains the computation. Once invoked, the `Lazy` object invokes the computation and returns the result of the computation. `Lazy` type in .NET is a wrapper around `Func<R>` which also, like the rest of the delegate types, represents a lazy computation.

For lazy pattern-matching, we are introducing 4 new types: `Pattern`, `AsyncPattern`, `TypePattern`, and `AsyncTypePattern`. All of them are value types and implement the `IEnumerable` interface. They implement `IEnumerable` because of the `Add` method even though it is not necessary to properly implement the `IEnumerable` interface. However, when the correct `Add` method (needs to be void) is present it can be used implicitly through collection initializers from C# 6 onwards. Every item in the collection initializer calls the underlying `Add` method. The compiler is following the pattern because it can find a correct `Add` method and therefore it is not necessary to call `Add` method anymore. This *magic* happening inside of the compiler is outside of the scope of this thesis as we only care about how we can take advantage of that. This convention-based pattern-matching is explained in detail by Eric Lippert in his article „Following the Pattern“.

Now we can imagine a scenario when we want to have a lazy pattern-matching computation. In Snippet 48, we have a function that performs an expensive computation based on input value that will be provided later on in the process.

```
Task<Exc<Resource, Error>> GetResource(ResourceType type)
{
      // ..
}

public static AsyncPattern<Exc<Resource, Error>> GetResource()
{
     return new AsyncPattern<Exc<Resource, Error>>
     {
            (ResourceType.Info, i => GetResource(i)),
            (ResourceType.Publications, p => GetResource(p)),
            (ResourceType.Contributors, c => GetResource(c)),
            ("info", _ => GetResource(ResourceType.Info)),
            ("publications", _ => GetResource(ResourceType.Publications)),
            ("contributors", _ => GetResource(ResourceType.Contributors)),
            ("I", _ => GetResource(ResourceType.Info)),
            ("P", _ => GetResource(ResourceType.Publications)),
            ("C", _ => GetResource(ResourceType.Contributors)),
            (1, _ => GetResource(ResourceType.Info)),
            (2, _ => GetResource(ResourceType.Publications)),
            (3, _ => GetResource(ResourceType.Contributors))
      };
}
```

Snippet 48

Now this function, instead of evaluating the expression immediately, creates the `AsyncPattern` type object where the correct value will be matched using the `Match` function. We achieved that, instead of performing a possibly unnecessary computation, we have an object that contains a computation that can be executed when desired. This pattern-matching is more powerful compared to Match functions we saw until now because we were able to define cases of different types (enum, string, int), all in one expression. Additionally, this way of pattern-matching has no limitations regarding the number of cases whereas the Match extension method is limited to 10 cases.

This AsyncPattern type object can then be evaluated with the Match function as shown in Snippet 49.

```
var resource = await GetResource().Match(Console.ReadLine());
```

Snippet 49

The important thing to note here is the return type of the Match function. The Match function accepts an argument of type object and returns a Maybe type object of the result. It is because the case might not be matched and also because it does not provide the default operation as the default value can be later specified using some of the extension methods for Maybe type (like the GetOr function that we saw earlier). This pattern that we created is not only compatible with string inputs but with ResourceType enums and integers as well and can be used in multiple situations.

Pattern type behaves similarly as the AsyncPattern type but in the synchronous environment.

Another useful scenario is when you need to pattern-match on the type of the object. Now we can imagine a case where we have classes (User, Account, Cart) that represent a domain model in our system and they all implement the IEntity interface. We have a generic Repository class with the generic Update function that calls the Log function once the update operation is successfully finished. Since we want to for some reason log the information in the *Persistence* layer rather than in the *Business* layer of our application we need to somehow know with what entity are we exactly working with at the moment.

```
public void Log(IEntity entity)
{
    _logger.Log(entity.GetType().Match(
        typeof(User), _ =>
        {
            var user = (User) entity;
            return $"User with id {user.Id} was updated.";
        },
        typeof(Account), _ =>
        {
            var account = (Account) entity;
            return $"Account with number {account.Number} was updated.";
        },
        typeof(Cart), _ =>
        {
            var cart = (Cart) entity;
            return $"Cart from the user {cart.User.Id} was updated.";
        },
        _ => "Something else."
    ));
}
```

Snippet 50

In Snippet 50 we can see one way to do this operation. The problem here is that we are forced to use the `typeof` operator together with the dangerous casting to make use of the `Match` extension method. Instead, we could use `TypePattern` type.

```
public void Log(IEntity entity)
{
    _logger.Log(new TypePattern<string>
    {
        (User u) => $"User with id {u.Id} was updated.",
        (Account a) => $"Account with number {a.Number} was updated.",
        (Cart c) => $"Cart from the user {c.User.Id} was updated."
    }.Match(entity).GetOr(_ => "Something else."));
}
```

Snippet 51

This modified function from Snippet 51 looks much clearer. There was no need to use the `typeof` operator or to cast anything. We also reduced the code size and gained the optional laziness (but we called the `Match` function immediately). Since the function returns a `Maybe<string>` object we needed to unwrap the `Maybe` type object to get the string. Here we can see how the default value can be provided using the extension method for the Maybe type.

`AsyncTypePattern` type behaves similarly as the `TypePattern` type but in the asynchronous environment.

All pattern types are value types and therefore cannot be null!

4.3. Functional prelude

The majority of functional programming languages have certain operations, specific constructs, expressions, or complex values available through some fields or properties. `Prelude` in Funk is a class that can be imported as a static reference from C# 6 onwards. It provides simplifications for some constructs that are tedious to write. Prelude mostly focuses on providing functions for creating Funk types or native .NET types that sometimes cannot be inferred by the compiler when defined as local variables using the `var` keyword instead of the explicit type definition.

Creating a `Maybe` type object can be done using the `may` function as shown in Snippet 52.

```
using static Funk.Prelude;

// ..

var maybe = may("Funk");
```

Snippet 52

Working with immutable collections is great but sometimes it can be painful to create them.

```
var l1 = ImmutableList.Create(1, 2, 3);

var l2 = list(1, 2, 3);
```

Snippet 53

From the example in Snippet 53 we can see two ways of creating an immutable list. The function `Create` returns `ImmutableList` object whereas the `list` function returns an `IImmutableList` object. As we can see from the example, the second option requires less writing.

Defining the `Exc` type object in a successful or a failed state is also tedious work. It is much easier to use the function available in `Prelude` as shown in Snippet 54.

```
var suc = success<string, InvalidOperationException>("Funk");

var fail = failure<string, Exception>(new InvalidOperationException());
```

Snippet 54

Defining a `Record` type object can be painful when using the factory method. `Prelude` also provides the substitute for the `Record` type factory method as we can see from Snippet 55.

```
var record = rec("John", "Doe", 30);
```

Snippet 55

The `rec` function makes it easy to replace locally defined `ValueTuple` type objects with the `Record` type objects by adding rec in front of parenthesis.

Sometimes we need to create an empty value under a certain condition in the expression. Since there is an implicit conversion between the `Unit` type and `Maybe` and `Exc` types, we can use the `empty` function from `Prelude`. In most cases (when the type is not wrapped in some other type like `Task`), the compiler can infer which type is meant to be used.

```
private static Exc<string, Exception> Get(this Exc<int, Exception> exc)
{
        return exc.Match(
            _ => _,
            v => success<string, Exception>($"{v}"),
            e => empty
        );
}
```

Snippet 56

In Snippet 56, we have a function that pattern-matches on the `Exc` type object. In this case, we are only interested in a successful case. If the state of the `exc` object is not successful, we are just returning an empty `Exc` type object. In the first case, we are returning the value of the `Unit` type provided in the expression. In the second case, we are creating a string from the integer value wrapping it in a successful `Exc` type object, and in the last case, we are again returning the value of the `Unit` type by calling the `empty` function. The `empty` function is the static property that returns the `Unit` type object. We can do the same with the `Maybe` type.

Defining local `Func`, `Action`, and `Expression` variables is also tedious work as it requires an explicit type definition because the compiler cannot infer the type.

```
Func<int, int, int> Add = (first, second) => first + second;

var add = func((int first, int second) => first + second);
```

Snippet 57

As we can see from Snippet 57, we can define a local `Func` type objects without the explicit type definition using the `func` function. For `Action` types, there are `act` functions and for the `Expression` types, there are `exp` functions. `func`, `act`, and `exp` functions are available for `Func`, `Action`, and `Expression` types of various arities.

Creating the `EnumerableException` type object when working with it outside of the `Exc` type can be tedious as well.

```
var ex = EnumerableException.Create(new InvalidOperationException());

var e = exception(new InvalidOperationException());
```

Snippet 58

From Snippet 58, we can see how we can use `exception` function from `Prelude` to simplify the creation of the `EnumerableException` type object. The `exception` function also works with the sequence of exceptions.

Another type that is simplified by the `Prelude` regarding its creation is the `Task` type from .NET. In Snippet 59 we can see the traditional way of creating and running a `Task`.

```
var task = Task.FromResult("Funk");

var another = Task.Run(() => "Funk");
```

Snippet 59

Using functions provided by `Prelude` we can define the variables from the example in Snippet 59 much easier.

```
var task = result("Funk");

var another = run(() => "Funk");
```

Snippet 60

From Snippet 60 we can see how to define or run a `Task` using `result` or `run` functions. There are also overloaded `run` functions in `Prelude` that resemble `Run` functions provided by the `Task` type.

4.4. Extensions

As we already covered in Section 3.2, extension methods are functions available as `static` methods that are accessed as class functions rather than instance functions but make an impression that are available in the object directly. They make fluent API possible in C#. Funk provides various types of extensions for different types and use cases. There are currently more than 200 extension methods available in Funk but we will cover only the most important ones.

4.4.1. Funk types

We already saw a few extension methods earlier in Section 4.2 while covering available types. We saw an example of the `GetOr` function that is quite useful in case when we want to get the value from the `Maybe` type object but provide a fallback in case the object is empty. There is also an asynchronous version of the `GetOr` function as shown in Snippet 61.

```
Task<Maybe<string>> GetAccessToken(string username, string password)
{
	// ..
}

var token = await GetAccessToken("John", "Doe123").GetOrAsync(async _ =>
{
	var other = await GetAccessToken("John", "JohnDoe123");
	return other.GetOr(__ => $"Basic {GenerateBasicToken("John", "Doe")}");
});
```

Snippet 61

We can have a function like in Snippet 61 that generates access token based on credentials. However, because the credentials might not be correct, we are returning a `Maybe` type object of the generated token. Since we don't want to perform any operation with the token until we are satisfied that we have at least the basic token generated, the easiest way is to call the `GetOr` function. We call the `GetAccessToken` function with the first credential pair. Since credentials might not be correct, we are calling the `GetAccessToken` function again with the new credential pair inside of the

GetOrAsync fallback argument. Since those credentials might also not be correct we are returning a basic token generated from the user's name and surname.

Another useful extension is when we want to perform a similar operation to the one in Snippet 61 but we want to stay in the elevated world of the Maybe type.

```
var token = await GetAccessToken("John", "Doe123").OrAsync(_ =>
    GetAccessToken("John", "JohnDoe123")
);
```

Snippet 62

In Snippet 62, we defined an expression where we stated that if the Maybe type object as a result of the first GetAccessToken function call is empty, we want to perform another GetAccessToken function call that will be evaluated only if the previous one evaluates to the empty Maybe type object.

Lifting an object to the elevated world of the Maybe type using a factory method is tedious. Prelude can help is certain cases but extension methods are usually a perfect fit as they provide the most functional way of working with the data in C#. Therefore, Funk provides an extension method for lifting any type to the Maybe type.

```
var funk = "Funk".AsMaybe();

int? nullable = 2;
var two = nullable.AsMaybe();
```

Snippet 63

In the example from Snippet 63, we lifted a string to the Maybe type object using the AsMaybe extension method. We also lifted a Nullable<int> type object to the elevated world of the Maybe type using the same function.

Lifting objects to the elevated world of the Record type can also be painful if we don't use the function provided by Prelude. However, as for the Maybe type, Funk provides an extension method for lifting types to the Record type.

```
var record = "Funk".ToRecord();

var fromTuple = ("John", "Doe", 30).ToRecord();
```

Snippet 64

In Snippet 64, we can see that the first `ToRecord` function lifts the `String` type object to the `Record<string>` and the second `ToRecord` function lifts the `ValueTuple<string, string, int>` type object to the `Record<string, string, int>` type object.

Checking equality and casting are sometimes dangerous operations to perform when we don't know the underlying value of the object we are working with. Funk provides safe alternatives called `SafeEquals` and `SafeCast` functions.

```
var first = "Funk";
var second = "Funky";
var areEq = first.SafeEquals(second);
```

Snippet 65

In Snippet 65, we can see that we can compare only the objects of the same type. There is also the `SafeNotEquals` function that checks for inequality. Additionally, there are `SafeAnyEquals`, `SafeAllEquals`, `SafeEqualsToAny`, `SafeEqualsToAll` functions that compare a single object to items from the `IEnumerable` sequence.

```
public object GetUser(int id)
{
        // ..
}

var user = GetUser(123).SafeCast<User>(); // Maybe<User>
```

Snippet 66

In Snippet 66 we can see how `SafeCast` function is returning a `Maybe<T>`. It is because the casting operation might not be successful. Therefore, if it is not the correct type of the object, the `Maybe` type object will be empty.

We already saw an improvement in creating a `Task` object using the functions from `Prelude`. Funk also provides extension methods that enable the creation of the `Task` object more fluently as shown in Snippet 67.

```
var task = "Funk".ToTask();
```

Snippet 67

Sometimes, we have methods in our codebase that perform some async operation without returning a result. They return a `Task` instead of `Task<R>` type object. Funk provides an extension method that addresses this issue.

```
public async Task DeleteItem(int id)
{
    // ..
}

var taskWithResult = await DeleteItem(123).WithResult();
```

Snippet 68

From Snippet 68 we can see that `DeleteItem` function returns a `Task` object that can be awaited but not assigned. However, sometimes we want a result of the operation because we want to follow the best practices of functional programming and to avoid non-returning functions as much as possible. With the `WithResult` function, we are returning a `Task<Unit>` object if we don't provide an argument. There is an overloaded version that returns the result specified by the selector that will be executed after the provided `Task` object is finished processing.

When we are working with `IDisposable` objects we should dispose them after the usage. C# provides a nice way of doing this by wrapping it in a *using statement*. However, if we are working with more `IDisposable` objects at the same time, the code can get messy soon. Funk provides a nice way of doing this with the `DisposeAfter` extension method.

```
using (var client = new HttpClient())
{
     var exampleResult = await client.GetAsync("www.example.com");
}

var funkResult = await new HttpClient().DisposeAfterAsync(c =>
    c.GetAsync("www.funk.com")
);
```

Snippet 69

From Snippet 69, we can see the first part of the example that shows the C# way of working with `IDisposable` objects. The problem here is that the object is not disposed until we leave the scope of the using statement. With the `DisposeAfterAsync` function, the `IDisposable` object is disposed immediately after the function returns the result. Additionally, we managed to express this operation more fluently.

We already saw an improvement in creating the `EnumerableException` type object using the functions from `Prelude`. Funk also provides extension methods that enable the creation of the `EnumerableException` type object more fluently as shown in Snippet 70.

```
var e = new InvalidOperationException().ToEnumerableException();
```

Snippet 70

Working with the `Exc` type using mapping and binding operations in case of a continuation is sufficient but sometimes we want to react in case the `Exc` type object is empty or in a failed state. Therefore, Funk provides these functions as well.

```
var result = await Exc.CreateAsync<string, ArgumentException>(_ =>
GetNameByIdAsync(invalidId)).OnFailureAsync(e => GetNullStringAsync())
    .OnEmptyAsync(_ => GetNameByIdAsync(validId))
    .MapAsync(async s => s.Concat(await GetNameByIdAsync(validId)));
```

Snippet 71

In Snippet 71 we can see a pipeline of operations around the Exc type. First, we are creating the `Exc` type object calling a function and passing an invalid argument that is going to result in an `Exc` type object in a failed state. Then, we are reacting in case it is failed by calling the `OnFailureAsync` function providing a callback. In the callback function we passed as an argument, we are calling a function that returns a null `String` object. Then, we are reacting in case the `Exc` type object is now empty by calling the `OnEmptyAsync` function. In the end, we are simply mapping the resulting object to the new one specified by the selector. From this example, we can see how we can fluently express a pipeline of operations that may not be successful and a way to react in that case without using statements and helper variables. We expressed quite a complex asynchronous pipeline of operations in a single line of code.

4.4.2. Partial application

One of the most useful concepts in programming is *partial application*. It was not covered in previous chapters as it is not only related to functional programming or C#. However, in programming languages where functions are the bread and the butter of programs, partial application is a very important concept. Partial application is a concept where we are producing functions of smaller arity from functions of bigger arity by supplying a number of arguments to them. Usually, partial application is done by supplying arguments one by one. Therefore, if there is a function that accepts two arguments and returns a value we are, by using the partial application and by providing an argument to it, producing a function that accepts one argument and returns a value. The partial application can be quite useful when we want to abstract away or hide certain details of some operation.

For example, we can imagine a system that has a three-layer architecture including the Core layer, the Infrastructure layer, and the API layer and how the partial application can help with the necessary abstraction. Usually, we would create different classes responsible for dealing with operations specific to a certain layer in the system. Those services would be injected as dependencies using some *Dependency Injection* (DI) framework. In functional programming languages, we can think of dependencies as functions rather than classes or interfaces and we can use that approach in C# as well. Instead of creating classes with different responsibilities, we can define a function that is going to be partially applied and passed as dependency throughout the system.

```
public Func<IClient, Configuration, Request, Response> Function => (cl, co, r) =>
{
        return cl.DisposeAfter(c =>
        {
              c.AddConfiguration(co);
              return c.Process(r);
        });
};
```

Snippet 72

In Snippet 72 we can see a function that performs some processing operation. First, we need an `IClient` object that will be provided in the Core layer. Next, we need a `Configuration` object that will be provided in the Infrastructure layer. In the end, the API layer is providing the `Request` object that is received from the user and returns the `Response` object that will be the result of this function. By providing an argument one by one, we can abstract away unnecessary details of a specified function from certain layers by providing a function of a different signature.

```
// created in the Core layer.
// Func<Configuration, Request, Response>
var coreFunction = Function.Apply(new SmartClient());

// created in the Infrastructure layer.
// Func<Request, Response>
var infrastructureFunction = coreFunction.Apply(configuration);

// created in the API layer.
// Response
var response = infrastructureFunction.Apply(request);
```

Snippet 73

In Snippet 73 we can see the `Apply` function used in all three cases. The `Apply` function is a partial application function provided by Funk. We started with the `Function` object from Snippet 72 and by applying the first argument in the Core layer we ended up with the new function of smaller arity. The argument applied is remembered and will be used of course during the execution of the function but from the signature of the function, it is not visible in the next layer. Applying argument by argument we are separating the concerns where each layer is responsible for a specific set of operations and responsibilities. By using the partial application we can reason about the abstraction on a whole new level that we cannot do by using the traditional dependency injection approach.

4.4.3. Currying

A similar yet different concept is the concept of *currying*. Currying is the process of decomposing a function with any arity to the unary function that takes a single argument and returns another unary function that takes a single argument and so on. We can use currying with the partial application as well. We can take the same function from Snippet 72 and, by using `Curry` function provided by Funk, convert it to unary function with the signature `Func<IClient, Func<Configuration, Func<Request, Response>>>`. Currying does not perform any operation but is rather used for function optimization. Currying can be used as a preprocessing function for the code example shown in Snippet 73.

```
// Func<IClient, Func<Configuration, Func<Request, Response>>>
var curriedFunction = Function.Curry();

// created in the Core layer.
// Func<Configuration, Func<Request, Response>>
var coreFunction = curriedFunction.Apply(new SmartClient());

// created in the Infrastructure layer.
// Func<Request, Response>
var infrastructureFunction = coreFunction.Apply(configuration);

// created in the API layer.
// Response
var apiFunction = infrastructureFunction.Apply(request);
```

Snippet 74

As we can see from Snippet 74, only the first line is added to the code shown in Snippet 73. It creates a unary function that when invoked with the `Apply` function (or by the native `Invoke` function or by just adding parenthesis, because the partial application for unary functions is the method invocation). The important thing to notice is the signature of the `coreFunction` method. It is, compared to the binary function from Snippet 73, a unary function. Using the partial application with or without currying is a personal preference. Many developers stick only to the partial application as currying can be sometimes hard to grasp.

4.4.4. Applicative functors

The concept of monads and the monad laws are already explained in previous sections. However, there are other interesting concepts that include *functors* and *applicative functors* (*applicatives*) that are less powerful than monads but are sometimes sufficient enough to tackle a more general problem. The types that Funk provides that are monadic structures are also functors. A functor is a type for which the `Map` function is defined. Therefore, the `Record` type, the `Maybe` type, and the `Exc` type are functors, and because of the return and binding functions that they provide, they are also monads. Applicatives, on the other hand, are a similar concept but more powerful than functors and less powerful than monads. Applicatives are types for which the `Apply` and the return functions are provided. The return function is the same as for the monad type. It lifts the value from the regular world to the world of elevated values. The `Apply` function, on the other hand, is different. It is not the `Apply` function from the partial application concept but a mapping function that works in the elevated world. However, there are some similarities between them as we will see in Snippet 75.

Functors and monads as we already said provide mapping and binding functions respectively. They are classified as *world-crossing functions* (Wlaschin, 2015). World-crossing functions are functions that either take a value in the regular world and return a value in the world of elevated values or vice versa. Mapping functions take a value in the elevated world and a regular function (`T -> R`) and return a value in the elevated world. Binding functions take a value from the elevated world and a world-crossing function (`T -> E<R>`) and return a value in the elevated world. Applicatives, on the other hand, work in the elevated world. The `Apply` functions they provide take a function in the elevated world and a value in the elevated world and return a value in the elevated world.

```
using static Funk.Prelude;

public static Maybe<int> AddM(this Maybe<int> first, Maybe<int> second)
{
    var add = func((int a, int b) => a + b);
    return first.FlatMap(f => second.Map(s => add(f, s)));
}

public static Maybe<int> AddA(this Maybe<int> first, Maybe<int> second)
{
    var add = func((int a, int b) => a + b);
    return add.AsMaybe().Apply(first).Apply(second);
}
```

Snippet 75

In Snippet 75 we can see two functions for adding numbers. The `AddM` function is an addition function with the monadic implementation. The `AddA` function is an addition function with the applicative implementation. They produce the same result but we can reason about them differently. The monadic implementation first binds on the `first` variable and then maps on the `second` variable. If any of them is empty, the result of the operation will be empty as well. If both values are present, the `add` function that is called inside the `Map` function will be executed. The applicative implementation as already said stays in the elevated world. We are first lifting the `add` function to the elevated world of the `Maybe` type. Then we are applying the `first` argument to the `Apply` function and then immediately we are applying the `second` argument to the `Apply` function. We are actually, partially applying arguments in the elevated world. The `Apply` function for the applicative functor uses the `Apply` partial application function in its implementation. Now with the `Apply` function available, the `Maybe` type besides being a functor and a monad is also an applicative functor.

The `Exc` type is also the applicative functor as it provides the `Apply` functions following the same logic as the `Maybe` type. However, the `Exc` type provides validation functions as well. They are a different type of applicative functions created specifically for the `Exc` type. When we work with the `Exc` type, we usually use the continuation mapping or binding functions where we want to chain the operations as long as the result is successful. Sometimes, we use the callback mapping and binding functions when we want to prevent the operation for failing or resulting in an empty value. The problem in all of these cases is that when we get the failed `Exc` type object, we have the information about the failure of a single operation (unless we manually fill in the `EnumerableException` object with additional errors). It is of course a correct approach when we are interested in the usual operation pipeline (we either continue on success, or we prevent the failure). But in terms of validation, we are interested in getting all failures that happened along the way. Therefore, Funk provides `Validate` functions that, as opposed to `Apply` functions in case of failure, return all the errors that occurred.

```
public static Exc<int, DivideByZeroException> Add(
    this Exc<int, DivideByZeroException> first,
    Exc<int, DivideByZeroException> second
)
{
    var add = func((int a, int b) => a / b);
    return success<Func<int, int, int>, DivideByZeroException>(add)
        .Validate(first)
        .Validate(second);
}
```

Snippet 76

In the example from Snippet 76, from all items that represent a failure, the `EnumerableException` object will be merged with the rest of `EnumerableException` objects from other failures. Therefore, we will know all the operations that failed and the errors that caused that. Additionally, the `add` function will be executed only if the applied arguments in `Validate` functions all evaluate to success. In case the `add` function itself fails, it will result in a failed `Exc` type object.

4.4.5. IEnumerable sequences

When working with the `IEnumerable` interface, developers have good support from LINQ regarding the available extension methods for various operations. However, Funk provides interesting and useful extension methods that, instead of returning `IEnumerable`, return the `IImmutableList` object (if the return type is a sequence). It is because Funk is trying to make developers get used to working with immutable types as much as possible.

LINQ provides the `FirstOrDefault` function that finds the first element in the sequence or returns the default value (there is also the `First` function which throws an exception if the element is not found). Since we want to minimize the number of possible null values, Funk provides the `AsFirstOrDefault` function that instead of returning T returns the `Maybe<T>` type object.

```
var l = list(null, "Funk", "Funky", null);

var empty = l.AsFirstOrDefault();
var element = l.AsFirstOrDefault(i => i.IsNotNull());
```

Snippet 77

In Snippet 77, we can see the `IImmutableList` object initialized where the first and the last items are null. The first `AsFirstOrDefault` function is called without a predicate and returns the empty `Maybe` type object. The second `AsFirstOrDefault` function is called with a predicate where it is specified that the item cannot be null (using the `IsNotNull` extension method provided by Funk) and it will evaluate to non-empty `Maybe` type object where the underlying value will be "Funk". There are also `WhereOrDefault` and `AsLastOrDefault` extension methods that behave similarly.

Sometimes, we have a sequence of items that might be itself null or contain no values. There are native `Any` and `Count` function but they are usually combined with statements. Instead, we can use the `AsNotEmptyList` function, as shown in Snippet 78, that returns the `Maybe` type object of the `IImmutableList` sequence which enables us to map on it, and only in case it is not empty, we are calling the `First` function.

```
var first = list.AsNotEmptyList().Map(li => li.First());
```

Snippet 78

By using the `AsNotEmptyList` function, we are certain that the `First` function will not throw the exception in case the sequence was empty. In Funk, the majority of extension methods provided for the `IEnumerable` objects are handling possible null sequence objects.

Another useful extension method is when we are working with a sequence of objects that can be split into two categories. Instead of grouping them, we can use the `ConditialSplit` function provided by Funk.

```
var l = list("Funk", "Funky", "Fun", "FYI");

var split = l.ConditionalSplit(i => i.Contains("Funk"));
```

Snippet 79

From Snippet 79, we can see the `ConditialSplit` function is called with a predicate. It returns the `Record` type object of two `IImmutableList` objects. The first member of the `Record` type object is a sequence that satisfies the predicate and the second one is the one that doesn‘t.

Funk also provides different versions of the `MapReduce` function. Instead of first projecting and then aggregating items from the sequence, we can use the `MapReduce` function instead and do everything at once.

```
var l = list(1, 2, 3, 4);

var maybe = l.MapReduce(i => $"{i * 3}", (i, j) => $"{i} < {j}");
```

Snippet 80

As we can see from Snippet 80, the `MapReduce` function returns the `Maybe` type object of the result. It is because the sequence might be empty. In the `MapReduce` function, we are first projecting each item to the new object (like the `Select` method does) and then we are aggregating on the projected values (like the `Aggregate` method does). The underlying value of the `maybe` object in the example from Snippet 80 will be "3 < 6 < 9 < 12".

4.4.6. LINQ compatibility

The last thing we will cover in this section is the LINQ query support for the `Maybe` and the `Exc` types. We already explained that the LINQ query support is the syntactic sugar for expressing LINQ extension methods functions with the LINQ query syntax. To be able to do that for some type, the appropriate `Select`, `SelectMany`, and `Where` functions need to be defined. Since Funk provides these functions, sometimes complex mapping and binding operations can be expressed in a much simpler and more readable way. The following example shown in Snippet 81 shows the expression of the operation using the LINQ fluent API (using extension methods directly).

```
var listOfKeywords = list("Funk", "Funky", "C#", "Functional");

var filtered = listOfKeywords.WhereOrDefault(i => i.Contains("Funk"));

var result = filtered.SelectMany(
            s => s.AsLastOrDefault(j => j.Contains("Funky")),
            (s, i) => new {s, i}
    )
    .Select(t => new {t, len = t.i.Length})
    .Where(t => t.len == 5)
    .Select(t => filtered);
```

Snippet 81

In the example from Snippet 81, we are filtering the list for items that contain the word "Funky". Then, we are getting the last item on the list only if it contains the word "Funky". After that, we are projecting the optional filtered sequence with the optional last item (optional because they can be empty), where we are evaluating the length of the last item. If it satisfies the condition of having a length of 5, we are returning the filtered list. If any of the values evaluates to empty, the result will be empty. Only if all conditions are met, the underlying value of the return object will contain two elements: "Funk" and "Funky". From Snippet 81, it is quite hard to understand what is going on. In this case, it is much clearer to represent this query using the LINQ query syntax.

```
var result = from s in filtered
    from i in s.AsLastOrDefault(j => j.Contains("Funky"))
    let len = i.Length
    where len == 5
    select filtered;
```

Snippet 82

In Snippet 82, we can see the quite complex pipeline that is expressed in Snippet 81 is now much more readable and understandable. First, we are getting the underlying value from the filtered sequence if not empty. Then, we are getting the underlying value of the last item in the sequence that satisfies the condition that it needs to contain the word "Funky" if not empty. Then we are defining an intermediate variable that we will use in our condition to be satisfied that the length must be equal to 5. In the end, we are returning the filtered sequence. And again, if all the conditions are met, the resulting sequence will contain two elements: "Funk" and "Funky".

LINQ query syntax for the `Exc` type object follows a similar logic. Only if every part of the expression evaluates to success, the result of the operation will be successful. However, in case some condition is not met along the way, the operation will result in the empty `Exc` type object.

```
var success = Exc.Success<string, Exception>("John");

var result = success.SelectMany(
            s => Exc.Success<string, Exception>($"{s} Doe"),
            (s, f) => new { s, f }
    )
    .Where(t => !t.f.Contains("John"))
    .Select(t => t.f);
```

Snippet 83

In Snippet 83, we can again see the fluent API way of expressing the operation. As with the previous comparison, we will see how to make this expression much more readable using the LINQ query syntax. In this example, we are defining a successful Exc type object. Then we are binding on its value similarly as we did in the example from Snippet 81. Then we are projecting both values (first success, and the second one that is created from the underlying value of the first one) to the new object that will be evaluated in the `Where` extension method. Here, we are working with success values for simplification but we could instead use the `Exc.Create` function to evaluate the specific operation. If everything is satisfied, the resulting operation will return the `Exc` type object in a successful state. Otherwise, it can be either failed or an empty `Exc` type object. In this example, since the condition specified in the `Where` function is not met because the second object contains the word "John", the result will be an empty `Exc` type object. As we can see, even though it is a simpler operation then the one we expressed in the example for the `Maybe` type, it is still hard to understand what is going on. We can make this much more readable using the LINQ query syntax.

```
var result = from s in success
    from f in Exc.Success<string, Exception>($"{s} Doe")
    where !f.Contains("John")
    select f;
```

Snippet 84

In Snippet 84, the expressed operation is much more readable compared to the one in Snippet 83. We can understand what is happening in the operation as we read one line after another. First, we are getting the optional value from success. Then, we are getting the optional value from another success. After that, we are expressing the condition that has to be met. In the end, we are returning the second optional value. Again, they are all optional as they can be successful, failed, or empty.

As we can see, Funk can simplify a very complex pipeline of operations. It does so by either providing dedicated types with their transformations to abstract away repetitive and tedious work or by taking it one step further where it simplifies the already simplified code.

5. Implementation

This chapter will talk about the way the Funk library is implemented. It is going to talk about the structure of the projects that are part of the Funk solution. It is also going to cover the tools and technologies used to develop the Funk library as well as the technical challenges encountered during the Funk design and development.

5.1. Structure, technologies, and tools

The Funk solution contains three projects: `Funk`, `Funk.Tests` and `Funk.Demo`. `Funk` is a project that contains the library and is targeting the .NET Standard 2.0 framework. It is written in C# 7.3. .NET Standard 2.0 is compatible with .NET Framework 4.6.1 and .NET Core 2.0. Some older projects might not be able to use the Funk library, but it is possible to compile the Funk library from the source code targeting the older framework versions. `Funk.Tests` project contains unit tests and is targeting .NET Core 3.1 and is using `xUnit` and `FsCheck` testing libraries for assertion-based and property-based testing. There are around 150 unit tests in the Funk.Tests project. `Funk.Demo` is a project that contains the demo application that is built on top of the Funk library. It will be described in detail in the next chapter.

Table 1 shows the tools used for the development of the Funk library.

Table 1

Purpose	**Tool**
Development	Visual Studio 2019, Visual Studio Code, JetBrains ReSharper
Testing	Visual Studio 2019, JetBrains dotCover, GitHub
Communication	Gmail
Planning	Taskade
DevOps	git, GitHub, NuGet

5.2. Technical challenges

The development of the Funk library was an interesting journey where we faced a few difficult problems that we had to find a solution for. These problems were more related to reasoning about the functional programming concepts and how to implement them by making them useful and easy to grasp for a .NET developer. Functional programming, being a completely different paradigm from object-oriented programming, requires a different state of mind and a different problem-solving approach.

The first problem that was difficult to tackle was the representation of the absence of data. It may seem easy now after we went through the previous chapters but properly representing the absence of data is hard when there is no great language support. As we already mentioned, the `Void` type cannot be used directly in C#. Therefore, creating the `Unit` type was the foundation of reasoning about the representation of the absence of data in Funk in the functional programming way. After the `Unit` type was implemented, implementing the `Maybe` and the `OneOf` types that handle the possible absence of data was much easier to reason about.

One of the difficult problems to solve was the pattern-matching functionality outside of the `Match` extension method. The `Match` function is intended to be used for simple pattern-matching operations that don't have to be lazy-evaluated. Additionally, the `Match` function has a limitation of 10 cases excluding the fallback functions. Pattern types in Funk for value and type evaluation were first planned to be normal `structs`. However, implementing them as the `IEnumerable` (not generic) objects made it possible to define cases with different types whereas the `Match` extension method is limited to the type that is being evaluated. Additionally, it made the pattern-matching syntax look more like the one that is available in C# 8.

It was also quite hard to reason about pattern-matching, mapping, and binding functions for Funk types and making them feel the same as if C# had higher-kinded types. Therefore, all Funk types are made in a way to follow a similar pattern. A lot of duplicate code was needed to be written. For every Funk type, a dedicated set of pattern-matching, mapping, and binding functions needed to be created.

Additionally, during testing and the demo program writing process, the `Exc` type was the type that changed the most. It is because of many edge cases that we found worth exploring and that needed to be covered enabling seamless adoption of the type without the need for developers to write mapping and binding fallback functions themselves.

6. Case Study

The case study includes demo projects written on top of the Funk library and the evaluation of results. Demo projects include the smaller project that is created on top of Funk from scratch and the larger project that was refactored to use the Funk library.

6.1. Small demo project

The small demo project is a console application that implements the client for Medium, an online publishing platform, for communicating with Medium open APIs. It is built on top of the Funk library using C# 8 and .NET Core 3.1. The `Funk.Demo` project as already mentioned is part of the Funk solution.

The usage of the application is rather simple. Users are asked for their integration tokens in the beginning. After that, users can choose what resources they are interested to see. Users can check their information, publications, and publication contributors. The Medium client is limited only to `GET` requests for the sake of simplicity.

The integration token is represented as the `Identity` class.

```
public sealed class Identity
{
    public Maybe<string> Token { get; }

    public Identity(string token)
    {
      Token = token.AsNotEmptyString();
    }
}
```

Snippet 85

In Snippet 85, we can see that the `Identity` class is the simple type where the constructor accepts the `token` parameter and applies it to the `Token` property where the `AsNotEmptyString` function is called. `AsNotEmptyString` function checks whether a specified string is null or empty, and returns a `Maybe` type object of that string.

Resource as the base entity is implemented as the `OneOf<T1, T2>` type.

```
public sealed class Resource : OneOf<Info, Publication>
{
    public Resource(Info info)
      : base(info)
    {
    }

    public Resource(Publication publication)
      : base(publication)
    {
    }
}
```

Snippet 86

From Snippet 86, we can see that it implements the `OneOf` type constructor with the `Info` and `Publication` types. `Info` type is a type with the user properties whereas the `Publication` type is also the `OneOf<T1, T2>` type where T1 represents the list of publications and T2 represents the list of contributors.

All functions in the `Funk.Demo` program are static and made as extension methods so that they can be chained together with functions that Funk provides making the code more readable and more expressive. The core function in the program is the `Get` function.

```
private static Task<Exc<T, Error>> Get<T>(this Identity identity, Uri uri)
{
    return identity.Token.ToExc<string, Error>(_ =>
        new InvalidRequestError("Token cannot be empty.")).FlatMapAsync(token =>
            uri.CreateGetRequest(token).DisposeAfterAsync(req =>
                req.SendAsync().DisposeAfterAsync(res =>
                    res.GetContent().FlatMapAsync(r =>
                        result(r.SafeDeserialize<T>().MapFailure(e =>
                            new Error(e.Root.Map(ex => ex.Message).GetOr(_ =>
                                "Unable to deserialize response returned by the server.")
                            )
                        )
                    )
                )
            )
        )
    );
}
```

Snippet 87

From Snippet 87, we can see that we can write a complex pipeline of operations in a few lines of code (in this example, the code is written in more lines because of the readability and can be written in a single line of code). Here, we can see a complete code of the example shown in Snippet 46 when we were talking about the Exc monad. Here we express quite a few operations that are important to get the correct result. From the interface of the Get function, we can see that it is a generic function and that it accepts the Identity object and a Uri object and returns the Exc type object of the T type and the Error exception (wrapped in the Task object).

The Get function first checks the Identity object by applying the ToExc extension method to the Token property. The ToExc function is an extension method for the Maybe type that, in case the Maybe type object is empty, returns the Exc type object in a failed state. In the argument of the ToExc function, we are specifying the exception that we want to create in case the Maybe type object is empty. If it is not empty, on the other hand, the Exc type object will be in a successful state. Then, we are binding on the resulting Exc type object, where we are defining a continuation in case the object is successful. Using the CreateGetRequest function, we are creating a new HttpRequestMessage object. Since HttpRequestMessage implements the IDisposable interface, we are calling the DisposeAfterAsync function while passing the next operation. In the next operation, we are calling the SendAsync function that returns the HttpResponseMessage object which is also IDisposable. Then, we are again calling the DisposeAfterAsync function where we are passing the next operation. The next operation calls the GetContent function passing the HttpResponseMessage object as an argument. The GetContent function returns the serialized response (string object) in case the response was successful. After that, we are binding on that response, and we are calling the SafeDeserialize function that tries to deserialize the string to the specified type T. Since the SafeDeserialize function returns the Exc type object with a different kind of exception, we are calling the MapFailure function to map the JsonException to the Error exception type in case the SafeDeserialize function returns the failed Exc type object.

```
var info = await identity.Get<Info>(new Uri($"{Medium.BaseUrl}{Medium.Info}"));
```

Snippet 88

From Snippet 88, we can see how the Get function is called to return the user's information. The Get function is used by the GetResource function that, based on the user's input, decides what resource to return. Then, in the Main function, based on the user's input, the resource is pattern-matched to return the appropriately formatted response.

As we can see from this small demo application, we were able to define a complex set of operations in a very simple and readable way and any developer that looks at this code will be able to deduce what is happening in the code after a single reading. Additionally, we also took care of the memory by disposing managed objects without impacting the readability and without breaking the fluent way of expression. This example in Snippet 87 shows the power, safety, and clarity that the Funk library provides. Without using the Funk library, we would define this set of operations in a way shown in Snippet 1 with manual exception handling, null checking and would still end up with a dishonest function.

6.2. Large demo project

The large demo project is the refactoring of the backend part of the STOCK project that was implemented as part of the Group Software Project course at MFF UK. The purpose of the refactoring was to prove that the Funk library can be used in larger and enterprise applications and can significantly simplify the overall codebase with its powerful abstractions.

The STOCK project was developed throughout 2019 and was defended in April 2020. The purpose of the project was to create a platform for the creation and analysis of the machine learning models built to predict future stock prices based on historical data and sentiment extracted from financial articles. Users were able to create various kinds of machine learning models including regression, decision tree, and neural network machine learning models. Users were also able to read the latest financial articles, see the latest stock prices, follow certain companies, etc. The project was split into three modules: the backend, the frontend, and the Machine Learning (ML) module. The backend module was split into 7 projects and it was built using C# 8 and .NET Core 3.1 (ASP.NET Core 3.1 for the API project).

The backend is built using the *Clean Architecture* (Martin, 2017) approach. The clean architecture is a software architecture pattern where the software is split into three main modules: the core, the infrastructure, and the presentation. The core layer defines the interfaces together with the domain models. The infrastructure layer implements these interfaces. The presentation layer uses these implemented interfaces through the *Inversion of Control* (IoC) pattern. The inversion of control is a pattern that helps us depend on abstractions rather than concrete implementations. Dependency injection frameworks provide mappings between these abstractions and implementations. This way, we can inject corresponding dependencies with respect to different environments and configurations which makes our system much more flexible for potential change. In the clean architecture sense, it means that the presentation and the infrastructure don't communicate or depend on each other directly but rather through the core layer. By abstracting away the implementation as much as possible and by depending only on interfaces we can, when necessary, modify the implementation of one part of the system without the need for modifying the rest. Additionally, it makes unit testing much easier. The clean architecture does not have strict rules as long as the core concepts of the pattern are kept intact.

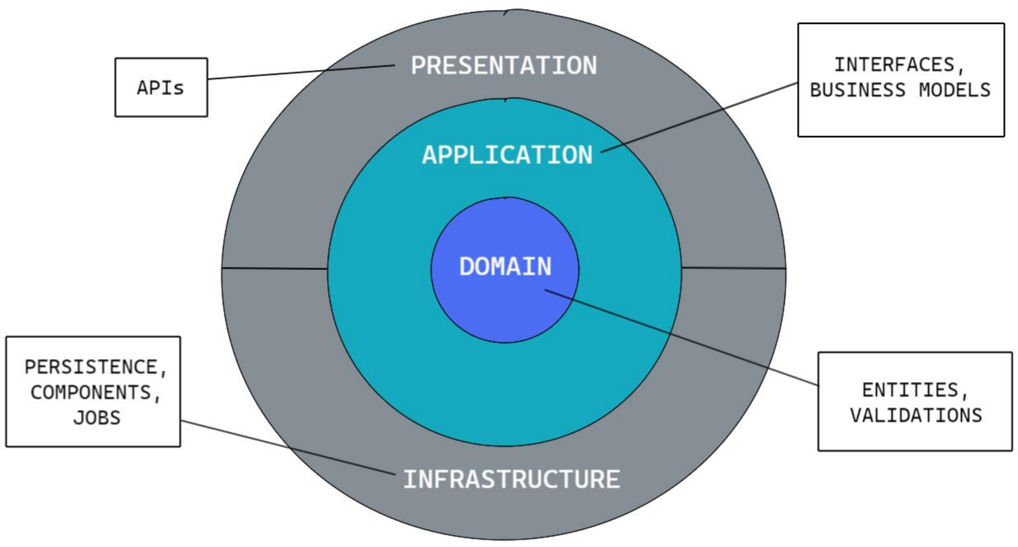


Figure 7: STOCK architecture

In Figure 7, we can see how the backend of the STOCK project was designed. The domain and the application layers represent the core unit of the clean architecture pattern. The outer layer consists of the presentation and the infrastructure layers that resemble the ones in the clean architecture pattern. As we can see from Figure 7, the application layer depends on the domain layer, and the infrastructure and the presentation layers depend on the core but not on each other. These layers are split into separate projects as shown in Figure 7. However, describing the details of the architecture of the backend is outside of the scope of this thesis. Here, we will see how the Funk library improved some parts of the code and made some abstractions even easier to implement.

In the domain layer of the STOCK backend, entities are constructed using factory methods where the validation is immediately applied. Validation is based on the design and business regulations of the entire application.

```
public static User Create(
      string username, string firstName, string lastName,
      string language, IEnumerable<Company> favoriteCompanies,
      IEnumerable<SearchedCompany> searchedCompanies
)
{
      var user = new User(username, firstName, lastName, language,
            favoriteCompanies, searchedCompanies
      );
      user.Validate();
      return user;
}
```

Snippet 89

The problem with the `Create` function from Snippet 89 is that it is dishonest. The `Validate` function throws the `ValidationException` if the constructed `User` object is not valid. We cannot see that from the signature of the `Create` method which forces us to look into the implementation. Additionally, we need to wrap the call of this function with a try-catch block. We were able to fix that with the `Exc` type.

```
public static Exc<User, ValidationException> Create(
      string username, string firstName, string lastName,
      string language, IEnumerable<Company> favoriteCompanies,
      IEnumerable<SearchedCompany> searchedCompanies
)
{
      return Exc.Create<User, ValidationException>(_ =>
      {
            var user = new User(username, firstName, lastName, language,
                favoriteCompanies, searchedCompanies
            );
            user.Validate();
            return user;
      });
}
```

Snippet 90

In Snippet 90, we can see the modified version of the `Create` function. The modified version tells the caller that it may not return the `User` object because the creation may fail due to the failed validation process which is visible from the return type. Now, the caller is forced to handle the return type by either pattern-matching on the `Exc` type object or by performing mapping or binding operations depending on the scenario.

The persistence layer implements one generic repository class with `virtual` methods in case there is a need for overriding a behavior for a specific entity.

```
public virtual async Task<TEntity> GetAsync(
      IClientSessionHandle session,
      Func<FilterDefinitionBuilder<TEntity>, FilterDefinition<TEntity>> filter
)
{
         var many = await GetManyAsync(session, filter);
         return many.FirstOrDefault();
}
```

Snippet 91

In Snippet 91, taken from the original STOCK code, we can see the `GetAsync` method for returning one item from the database. It calls the underlying `GetManyAsync` function to return more items based on the `FilterDefinition` object (MongoDB specific). Both `GetAsync` and `GetManyAsync` methods are dishonest as they may not return anything. Therefore, the caller is forced to check if the returned object is null. We were able to fix that with the `Maybe` type.

```
public virtual async Task<Maybe<TEntity>> GetAsync(
      IClientSessionHandle session,
      Func<FilterDefinitionBuilder<TEntity>, FilterDefinition<TEntity>> filter
)
{
      return await GetManyAsync(session, filter).MapAsync(e =>
            e.FirstOrDefault().ToTask()
      );
}
```

Snippet 92

In Snippet 92, we can see the modified version of the `GetAsync` method. Since the `GetManyAsync` function returns the `Maybe` type object of the sequence of elements, we can map on the result and return the first or the default value. The resulting object will be wrapped in the `Maybe` type object and in case it is null, it will be empty.

In the infrastructure layer, the `Transaction` class implements the `ITransaction` interface defined in the core layer. It is then used in the API project which is part of the presentation layer.

```
public TTransactionResult TryTransaction<TTransactionResult>(
    Func<IClientSessionHandle, TTransactionResult> action
)
{
    using var session = _context.Client.StartSessionAsync();
    try
    {
        session.StartTransaction();
        var result = action(session);
        session.CommitTransaction();
        return result;
    }
    catch (Exception)
    {
        session.AbortTransaction();
        throw;
     }
}
```

Snippet 93

In Snippet 93, taken from the original STOCK code, we can see the `TryTransaction` method that serves as a wrapper around a set of related operations where, in case one operation fails, the whole transaction is aborted. It is used when executing background jobs and that is why this method is synchronous. The first issue we can notice in this example is the try-catch block and the second even bigger issue is that this method is dishonest. It handles the exception only so it can abort the transaction and then it rethrows the exception. It is much better to return the `Exc` type object and handle this operation in a functional programming way. Another detail to notice is that the `IClientSessionHandle` object (MongoDB specific) needs to be defined outside of the try-catch block so it can be accessed in the catch block as well. We can improve that as well using the `DisposeAfter` function provided by Funk.

```
public Exc<TTransactionResult, Exception> TryTransaction<TTransactionResult>(
    Func<IClientSessionHandle, TTransactionResult> action
)
{
    return _context.Client.StartSession().DisposeAfter(s =>
    {
        var result = Exc.Create<TTransactionResult, Exception>(_ =>
        {
            s.StartTransaction();
            return action(s);
        });
        exc.IsFailure.Match(f => s.CommitTransaction(), f => s.AbortTransaction());
        return result;
    });
}
```

Snippet 94

In Snippet 94, we can see the modified function which is now honest as it notifies the user what exactly might happen. In the `DisposeAfter` function context, we are executing the operation wrapped with the `Exc.Create` function. Then, we are pattern-matching on the `IsFailure` property. In case the operation failed, we are aborting the transaction. We simplified the code, used fewer lines, and made the implementation much more powerful.

We can see from these examples how Funk improves the readability of the code and makes abstractions even more powerful. The overall size of the STOCK backend code is reduced by roughly 30 percent. Parts of the code that served as helper functions to abstract away the exception handling and null checking were completely removed as they were not necessary anymore. The overall performance of the application was not impacted by this refactoring as most of the processing tasks including machine learning model building happens in the background and is done by background jobs.

We can see how we can use functional programming concepts that give us power, safety, and clarity and still have a performant application if designed properly. In the next chapter, we will see related work regarding functional programming concepts in C# and the comparison with the Funk library. We will also look at some of the new functional programming features available in C# 8.

7. Related Work and Comparison

Functional programming is a programming paradigm that is becoming more popular every year. Object-oriented programming languages are embracing and introducing concepts from functional programming languages because of the power, safety, and clarity that they provide. However, the introduction of these concepts is happening slowly and usually only some concepts happen to be introduced. Therefore, the community is usually one step ahead with new ideas and suggestions that get implemented in the form of libraries and frameworks that enable working with these concepts almost as if they were native to the programming language.

Since C# is a multi-paradigm programming language with good support for functional programming, the community has done some interesting work implementing these concepts similar to what we did in the Funk library. Additionally, since the C# team noticed a late interest and the need for these concepts, the latest versions of C# are now standing between object-oriented and functional programming.

In this chapter, some features from the FuncSharp (Široký, 2015) and the Language-ext (Louth, 2014) functional programming libraries for C# will be described and the overall set of their features will be compared with the features provided by the Funk library. Additionally, some functional programming features from the latest version of C# (version 8) will be analyzed.

7.1. FuncSharp

FuncSharp is a functional programming library for C# created by Jan Široký. It is a simple yet powerful set of functional programming concepts. The project started in 2015 and now FuncSharp is already a mature library.

FuncSharp provides the `Option` type for handling the possible absence of data. There are similar functions for the `Option` type as the ones described for the `Maybe` type from the Funk library. Mapping and binding functions are also available for the `Option` type as the `Option` type is a monad. However, there are some differences between the `Option` in FuncSharp type and the `Maybe` type in Funk. The `Option` type is a `class` and therefore can be null whereas the `Maybe` type is a `struct` and cannot be null. We found that this difference is a big advantage in favor of the Funk library as it introduces additional safety. Moreover, the FuncSharp library does not provide asynchronous functions for the `Option` type where mapping and binding operations in the asynchronous environment are not possible.

FuncSharp provides a `Coproduct` type which is similar to the `OneOf` type that the Funk library provides. The difference is that the `Coproduct` type is a type constructor with the larger arity (up to 19 arguments) whereas the `OneOf` type is currently limited to 5. However, the advantage of using the `OneOf` type is because it handles the possible absence of data whereas the `Coproduct` type does not. The `Coproduct` type is a base type for the `Option` type.

FuncSharp also provides a `Product` type that is similar to the `Record` type that the Funk library provides. Another difference is that the `Product` type is a type constructor with the larger arity (up to 19 arguments) whereas the `Record` type is currently limited to 5. The difference between them is that the `Product` type is a reference type whereas the `Record` is a value type and therefore cannot be null. Additionally, they are targeting different issues. The `Product` type is trying to make the creation of the immutable reference types more straightforward whereas the `Record` type is intended to replace the `ValueTuple` type.

Another type available in the FuncSharp library is the `Try` monad. It is similar to the `Exc` type available in the Funk library. However, the `Exc` type is more powerful as it provides asynchronous functions and handles the possible absence of data. The `Exc` type is using the `EnumerableException` provided by Funk whereas the `Try` type from FuncSharp is acting more like an *Either* type from the Haskell programming language where the failure does not necessarily need to be the `Exception` type object. Moreover, the `Exc` type provides fallback mapping and binding functions whereas the `Try` type does not.

Finally, the Funk library provides more powerful pattern-matching capabilities as it provides extension methods as well as the dedicated types for the lazy pattern-matching evaluation. It also provides various extension methods for native .NET types as well as the `Prelude` which are not available in the FuncSharp library.

7.2. Language-ext

Language-ext is a base class library for functional programming in C# created by Paul Louth. It is the most popular functional programming library for C#. Language-ext is split into more libraries that can be used individually (for code generation, benchmarking, F# interoperability), however, all of them depend on the `Language-ext.Core` library which implements the most important functional programming concepts. For the rest of this section, we will be talking about the core part of this library.

Language-ext is a library created for functional programmers who happen to be using C# out of necessity. It is a very complex library with a lot of advanced functional programming concepts that object-oriented programmers are not familiar with and can find very hard to understand. Language-ext is almost like a new language and it requires quite some time to learn it. It provides many more features than the Funk library. However, Funk provides some interesting extension methods that are not available in the Language-ext library (e.g. `IEnumerable` extensions). Additionally, Funk provides a much easier and straightforward way of handling exceptions as the Langauge-ext has two distinct types: `Validation` and `Try` (which is a `Delegate` type).

Funk on the other hand is created for C# developers who are more comfortable with the object-oriented programming paradigm. It is trying to make the life of C# developers much easier without the need for them to invest too much time learning advanced functional programming concepts. Funk is a library that is intended to be easily embraced by anyone familiar with basic functional programming concepts.

7.3. C# 8

The eight version of C# was released in September 2019. It is officially released as part of the .NET Core 3 but it can be used in earlier versions of the framework as well with some limitations. C# now stands in between object-oriented and functional programming because of the new functional programming features it introduced in the eight version and the possible direction it may take (C# 9 will bring mostly functional programming features). In this section, we will show how to work with the nullable reference types and the new pattern-matching that C# 8 brought.

```
public static User? Create(
    string username, string firstName, string lastName,
    string language, IEnumerable<Company> favoriteCompanies,
    IEnumerable<SearchedCompany> searchedCompanies
)
{
    try
    {
        var user = new User(username, firstName, lastName, language,
            favoriteCompanies, searchedCompanies
        );
        user.Validate();

        return user;
    }
    catch (ValidationException)
    {
        return null;
    }
}
```

Snippet 95

In Snippet 95 we can see a similar function to the one defined in Snippet 90. Instead of returning the `Exc` type object, we are returning a nullable `User` object. In case there is some validation error, the `User` object will be null. If we would try to do some operation on this object, we would get a warning that the object may be null similar to the warning for the nullable value type. Nullable reference types are defined as the usual reference types followed by the **?** as we can see from the return type of the `Create` method in Snippet 95. This feature is helpful to issue a warning about the possible null-reference exception. However, developers are not forced by the compiler in any way to resolve the nullable reference type objects. Instead, we could return a `Maybe` type object (instead of the `Exc` type object in this example) where the caller of the `Create` method would be forced to resolve the return value. This example shows the advantage of using the Funk library over the new feature of C# 8 for working with the possible absence of data.

Pattern-matching in C# 8 is a more powerful `switch` statement. There are different types of pattern-matching including the normal switch expressions, property patterns, tuple patterns, type patterns, and positional patterns.

```
var greeting = user switch
{
	{ FirstName: "Jane" } => "Hello Mrs. Doe",
	{ LastName: "Mouse" } => "Hello Mickey",
	_ => "Something else."
};
```

Snippet 96

In Snippet 96, we can see property based pattern-matching where we are pattern-matching on a specific property of the object that is being evaluated. This is a powerful feature that enables developers to avoid using statements and minimize the code size. The only disadvantage of native pattern-matching operations is that they are evaluated immediately. This issue can be addressed by creating functions instead of plain objects inside the pattern-matching operation and invoking the function when the result is needed.

As we can see from these examples, C# is evolving in the direction of the functional programming and we can only guess what interesting and powerful new features are coming in the future versions of C#.

8. Future Plans and Conclusion

This chapter will mention the current state of the Funk project with features planned for the future and conclude with final remarks.

8.1. Funk in .NET community

Funk is an open-source library available as the NuGet package at https://www.nuget.org/packages/Funk. NuGet is an open-source package manager from where developers can get different kinds of useful libraries to use in their .NET projects.

Developers can get the Funk library by adding a reference directly into their C# projects or through .NET Command Line Interface (CLI) or using a Package Manager Console (PMC) which is available in Visual Studio or JetBrains Rider. Funk has been downloaded from NuGet more than a thousand times as of July 2020.

Funk is also available on GitHub at https://github.com/hcerim/Funk. GitHub is a development platform where developers can cooperate on projects, look into the content of other public repositories, follow other developers, and watch repositories for the latest changes. Developers can also cooperate and contribute to projects, therefore we are welcoming other developers who are interested in contributing to the Funk library to make it even better in the future.

Funk is licensed under the MIT license.

Funk is under active development where currently mostly performance issues are addressed. Funk is developed as part of this thesis as a proof of concept with a rich set of features. More functional programming concepts are planned in the future that include current types with larger arity of arguments, property-based pattern matching, memoization, current types with lazy evaluation, I/O monads, F# interoperability, etc.

8.2. Final remarks

Working on this thesis and the Funk library project has been a wonderful experience where we discovered and learned interesting concepts that most C# developers are not familiar with. Switching from an object-oriented to the functional programming state of mind in an object-oriented programming language was not easy. Functional programming concepts are not related to any specific programming language but to a paradigm that implies that certain rules need to be followed to implement these concepts properly.

Even in future versions of C#, there will still be issues that the Funk library will be able to address as C# will always stay object-oriented programming language and certain issues of the paradigm will stay as they are now.

While developing the Funk library, we discovered that C# is not only a powerful but also a beautiful programming language that enables great expressiveness and fluency while still being simple and easily understandable.

Funk is a C# library devoted to any developer seeking for power, safety, and clarity inside their codebase while still requiring performant programs.

Bibliography

Lane, S. M. (1971). *Categories for the Working Mathematician.* Springer.

Hoare, T. (2009). *The Billion Dollar Mistake*. *QCon.* London.

Martin, R. C. (2006). *Agile Principles, Patterns, and Practices in C#.* Pearson Education.

Milewski, B. (2019). *Category Theory for Programmers.*

Bounanno, E. (2018). *Functional Programming in C#.* Manning Publications.

Wlaschin, S. (2014). *Railway-oriented programming*. *NDC.* London.

Albahari, J. (-). *Why LINQ beats SQL.*

Lippert, E. (2011). *Following the Pattern.*

Wlaschin, S. (2015). *Elevated World.*

Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* O'Reilly.

Široký, J. (2015-present). *FuncSharp*. https://github.com/siroky/FuncSharp.

Louth, P. (2014-present) *Language-ext*. https://github.com/louthy/language-ext.

List of Figures

List of Snippets

List of Tables