

## White Paper



### Functional Programming on the .NET Platform

---

Chandrasekhar Prabala

#### Abstract

In this paper we will look at how concepts of functional programming have been implemented on the .NET framework. The emphasis of this paper is FSharp (F#,) a new functional programming language on the .NET framework. This paper discusses the idea of functional programming and few concepts of the paradigm in a language agnostic way. Discussions then move on to how the ideas of functional programming have crept into C# which enabled the creation of LINQ. Finally, the paper concludes by pointing out where languages like F# can be used and where the whole idea fits in current circumstances.

## Perils of Imperative programming:

An imperative programming language is one in which both what and how the program should accomplish a task, is explicitly specified. Examples of imperative programming languages include C, C++, Java, C# etc. These programming languages are ridden with what are known as side effects. A side effect introduces a dependency between the global state of the program and the local state of a function thus having a detrimental impact on its behavior.

Consider the example of a global variable. Any function can manipulate the global variable. Thus, a function using the global variable is not wholly sure if the global variable has the intended value or not. Thus, the success or failure of the function in producing the desired output depends upon the global state, though the function itself never manipulates it. A function that has a side effect is called an impure function. In functional programming languages, care is taken to contain these side effects. The idea is to eliminate the usage of shared state as much as possible. Every function that's written in a functional programming language guarantees to return the same output every single time the same input is passed to it. State here is not maintained in variables but is maintained in the functions themselves as parameters, on the stack. The function depends only on the inputs given to it and nothing else. In that sense, functions here are said to be pure. Thus, there are no data races or reader/writer problems.

In an imperative programming language values are mutable. A variable  $x$  can hold different values in the same scope depending on the order in which values have been assigned to it. So every time you use  $x$  with a function you are not sure of the output because  $x$  is mutable. In a functional programming language values are bound to symbols in the form of expressions. Once a value is bound to a symbol, its value never changes. It's immutable in that sense. One can always substitute the expression with the value. This property in functional programming is known as referential transparency. In an imperative programming language this property does not hold.

The previous few sentences use the word “expression” instead of the more commonly used word “statement” deliberately. Imperative programming languages work with statements and not expressions. The key difference is that a statement is not guaranteed to return a value where as an expression is. Thus, in an imperative programming language, functionality is driven by code and not by data. Why is there such an emphasis on purity in functional programming? The reason is that a lot of programming errors or security flaws in code occur due to the presence of side effects. In functional programming languages things like printing to a file or console, accessing a network or a database etc., are separated and contained from code that does business logic, which is inherently pure and modular. The result is a neat architecture and easier debugging and testing.

## Functional Programming

Functional programming is the name of a programming paradigm which considers programming as the art of composing pure mathematical functions together. Thus, a function is the basic unit in this paradigm. Even the simplest of operations are modeled as functions. Functional programming like OOP is a set of ideas. These ideas are composed of a set of principles, techniques and patterns which when used correctly can lead to the creation of compact and highly reusable code. In the following sections a few of those ideas will be shown. Since no functional programming language has been introduced as such, examples in the following sections shall be kept language syntax agnostic.

Before touching on a few of these concepts, let's understand what is a function.

Function is a correspondence relation between source and a target.

$$f :: A \rightarrow B$$

indicates that the function  $f$  takes inputs in  $A$  and returns output in  $B$ . Here,  $A$  and  $B$  are generic types. When we say  $f(x)$  or  $f\ x$ , we mean we are applying the function to argument  $x$ . The ‘ $\rightarrow$ ’ operator can be read as “goes to”. The interesting thing to note here is that it's not just  $x$  and  $f\ x$  that have types  $A$  and  $B$  respectively, the function  $f$  has a type  $A \rightarrow B$ . Thus, in functional programming, functions are treated as values and vice versa. In principle there is no difference between either of them. As an example the following function `add` takes two parameters and returns the result which is their sum.

```
add a b = a + b
The type of this function is
add :: num -> num -> num
```

(assuming that + is not overloaded for strings and types other than numeric types such as int, float, double etc.)

Given a function such as add, how is the type of the function decided? The right hand side of add has the operator +. (+) has the type num -> num -> num. Thus, a, b and add get their respective types. If there is a function (.) f g x = f(g x) then the type is decided in the following way. Note that function application is right associative.

```
f :: (A->B)
g::(C->D)
x :: (E)
Thus, (.) has the following type.
(.) :: (A->B) -> (C -> D) -> E -> F
```

This method of type deduction is often called as Automatic type inference and is a feature of every functional programming language and even a few imperative programming languages like Python. C# has recently made automatic type inference as its part when it added the var type. This particular feature was introduced in C# with the 3.0 version of the language.

## Currying

The function add that was shown in the previous section had a type num -> num -> num.

The operator '->' is right associative. Thus, add type can be re written as num -> (num -> num). This means that the add functions takes a numerical value and gives as result a function that has type num -> num. Let us name that function curryFunction. Thus,

```
curryFunction b = add 3
curryFunction 4 = 3 + 4
                =7
```

(add 1) now is the successor function. (add (-1))is the predecessor function, if we consider integer values. If this is not intuitive in the first go, this is in some ways similar to the adapter pattern that's used with OOP languages.

Currying is of great use while designing APIs. It lets the specification of optional parameters succinctly.

## Higher Order Functions

Functions which take other functions as input values are known as higher order functions. Higher order functions are a mechanism by which abstraction is achieved in functional programming languages. Consider the example of d/dx(x<sup>2</sup> +3) the result is 2x which is also a function. Thus d/dx here is a higher order function which takes another function x<sup>2</sup> + 3 as its parameter and returns another function. This concept can be explained with the analogy of delegates in C#. Delegates treat methods as data. Since, in functional languages every function is nothing but a value, a function can be passed as a parameter to another function thus creating neat abstractions. Higher order functions are useful in capturing common programming patterns.

```
map :: (A -> B) -> [A] -> [B]
```

Here, map is an example of a higher order function. The function map takes another function and a list as input and applies the function over all elements of the list and returns the resulting list.

## Lazy Evaluation

Generally, compilers adopt a reduction strategy in order to compute. Reduction strategies are of two kinds:

1. Applicative order evaluation or Eager Evaluation
2. Normal order evaluation or Lazy Evaluation

Consider the following example:

```
sum_of_squares a b = square (a+1) + square (b+1)
square a = a * a
sum_of_squares 4 5 can be reduced in multiple ways. One way is as follows
sum_of_square 4 5 = square (4+1) + square (5+1)
                  = square 5 + square 6
                  = 5*5 + 6*6
                  = 25 +36
                  = 61
```

(OR)

```
sum_of_square 4 5 = square (4+1) + square (5+1)
                  = (4+1)*(4+1) + (5+1)*(5+1)
                  = 5*5 + 6*6
                  = 25 + 36 = 61
```

There is a subtle difference in both the evaluation schemes. In the first case the reduction is an inner most reduction, the inner most values are evaluated first and then the evaluation proceeds outwards. Such an evaluation scheme is called Applicative order evaluation or eager evaluation. By contrast, in the second case, the inner most values are not evaluated as long as they are required. Here, evaluation proceeds from outside inwards. This kind of evaluation scheme is called Normal order or Lazy evaluation. Though, in this example the result is same in both the cases, this may not be true everywhere. Consider the following function

```
fst (a,b) = a
```

Here, `fst` is a function which gives the first element in a tuple. Consider the following example,

```
first = fst (42,loop)
loop = a loop
```

If applicative order evaluation is considered, the reduction would proceed from inwards to outside, giving,

```
fst (42, a loop)
fst (42, a (a loop).....)
```

Here `loop` is replaced with its definition 'a loop'. If normal order evaluation is considered, the reduction proceeds from outside to inwards, giving,

```
fst (42, a loop) = 42
```

Languages like C# usually follow applicative order evaluation. As a consequence, they need to treat operators like `||` (or) different from the rest. In many functional languages this is not the case as they use normal order evaluation scheme.

There are advantages and disadvantages with the usage of the lazy scheme of evaluation. Lazy evaluation scheme makes code optimizations easy, it lets one implement infinite data structures and gives a nice abstraction. However, in cases where there is a strict evaluation order required, in places where code needs to be executed sequentially, lazy evaluation does not give the intended results.

## Pattern Matching

Pattern matching is perhaps the most frequently used technique in functional programming. In appearance pattern matching is similar to a switch case statement used in C like languages. A pattern matcher is a piece of code which matches one of the specified patterns. The success or failure of a pattern match can be used to trigger the execution of different expressions accordingly. For example, a list [1; 2; 3; 4] matches the pattern a cons [num]. Here, cons is an operator which adds an element to the head of the list. [num] means a list that has a numeric type. The pattern match results in the assignment of 1 to a and [2;3;4] to [num]. The list also matches the pattern h cons t, where h is the head of the list and has a value 1 and t is the tail of the list [2;3;4]. The pattern can be thought of resembling a regular expression for the purpose of analogy though there is a huge difference between both of them in practice. It also resembles switch-case statement in imperative languages such as C, C++, C#, etc, and again there is a huge difference. Pattern matching is a very powerful tool and is used in many ways. A simple example of pattern matching usage can be summing the elements of a list. The following is a pseudo code that uses pattern matching to sum the elements of a list.

```
sum [] = []
sum [x:xs] = x + sum [xs]
```

## Tail Recursion

Recursion is a mechanism by which a function calls itself. In popular compiler implementations, recursive code consumes a lot of stack space and deeply recursive calls can lead to stack overflow. However, there is a technique of recursion in which the amount of stack space can be maintained at a constant however deep the recursive calls might be. Consider the following example

```
fact n =
  If n<=1 then 1
  else n * fact n-1
```

The above procedure calculates factorial for a number. However, as n becomes high the amount of space that the recursive procedure occupies on the stack also increases. Consider the following pseudo code now:

```
fact n =
  fact_tail p prod =
    if n<=1 then 1
    else fact_tail p-1 prod*n
  fact_tail n 1
```

In the above pseudo code, the result of each iteration is being passed as a value to the next iteration and there is no need for the compiler to maintain the same in the stack. Thus, the amount of space needed on the stack is equal to the amount of space that needs to be allocated to the symbols like p and prod and no more. Thus, the algorithms space complexity is constant.

Such a recursive scheme is called a tail recursion. In imperative languages like C#, Java etc., the compilers inherent disability to support tail recursions, makes the language incorporate in itself in loops like, for, while, etc., to achieve the same objective. Thus, they are only syntactic sugar. A functional programming language doesn't share such a scheme.

## Continuations

Continuations are functions that receive the result of computation of another expression as their argument. Consider add and square functions from the previous sections.

```
s = add 3 5
8
square s
64
```

When the add function was called, the function internally evaluated the sum and returned the result to the point where the function was called. The function returns to the place where it was called because internally there is a pointer on the stack which indicates where the function should return its value. It is thus not difficult to make the function return the result of its execution to a valid but different location. This is what continuations achieve. Thus I can rewrite the above pseudo code in the following way.

```
S = add 3 5, square
64
```

In this case, add 3 5 was evaluated and the result of the evaluation is passed as argument to square. Thus square is a continuation here. Continuations are a very useful tool in achieving abstraction and also in writing code that is neat and succinct. Also, tail call optimizations can be done well using continuations.

## Closures

Pure functional programming puts a lot of restrictions on building programs that do lot more work with less effort and requires the introduction of few imperative constructs though they are not wholly welcome. One such feature is the concept of closures. Closure is a mechanism by which a procedure can access the state outside its local scope. Thus, closures are said to encapsulate a function and its environment. Consider the following example:

```
ex_fun =
  a = 10
  ex_fun2 =
    a = 20
```

In the above example the function ex\_fun2 redefines the value of a, which was originally defined in the scope of function ex\_fun, thereby creating a closure. Closures is a powerful technique but should be used with a lot of care. It is heavier than a function and can introduce additional complexity.

## And a few more

What is given above is by no means an exhaustive list of the features. There is lambda calculus and category theory which are branches of computational mathematics which form the basis of functional programming. There are monads and its implementations in many senses which led to the creation of functional concurrency, something which a language like Erlang or its application to SQL uses, which resulted in LINQ and many more. In pure functional languages monads are the way by which variable mutations, I/O etc... are implemented. These topics deserve individual papers in themselves and are not being covered here. One can always use their favorite search engine to find more about these.

## F#

F# pronounced as “FSharp” is a functional programming language on the .NET framework. F# has been included as part of the VSTS 2010 release, thus making it a first class language like C# and VB.NET on the .NET framework. F# has a syntax that is in many ways close to the language Scheme but the language features are inspired from CAML family of languages. The language is a decent mix of constructs both from the functional and the OO world, thus can be appealing to both the communities. The language is both static and strongly typed. It has automatic type inference facilities making coding all the more easier. In the next few paragraphs F# specific features will be discussed. By no means is this list exhaustive. But it is aimed at giving the reader a firsthand overview of the language.

A function or a value in F# is defined using the ‘let’ keyword.

```
let x = 10
let add a b = a + b
```

While, there is the facility of automatic type inference, one can also constrain a value to a specific type. The following function square has its parameter constrained to accept only floating point values.

```
let square ( x:float)
```

Similar to let binding, there is also 'use' binding which disposes off resources after use. This is similar to using syntax in C#.

```
use stream = System.IO.File.Open filename
```

Tuples are the simplest form of compound types in F#. A tuple can be created in the following way

```
let t = (1,2,3)
```

The tuple shown in the above example has 3 values in it. However, a tuple can have between 2 to any number of values. A tuple that has 2 values is called a 2-tuple. A tuple that has 3 values is called a 3-tuple or triple. A tuple that has more than 3 values is referred to as an n-tuple.

A tuple has values but there are no names by which one could refer to those values. A tuple that has names for its fields is known as record.

```
let name = {FirstName:String; LastName:String}
```

A record can be used in the following way.

```
let primeminister = {FirstName = "Manmohan"; LastName = "Singh"}
```

Tuples and Records are statically typed. Their types are known during compile time. It is possible to define types which acquire one of several possible types at runtime. Such types are known as variants and are close relatives of the union type in C/C++.

```
type Option = Some | None
```

The type option above can become either some or none at runtime. These can be used to create pretty complex data structures very easily like the below example which declares a binary tree. A bit of recursion is also involved in here.

```
type binary_tree =  
    | Leaf  
    | Node of binary_tree * binary_tree
```

For the purpose of decision making, F# provides an if construct. The syntax for it is as follows:

If *expr1* then *expr2*

If *expr1* then *expr2* else *expr3*

In functional programming language everything is a value. Hence, every expression is a value. Hence, the 'if expression' should evaluate to a value. In the above example, *expr1* in both the cases should evaluate to bool. *expr2* in the first case should necessarily evaluate to unit (), which is a type which is equivalent to void in C#. The reason is that if it did evaluate to something else other than void then what would happen if *expr1* returned false. In the second case *expr1* and *expr2* should return a value of the same type. The reason is, if they could return different types then what would you account to the type of the entire expression?

Functions always need not have names. A function can be declared anonymously as well. In such cases we use lambda expressions. Below is an example. 'fun' is the keyword which conveys that sense.

```
fun x -> x * x
```

Pattern matching has the following syntax in F#.

```
Match expr with  
|pattn1 -> expr1  
|pattn2 -> expr2  
.  
.  
|pattn -> exprn
```

Here is an example that uses pattern matching to calculate the Fibonacci number

```
let fib n =  
    match n with  
    | 0 -> 1  
    | 1 -> 1  
    | _ -> fib (n-1) + fib (n-2)
```

Exception handling in F# is implemented with the help of the pattern matching syntax an example is shown below.

```
try  
    raise (Failure "My problem1")  
with  
| Failure s -> s
```

Sequence expressions are used to generate a continuous sequence of integers and characters. The following is an example

```
seq{1..100}
```

In pure functional programming, values are immutable. Thus, once you have assigned something to a value, it never changes. However, F# also provides facilities for imperative programming. One will have to use the mutable key word to achieve that. If this keyword is used the value can be reset.

```
let mutable x = 6  
x <- 8
```

Where imperative programming languages use flower brackets to signal the start and end of a block F# uses space. Thus, space has a lot of meaning in F# and should be used carefully. The following example shows two functions defined.

```
let func1 =  
    let x = 5  
    let func2 =  
        let y = 6  
    let z = 9
```

In order to bring demarcation and to organize the code, F# uses modules. This is comparable to namespaces in C#. An example is shown below.

```
module Employee =  
    let name = {FirstName:String; LastName:String}  
  
    let findAge =  
        System.DateTime.Now.Year - date_of_birth.year  
    .  
    .
```

It being a functional programming language, F# has very nice ways to handle functions. The facilities that it provides to this are known as combinators.

$f \ll g$  composes the functions  $f$  and  $g$  so that its equivalent to  $f(g(x))$

$f \gg g$  composes the functions  $f$  and  $g$  so that its equivalent to  $g(f(x))$

$x \mid> f$  is a function application which means  $f x$ .  $\mid>$  is also known as the pass forward operator.

List is the primary data structure that's used in F#. A list has the structure shown below.

```
[1;2;3;4;5]
```

To a list we add data to its head. The operator which adds a single element to the list is called cons and is represented by  $::$  in F#

```
6 :: [1;2;3;4;5]
```

All elements in the list should be of the same type.

An array has a structure similar to that of a list with a slight difference. It's as shown below.

```
[|1;2;3;4;5|]
```

There are many operations that can be performed on a list, sequence and an array. But, there are two operations that are perhaps the most important of the lot. They are map and fold.

The map function applies a given function to every element in the list and gives out a new list. An example is shown below.

```
[1;2;3;4] |> List.map (fun x -> x*x)
```

As a result of the application of map the result that's produced is  $[1;4;9;16]$ .

A fold is an operation which takes every element of a list and applies a function to it threading an accumulator. What this means is that function is applied to each element of the list and the resulting values are added to each other generating a result. The following code snippet calculates the factorial for a given number

```
let fact n =  
    [1..n] |> List.fold (*) 1
```

In addition to these primitive tools F# can be used for language oriented programming. The idea of language oriented programming is to solve programming problems by using simple languages that target a specific problem domain called domain specific languages rather than using multiple APIs. This brings in more flexibility and ease of use. The tools that F# has for this purpose are fslex and fsyacc. These are used to decipher the content of a file using formal grammar. The examples for fslex and fsyacc are a bit verbose and the reader is requested to look for examples online.

F# also has support for asynchronous and concurrent programming using the asynchronous workflow syntax. Asynchronous workflows are the same as monads in Haskell. Asynchronous workflows allow a programmer to run a task in the background without hurting the responsiveness of the application. Asynchronous workflows allow parallelism to be brought into F#. The following is an example that uses F# asynchronous workflows to run two tasks in parallel.

```
let task1 = async {
    [1..50] |> List.fold (*) 1
}

let task2 = async {
    let! lst1 = [1..1000000] |> List.map (fun x -> (x,x+2))
    let! lst2 = [1..100] |> List.fold (+) 0
}

let result = [task1;task2] |> Async.Parallel |> Async.RunSynchronously
```

let!, pronounced 'let bang' Indicates that this operation should be allowed to run in the background and execution of the next operation can be continued.

## Final words on F#

F# is as much a general purpose language as any of its imperative counterparts like Java and C#. Its ability to support both functional and OO constructs give it a significant edge over other functional languages like Lisp or Haskell. F# can be used every where C# is being used. Having said that one would realize the true power of F# when used in areas where there is lot of mathematics and computation involved. The notion of functions in the language makes it a natural choice for solving such programming problems. In realization of the style of programming using DSLs, F#, as has been written above, has features that create lexers and parsers. Programs written using functions that are self contained and are pure are easily amenable to parallelism. If an application has a need wherein it has to be asynchronous and parallel, F# can get you there pretty quickly. Currently, F# is being used in many places to build libraries that do interesting work. If you have a big application, use your favorite imperative programming language to build the outer architecture of your application and do the computations internally using F#.

## C# and Linq

Functional programming is one of the oldest paradigms of programming. Thus, undoubtedly, this paradigm has influenced many a programming languages. In this section we will look at the influence functional programming has had on C#. C# 2.0 introduced the concept of generics. This concept in C# closely relates to parametric polymorphism in Haskell and other similar functional programming languages. Not just C#, the type variables in Java generics and templates in C++ also are created influenced by the concept of parametric polymorphism. C# 3.0 introduced several new concepts that are found in a functional programming language.

The var keyword allows one to declare a type anonymously. The automatic type inference feature which is also introduced along with C# 3.0 figures out the type of the value at compile time. An example is shown below.

```
var x = 10
```

Along with it support for lambda expressions has been added. One can write a lambda expression where one would normally write an anonymous function. An example is shown below.

```
Func<int, int, int> function = (a, b) => a + b;
```

A new construct known as expression trees was added. Expression trees allow one to treat code as data. An example is shown below.

```
Expression<Func<int, int, int>> expression = (a, b) => a + b;  
  
Console.WriteLine("Param 1:{0} Param 2:{1}",  
expression.Parameters[0], expression.Parameters[1]);
```

Along with what's mentioned a few others were added. All these were added to facilitate the implementation of LINQ (Language integrated query). LINQ is a language feature that allows one to query and transform in memory collections, databases, xml files and many others alike thus bringing uniformity in data access mechanisms. LINQ is the implementation of monad comprehensions that is found in Haskell to SQL. A trivial example that uses LINQ is shown below.

```
int[] integerArray = new int[] { 1, 2, 3 };  
  
IEnumerable<int> num = integerArray.Select(i => i);  
foreach (int item in x)  
{  
    Console.WriteLine(item);  
}
```

## Conclusion

Functional programming though an old idea has emerged as the new kid on the block. As writing software is becoming complex and as software is trying to solve newer problems, conventional OO languages may not be enough. Functional programming or tools created using functional programming languages is all set to fill that void and Microsoft's .NET platform is the place where lot of innovation is happening. A large number of people are beginning to embrace this style of programming. Hope this paper has provided you some insight into this world.

## References

1. Introduction to Functional Programming 1st edition, Richard Bird, Philip Wadler
2. Expert F#, Don syme
3. F# for scientists, Dr. Jon Harrop

## About the Author

Chandrasekhar Prabala is a senior software engineer in the High performance computing (HPC) group with Infosys' Microsoft Technology center (MTC). He has more than 2 ½ years of software development experience in F#, C#, ASP.NET and WCF.

## Acknowledgements

I profusely thank S. Naveen Kumar (Principal Architect, MTC) and Sudhanshu M. Hate (Senior Technical Architect, MTC) for their support and valuable guidance in bringing out the paper in the current form.

I would also like to thank Atul Gupta (Principal Architect, MTC) and Sripriya Thothadri (Technical Architect, MTC) for helping out at various stages during the review process.



For more information, contact [askus@infosys.com](mailto:askus@infosys.com)

#### About Infosys

Many of the world's most successful organizations rely on Infosys to deliver measurable business value. Infosys provides business consulting, technology, engineering and outsourcing services to help clients in over 30 countries build tomorrow's enterprise.

For more information about Infosys (NASDAQ:INFY), visit [www.infosys.com](http://www.infosys.com).