# FC#: Designing an Internal Functional DSL for C# 3.0*

Krisztián Pócza, Mihály Biczó, Zoltán Porkoláb

Eötvös Loránd University, Fac. of Informatics, Dept. of Programming Lang. and Compilers,
Pázmány Péter sétány 1/c. H-1117, Budapest, Hungary
kpocza@kpocza.net, mihaly.biczo@t-online.hu, gsd@elte.hu

**Abstract.** Based on the improvements of the C# programming language towards functional programming support, and motivated by the FC++ functional library for C++, we introduce the FC# functional library for C#. FC# itself is an internal Domain Specific Language (DSL) for C#, therefore solutions created using FC# can be embedded into native C#. FC# has a couple of useful features for programmers who like functional concepts or who write multi-paradigm programs. The most important features we support are lazy lists, basic list filtering and composition operations, high performance, extensibility, and easy integration with C#. To achieve it the following capabilities of the C# 3.0 language are exploited: enumerators, lambda expressions, type inference, currying, and extension methods. Beside expressiveness and high level of usability, FC# has also an efficient implementation that we also show in the performance comparison charts. The latest version of FC# can be downloaded from the following URL: http://www.codeplex.com/fcs/.

## 1 Introduction

The most widespread programming languages [19] (C, C++, Java, C#, Visual Basic, PHP, etc.) that are used by everyday programmers are mostly object oriented languages. It is hard to explain why functional programming languages [18] are not so popular for developing user or business applications, but it may be derived from the following facts:

1. Functional programming languages provide higher level of abstraction that an average programmer can understand
2. Functional programming is not educated adequately
3. Legacy applications were created in OO languages

Applications are getting more and more complex therefore it is important to structure them well. The most important structuring tool is modularization. Functional programming languages are very good at modularization by design [4, 5].

The higher level abstraction a programming language has, the easier it is to model real world problems in it. Moreover, the concepts used in functional languages are very close to human thinking and everyday problems. For example MapReduce [2] developed by Google is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes

* Supported by GVOP-3.2.2.-2004-07-0005/3.0

a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key.

Fortunately, the following trend can be observed in the industry as well:

1. Object oriented programming languages are getting more and more language elements that originates from functional programming
2. There are some functional programming languages that have gained significant popularity (e.g. Lisp, F# and Erlang) recently

It is possible and encouraged to write functional-style programs in object oriented languages [1, 15] (The reverse can also happen [7].) Informally, it means that functions should be written to accept parameters and to work by creating new objects and returning them so that side effects would be minimized. No global variables should be used and the usage of local variables (that store the internal state) should be kept to a minimum. The object oriented programming model encourages capturing data and functions to a class therefore functions depend on the internal state of the class and manipulate the internal state therefore introduce side effects.

Maybe C# is the programming language that is developing the most dynamically towards functional outlook therefore gaining advance in expressiveness [6]. Over the last 3 years the version 2.0 and version 3.0 of C# got many functional extensions that raised the expressiveness therefore reduced development time and cost. These concepts are: lambda expressions, closures, laziness, type inference, extension methods, currying.

It is LINQ (Language INtegrated Query) that already combines the above features however the syntax of LINQ reminds rather SQL than a functional language. LINQ can be thought of as a generic querying DSL [9, 16] that is able to operate over collections (LINQ over Collections), XML (LINQ to XML) or even data stored in a SQL Server (LINQ to SQL).

The FC++ library [8] has given us the main motivation to analyze the possibilities of designing a library that has similar functionality and concepts, however FC# enhances C# rather than C++. The FC++ library provides functional-like syntax by overloading the function application operator, "()" however C# does not allow it. Therefore we have chosen class inheritance and extension methods to simulate the same behavior.

In this article first we show the functional programming concepts of C# 3.0. After getting acquainted with these language elements and collecting the expectations to a functional library, an internal functional DSL called FC# will be presented. We show the services of the library and some common tasks will be implemented using it.

After the high level overview we dive deeper and using the functional toolset of C# 3.0 show the implementation details of the FC# functional library. The performance will be also measured against the F# [12, 17] implementation; the native and LINQ based implementation in C# and Clean.

In the last section the content and the message of this paper will be summarized and the current development directions will be reviewed.

The latest version of FC# can be downloaded from the following URL: http://www.codeplex.com/fcs/ [20].

## 2 Functional Programming Concepts in the C# 3.0 Language

Any application written in the C# language runs under the control of the Microsoft .NET Framework's Common Language Runtime, a managed environment for code execution that also takes care of garbage collection and runtime security.

The first version of the C# language did not know anything about functional programming. From the functional toolset we had only recursion and could create delegates [9] that are special kinds of objects derived from *System.Delegate* that is able to encapsulate a method or a sequence of methods. C# 1.0 did not even have generics. The most important innovation of C# 2.0 was generics, anonymous delegates, and the yield keyword. C# generics are similar to Java or Ada generics while anonymous delegates are some kind of inline delegate methods. The yield keyword is syntactic sugar that makes it easier to create enumerable types. The version 3.0 of the C# language introduced lambda expressions [10] that offer an easier form of anonymous delegates from the C# point of view. The other important features are type inference, extension methods, and currying. Type inference is the ability to automatically deduce the type of a value from the usage. Extension methods enable the programmer to add methods to an existing type without having to derive from it.

### 2.1 Recursion

This is a feature that does not need too much explanation and almost every programming language has it. Unlike many functional environments, C# does not handle tail recursion. This might lead to stack overflows. However, there is a supported IL instruction called *tail* therefore there is a possibility for language designers to add support for tail recursion to the C# language.

### 2.2 Delegates – Lambda Expressions – Closures

The biggest improvement of the C# 2.0 language was generics. However, there were less important, but interesting new features as well. One of them is anonymous methods. This feature allows programmers to create un-named or so-called anonymous methods. A similar concept of anonymous classes has earlier been introduced in the Java programming language. Java does not define the notion of function pointers; it operates with interfaces and their implementation that is why the level of anonymity is that of the class.

Function pointers in C# are called delegates, however they are not simply the C++ equivalents of function pointers. Delegates are strongly typed and might contain more than one function reference, so it is more appropriate to think of them as strongly typed function pointer arrays.

```
List<string> names = new List<string>(new string[] {
           "Clean", "Haskell", "C#", "LISP", "C++" });

public List<string> FilterRecordsWithDelegate()
{
```

```
   return names.FindAll(StartsWithC);
}

private bool StartsWithC(string name)
{
    return name.StartsWith("C");
}
```

In the above Listing a method is defined to decide whether a string starts with the letter ‘C’. A list of strings is defined, and we pass our delegate (the predicate) to the FindAll method of the generic List class.

In case of simple functions whose code is unlikely to be reused, it is inconvenient to define a separate function. C# 2.0 allows delegates to be defined without naming them explicitly. The *FilterRecordsWithDelegate* function can be rewritten as follows using an anonymous delegate:

```
public List<string>FilterRecordsWithAnonymousDelegate()
{
  return names.FindAll(delegate(string name) {
                   return name.StartsWith("C"); });
}
```

The *StartsWithC* function is defined in-place without assigning an explicit name to it. Anonymous delegates are handy when a simple sort or find operation is implemented that does not deserve a dedicated function. However, seemingly they are against the readability and reusability of the code.

In C# 3.0, lambda expressions are a nice replacement of anonymous methods in terms of simplicity of syntax. Using a lambda expression the *FilterRecordsWithAnonymousDelegate* method can be replaced with the following:

```
public List<string> FilterRecordsWithLambdaExpression()
{
  return names.FindAll(name => name.StartsWith("C"));
}
```

The syntax resembles the syntax of the functional language called Haskell [18].

Anonymous delegates of C# 2.0 and lambda expressions of C# 3.0 are able to capture variables/information from the environment under which they have been created. Shortly the usage of closures is allowed.

The *FilterRecordsWithLambdaExpression* method can be replaced with the following (showing closures in action):

```
public List<string> FilterRecordsWithClosure(string start)
{
  return names.FindAll(name => name.StartsWith(start));
}
```

### 2.3 Laziness

With lazy evaluation an expression is only computed when it is certain that its value is really needed for the result. The opposite of lazy evaluation is eager evaluation where

the function result is computed as soon as the actual parameters are known. With the help of lazy evaluation performance increases due to avoiding unnecessary calculations and there is the ability to construct infinite data structures.

In the following two different methods of lazy evaluation will be discussed:

1. Lazy parameter passing
2. Lazy data structures – lazy lists

### 2.3.1 Lazy parameter passing

In the next example various variations of a simple *IsNull** function with two parameter will be shown that returns its first argument when it is not null otherwise the second.

The first variation uses eager evaluation meaning that the first and the second parameter is also evaluated prior call.

```
string IsNullEager(string first, string second)
{
    return first != null ? first : second;
}
// Method call
IsNullEager(val, longTimeOperation());
```

When the second parameter is a long time operation as the example suggests it is not worth calculating when not needed.

The lazy version of the same function looks like the following:

```
string IsNullLazy(Func<string> first, Func<string> second)
{
    return first() != null ? first() : second();
}
// Method call
IsNullLazy(() => val, () => longTimeOperation());
```

In this example the lazy evaluation is useful because it prevents *longTimeOperation()* from being calculated when the *first()* function returns not null. However it is important to mention that the value of lazily evaluated functions is not preserved for subsequent calls (C# uses call-by-name parameter passing in that case) therefore in the above example the *first()* function will be evaluated twice when it returns not null. A possible workaround to this problem would be to introduce a variable, however, that can cause side effects. Therefore the suggested solution for the above case is to use the "??" operator that returns the left-hand operand if it is not null, or else it returns the right-hand operand.

### 2.3.2 Lazy data structures

The most important implementations of lazy data structures are lazy lists and even infinite lists that can be simulated using iterators in C#. Most often iterators are used for traversing through the elements of a collection by using a method that advances to the next element in the collection and another method that returns the current method possibly modified after some operation. This behavior is side-effect free. However iterators can also be used to create value sequences from scratch.

C# 2.0 introduced a new keyword called *yield* [9, 11] that is a coroutine that can

pause in the middle of an iteration loop and return a value. When you call it again it picks up right where it was when you paused, environment intact, and keeps on going. Yield is implemented using an FSM behind the scenes.

The first example in Section 5.2 shows a lazy infinite list implementation that is generic therefore should be parameterized by a concrete type. It is abstract that means it has to be inherited and its abstract methods should be implemented. As the example suggests iterators can be commanded to work by encapsulating the iteration loop in a *GetEnumerator* method that return *IEnumerator<T>*.

### 2.4 Type Inference

Type inference or implicit typing is the ability to automatically deduce the type of a value from the usage. C# is a strongly typed programming language whose variables are statically typed. The support for type inference in C# 3.0 is very limited compared to functional languages but the language breaks fresh ground in case of mainstream languages: type inference can be used only with lambda expressions and local variables.

Please refer to the last two examples shown in section 2.2 (lambda expression and closure). Both examples use the name variable without explicitly defining the type of it. The usage of type inference for local variables can be seen in the example of section 2.5. The usage of *var* keyword instructs the compiler to automatically deduce the static type of the variable if it is possible.

### 2.5 Extension Methods

Extension methods are language enhancements in the C# 3.0 language specification that mimic as if objects could be extended with new methods in runtime. Although they are regular static methods, and can be called as common static methods, they can also be called as if they were pure object methods.

Take a look at the following example:

```
public static class ExtensionMethods
{
    public static bool IsValidEmailAddress(this string s)
    {
        var regex =
               new Regex(@"^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$");
        return regex.IsMatch(s);
    }
}
```

The IsValidEmailAddress method is a regular static method that accepts a string parameter and checks if it matches the given regular expression. The parameter is marked with the this keyword therefore the method can be called as IsValidEmailAddress(str) and as str.IsValidEmailAddress() also.

### 2.6 Currying

Currying is "the process of transforming a function that takes multiple arguments into a function that takes just a single argument and returns another function if any arguments are still needed".

Consider the following example:

```
public class Functional
{
    public static FunPredicate<int> Even()
    {
        return a => a % 2 == 0;
    }

}
// usage
var isEven = Even()(2);
```

In the example we define the Even function that returns a lambda expression that checks if the number given by its integer parameter is even.

## 3 Concepts – Design and Implementation

A Domain Specific Language (DSL) [16] is a special language that is dedicated to powerfully solving tasks emerging in a special problem domain. They often provide a particular problem representation or solution technique. DSLs are for example regular expressions, user interface description languages, audio description languages, math processing languages, and maybe SQL and HTML can be also regarded as DSLs. There are two methods for representing DSLs: textual and graphical form. The other important subdivision of DSLs is internal (in other words embedded) and external DSLs. Internal or embedded DSLs are created inside a host language and use the constructs of the host language while external DSLs are totally independent languages.

Our aim is to create an internal DSL for the C# 3.0 language that helps programmers to solve problems or part of problems easier where some kind of functional thinking is practical and effective.

### 3.1 Design and Implementation Expectations

To define in one sentence the expectation demands related to the design and implementation of an internal functional DSL we could say: It should use the most common features and constructs of functional languages, suggest functional problem solution methods while its expressiveness and performance is great.

The current features of our solution:

- Supports lazy and infinite lists – default behavior
- Suggests the usage of lambda expressions and closures

- Exploits the advantages of type inference
- Integrates seamlessly into the host language
- Shares the type system of the C# language
- The following common list processing operations are supported:
    - Integer range or infinite lists
    - Empty lists
    - List level Equals operator
    - Different mapping and filtering operations
    - Left and right folds
    - List queries: head, tail, IsEmpty, indexing, take, length
    - List composition: cons, compose
    - Logical evaluation operators for list members: and, or
- Output operations
- Converting lazy lists to non-lazy lists
- Great expressiveness and performance
- Extendable design

## 4 Common Tasks

Instead of the pure listing of FC# features, we show some examples and flash on the important aspects, advantages, features and optimization possibilities.

The first example is a QuickSort algorithm implementation:

```
public class QuickSort : BaseListFunc<int>
{
  public QuickSort(IFunList<int> list) :
    base(list, (x, xs) => {
      return Compose(
              Compose(
               this.QuickSort(Filter(v => v < x, xs)),
               x),
               this.QuickSort(Filter(v => v >= x, xs)));
             })
  {
  }
}
```

Every functional solution should be embedded into a class which provides encapsulation. These classes should be inherited from one of the specific predefined generic classes. It is *BaseListFunc<T>* in the above example that informally speaking accepts and returns a list. The most abstract list type is *IFunList<T>* that requires being enumerable and T can be any C# type. The constructor of QuickSort accepts a functional list of integers and performs no extra operation but calls one of the constructors of its base class. The *BaseListFunc<T>* has different constructors; we use the one that accepts the list as its first parameter and a lambda expression that has the following type:

```
IFunList<T> FunListDelegateList<T>(T head, IFunList<T> tail)
```

Thanks to the type inference support of C# the type of *x* and *xs* can be deduced by the compiler to *int* and *IFunList<int>*. The implementation uses the *Compose* operation that accepts an *IFunList<T>* and a value with type *T* or another list. The *Filter* operation accepts a filtering lambda expression and a list that is actually filtered. The explanation why Compose and Filter works without the *this* keyword and *QuickSort* why needs it will be presented in the next section.

The following code fragment shows the usage of *QuickSort*:

```
var r = new Random();
Output(new QuickSort(Map(a => r.Next(100), IntList(1, 100))));
```

We randomize a 100 elements long list where the elements take values in the 0-99 range (The *Next* method of *Random* randomizes from 0 to n-1 where n is the first parameter). The *IntList(1,100)* is simply a trick to *Map* to force it to generate a 100 elements long randomized list. *IntList* returns a lazy integer list and *Map* will return a lazy randomized list also.

Another popular problem is finding prime numbers using the sieve of Eratosthenes:

```
public class Primes : BaseListFunc<int>
{
  public Primes(IFunList<int> list) :
     base(list, (n, ns) => { return Cons(n,
                    this.Primes(Filter(v => v % n != 0, ns))); })
  {
  }
}
```

As it can be seen the implementation does not have any unknown features and operations instead of the common *Cons* operation. The input of the algorithm is an *IntList(2, n)* that returns lazy list enumerating, iterating numbers from 2 to n. There is no operation that breaks the laziness therefore the calculation will be performed totally lazy.

The implementation of the calculation of the $n^{th}$ Fibonacci number is the following:

```
public class Fibonacci : BaseCaseFunc<int>
{
  public Fibonacci(int n)
  {
    Case(() => n == 0, () => 0);
    Case(() => n == 1, () => 1);
    Case(() => n > 1, () => this.Fibonacci(n - 1) +
                                        this.Fibonacci(n - 2));
  }
}
```

The classical implementation of the above algorithm uses pattern matching that is the basic feature of most functional languages however C# does not have it in any form. The easiest way to simulate it is to create a switch-case-like construct where both parameters are lambda expressions. If a first lambda expression evaluates to true then the second lambda expression is executed and only the first match is considered in the order of definition. Solutions that return a single value and simulate pattern

matching should be inherited from *BaseCaseFunc<T>*.

The last and most complex example is the n-queens problem (the Clean language implementation is in comments):

```
public class Queens : BaseCaseListFunc<IFunList<int>>
{
  public Queens(int n) : this(n, n) { }

  // queens 0 = [[]]
  // queens n = [ [q:b] \\ b <- queens (n-1),
  //          q <- [0..7] | safeOpt q b ]
  protected Queens(int n, int maxdim)
  {
    Case(() => n == 0, () => EmptyLL<int>());
    Case(() => true, () => MapL2L(
                b => ConsL(
       Filter(q => SafeOpt(q, b), IntList(0, maxdim - 1)), b),
                                  new Queens(n - 1, maxdim)));
  }

  // safeOpt q b = and [not (checksOpt q b!!i i)
  //                       i <- [0 .. (length b)-1]]
  static Func<int, IFunList<int>, bool> SafeOpt =
            (q, b) => And((bi, i) => !ChecksOpt(q, bi, i), b);

  // checksOpt q bi i = (q == bi) || (abs(q-bi)==(i+1))
  static Func<int, int, int, bool> ChecksOpt
       = (q, bi, i) => (q == bi || Math.Abs(q - bi) == i + 1);
}
```

The n-queens problem is about putting n queens on an n*x*n chessboard in such a way that none of them is able to capture any other using the standard chess rules.

The above algorithm accepts the value of n (number of queens and chessboard dimension) and returns a list where each list element is another list of valid queen positions. Because of simulating pattern matching and returning list the Queens class is inherited from *BaseCaseListFunc<T>* where T is IFunList<int>. In case of the last recursion (where n is 0) a list is returned which only element is an empty list of integers (*EmptyLL*). *ConsL* and *MapL2L* operations are the list version of the standard *Cons* and *Map* operations with the following semantics: *ConsL* accepts two lists and for each element of the first list the second list is concatenated therefore a list of lists is generated. *MapL2L* accepts a lambda expression as the first parameter that maps a list to a list of lists using the body of the lambda while the second parameter is a list of lists which elements will be fed by the lambda expression. As it can be seen the implementation of the main algorithm is a bit more complex than the Clean language implementation thanks to the fact that in C# we are not able to express list construction as smartly as in Clean.

The other important operation in the example is the *And* operation that loops through the list elements given as the second parameter and checks if all list element evaluates to true using the lambda expression given as the first parameter. In our example the lambda expression has the following definition:

```
delegate bool FunPredicateIdx<T>(T t, int idx).
```

It is important to know both the value of a list element and the index of it in the list in the current scenario because indexing a lazy list forces explicit enumeration that downgrades performance. It is a general rule for list processing primitives that there are two versions of them: one that serves only the value of the currently processed list element and another that serves the index also (List.map vs. List.mapi in F#).

## 5 Internal Implementation Details

After giving a high level overview of the FC# library, let's dive deep into the internal implementation details: design concepts of the class hierarchy, implementation of laziness, what is behind type inference and effective handling of heads and tails for the same list.

### 5.1 Design Concepts of the Class Hierarchy

Our main guiding principle of designing the class hierarchy of the library was the ease of use and readability.

Most algorithms can be effectively implemented using lists and the most essential data structure of functional languages is the list, therefore the internal implementation heavily depends on them. Our primary list type is the generic *IFunList<T>* interface that requires implementers to be enumerable. The only direct implementer is the generic and abstract *FunList<T>* class that encapsulates the base enumeration mechanisms. Every type of class that enumerates or composes a list inherits from this class: *Cons<T>*, *Foldr<T>*, *InfiniteList<T>*, *Map<T, U>*, *Compose<T>*, *Tail<T>*, *Filter<T>*, *Take<T>*, *NonLazy<T>*, etc.

The other important aspect of the library is the ability to create new operation based on predefined operation templates that are regular classes: *BaseCaseFunc<T>*, *BaseListFunc<T>*, *BaseCaseListFunc<T>*, and the most lightweight one called *BaseValueFunc<T>*. These classes directly inherit from the very important abstract class called *Functional*.

In C# classes can be instantiated using the *new* keyword therefore all the supported operation would have to be initialized using *new* (e.g. *new Map*, *new Cons*, etc.). Without applying a trick it would degrade code readability. The *Functional* class exposes every operation that FC# supports as a static method that makes operations look like regular functions instead of classes.

The following code fragment shows one of the wrapper methods of *Map<T, U>*:

```
public static IFunList<U> Map<T, U>(FunUnary<T, U> action,
                                    IFunList<T> list)
{
    return new Map<T, U>(action, list);
}
```

Of course it is important for custom operation and algorithm implementations (e.g. Primes example in this paper) to be instantiable without using the *new* keyword.

Because the *Functional* class is inside a separate module (with other word assembly) that defines the basic operations of FC# it cannot be modified by the user. C# does not support multiple inheritance therefore inheriting tricks cannot be applied. The only solution is to define extension methods that mimic as if objects could be extended with new methods in runtime.
In the following example the *Functional* class is extended with the *Primes* method that accepts and returns *IFunList<int>*.

```
static class FunctionalExt
{
    public static IFunList<int> Primes(this Functional f,
                                       IFunList<int> list)
    {
        return new Primes(list);
    }
    //…
}
```

The only drawback of this solution is that in classes inherited from *Functional* extension methods can be only called using the *this* keyword.

### 5.2 Implementation of Laziness

FC# supports lazy lists and there are two areas where laziness should be considered:

1. The ability to create new lists
2. Operations that uses lazy list return lazy lists also

A fine example for the first case is the construction of an infinite list. When enumerating the list all elements should be evaluated lazily. The *InifiteList<T>* class presented in the next example is completely lazy and *IntList* inherits from it:

```
abstract class InfiniteList<T> : FunList<T>
{
    protected T From { get; private set; }
    protected T To { get; private set; }
    protected bool HasTo { get; private set; }

    protected InfiniteList(T from)
    {
        From = from;
        HasTo = false;
    }
    protected InfiniteList(T from, T to)
    {
        From = from;
        To = to;
        HasTo = true;
    }

    public override IEnumerator<T> GetEnumerator()
    {
        if (Terminate())
```

```
            yield break;

        T val = From;
        yield return val;

        while (!HasTo || !val.Equals(To))
        {
            val = GetNext(val);
            yield return val;
        }
    }

    public abstract T GetNext(T v);
    protected virtual bool Terminate()
    {
        return false;
    }
}
```

The *Cons* class fits the second case where laziness matters because the concatenated value (an element and a list) should be evaluated lazily:

```
sealed class Cons<T> : FunList<T>
{
    private T First { get; set; }
    private IFunList<T> Tail { get; set; }

    public Cons(T first, IFunList<T> tail)
    {
        First = first;
        Tail = tail;
    }

    public override IEnumerator<T> GetEnumerator()
    {
        yield return First;
        if (Tail != null)
        {
            foreach (T val in Tail)
            {
                yield return val;
            }
        }
    }
}
```

### 5.3 Behind Type Inference

The effects of type inference are heavily used in the library. It is mainly used to separate behavior based on the parameterization of lambda expressions. A new operation definition (e.g. *QuickSort* example in Section 4) or an internal operation

(e.g. Map) behaves differently based on the constructor using it or its parent class was initialized.

To better flash the above statements we explain it through an example. As already mentioned the parent class of *QuickSort* is *BaseListFunc<T>*. The *BaseListFunc<T>* class can be initialized using two different constructors, one that accepts and stores a *FunListDelegate<T>* type body function as second parameter and one that accepts and stores a *FunListDelegateList<T>* type body function as second parameter. When the turn of actual processing or enumeration comes *GetEnumerator* method is called that branches based on the existence of any of the stored body function.

```
public abstract class BaseListFunc<T> : Functional, IFunList<T>
{
    private FunListDelegate<T> _Body;
    private FunListDelegateList<T> _BodyList;
    private IFunList<T> _List;
    private static readonly List<T> EmptyList = new List<T>();

    protected BaseListFunc(IFunList<T> list,
                                       FunListDelegate<T> body)
    {
        _List = list;
        _Body = body;
    }
    protected BaseListFunc(IFunList<T> list,
                                    FunListDelegateList<T> body)
    {
        _List = list;
        _BodyList = body;
    }


    public IEnumerator<T> GetEnumerator()
    {
        if (_Body != null)
        {
            return _Body().GetEnumerator();
        }
        else if (_BodyList != null)
        {
            var xs = new HeadTailHelper<T>(_List);
            if (!xs.HasHead)
                return EmptyList.GetEnumerator();

            T x = xs.Head;
            return _BodyList(x, xs).GetEnumerator();
        }
        return EmptyList.GetEnumerator();
    }
}
```

As it can be seen if _Body exists then it is called normally and its enumerator is returned. However, if _BodyList exists then the head and tail values are calculated and prepared and fed to _BodyList.

### 5.4 Effective Handling of Heads and Tails

Head value of a list can be calculated by initializing an iterator and returning the first value while the tail of a list can calculated by initializing an iterator omitting the first element and enumerating through the rest of the list.

It is important to know about iterators that they do not memorize the results, chaining of IEnumerables can lead to worsening asymptotic complexity [6]. Because the head and tail values complement each other and using two iterators is waste of resources when only one can be used to achieve the same effect some optimization should be performed. We optimize the usage of iterators in this case as it can be seen in Subsection 5.3. The details of the internal implementation concepts of *HeadTailHelper<T>* class are omitted because of space limitations [20].

## 6 Performance

A very important acceptance criterion for FC# was high performance. We expect a bit lower performance than the native LINQ based C# code for some algorithms and lower performance than the Clean [14] implementation. We have also measured the performance of the same algorithms coded in F# the functional language created by Microsoft Research.

It is a rule of thumb that the higher level of abstraction an implementation has the lower performance it delivers. The LINQ based C# implementation operates of course at a lower abstraction level than FC# therefore it may provide a bit better performance than FC#. The native C# is expected to perform better than the LINQ-based version. Although we have functional constructs the C# language compiler and the .NET Framework does not provide a graph rewriting system [13] to optimize performance therefore theoretically Clean should perform better than the C# implementation. The other important difference that C# application run on top of the .NET CLR that is not optimized for algorithmic operations rather business applications (database access, user interface, business services, workflows, etc.).

In spite of the above statements the library performs quite well:

| | Clean | C# | C# (LINQ) | F# | FC# |
|---|---|---|---|---|---|
| Primes (5x Eratosthenes, 2-20000) | 203 ms | 1327 ms | 1380 ms | 4844 ms (2-12000) | 1232 ms |
| Queens (12x12 board) | 2745 ms | 1894 ms | 3919 ms | 7702 ms | 4696 ms |

We calculated the primes in the 2-20000 interval using the sieve of Eratosthenes 5 times in Clean, C# (LINQ), F#, and using FC#. F# generated a stack overflow therefore we used the 2-12000 interval instead. We also solved the n queens problem on a 12x12 chessboard.

As the numbers show the FC# library performs quite well principally compared to other .NET based solutions.

## 7 Discussion

The FC# [20] functional DSL is a library that supports functional programming in the C# language. We demonstrated the functional concepts of the C# language and demonstrated the services of the library though real examples. The library adds many functional features through an internal DSL to the C# language, including

- Similar list support that functional languages have (lazy lists, querying, filtering, composing)
- Easy integration with regular C# code and type system
- Great expressiveness and performance
- Extendable design

The library is efficient for programmers who do not want to use a fully functional language rather want to taste the efficiency of functional programming and write only some parts of their applications using functional style programming embedded into the C# host language. The most important development directions are the following:

- More operations support
- More data type support
- Caching

Functional programming languages generally belong to the GPL (General Purpose Language) class of programming languages therefore they have operations for example for IO, communication and even GUI handling. FC# is an Internal DSL that integrates with C# therefore all operations can be used in FC# that the .NET Framework exposes. It follows that we should add operations to FC# that does not belong to .NET by nature.

Currently we support the list data type (the types of FC# are subclasses of *FunList*<*T*> and the operations work with *FunList*<*T*>). Queue, double-ended queue, stack, heap and different type of trees are also heavily used in everyday algorithms. Some of these types are part of the .NET BCL (Base Class Library) however because of their internal implementation properties they cannot be efficiently used in FC#. They are not lazy data structures; do not inherit from classes exposed by FC#. In the next version of FC# these data types can be implemented.

Although in the recent version of FC# we have already implemented caching for filters, maps and logical operations, caching does not always provide higher performance for long lists. Caching is efficient for short lists. The store and retrieval performance of the cache for elements at high-list-indices should be revised.

## References


1. M. Cochran. Introduction to functional programming in C#. C# Corner, January 2008. (last seen in: http://www.c-sharpcorner.com).
2. Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters, Google Inc., OSDI'04, San Francisco, 2004
3. J. Fokker. Functional Programming, 1995 http://www.haskell.org/bookshelf/functional-programming.dvi
4. D. P. Friedman, M. Wand, and C. T. Haynes. Essentials of Programming Languages. MIT Press, 1992.
5. John Hughes. Why Functional Programming Matters, Journal of Computer Science, 32(2): 98-107, 1989.
6. Andrew Kennedy. C# is a Functional Programming Language, Microsoft Research
7. S. E. Keene. Object-Oriented Programming in Common Lisp. Addison-Wesley, 1989.
8. B. McNamara and Y. Smaragdakis. Functional programming with the FC++ library. Journal of Functional Programing, 14(4):429–472, 2004.
9. Microsoft. C# Language Specification (Version 3.0), 2007.
10. Ph. Narbel. Functional Programming at Work in Object-Oriented Programming (with the C# Case)
11. Th. Petricek. Lazy computation in C#. Technical report, Microsoft, Visual C# Developer Center, October 2007.
12. Robert Pickering. Foundations of F#, Apress 2007,
13. R. Plasmeijer – M. Eekelen. Functional Programming and Parallel Graph Rewriting, Addison-Wesley, 1993
14. R. Plasmeijer et al. Programming in Clean, http://clean.cs.ru.nl/
15. Y. Smaragdakis and B. McNamara. Bridging functional and object-oriented programming, 2000. Georgia Tech CoC Tech. Report 00-37.
16. Diomidis Spinellis. Notable design patterns for domain specific languages. Journal of Systems and Software, 56(1):91–99, February 2001.
17. Don Syme, Adam Granicz and Antonio Cisternino. Expert F#, Apress 2007,
18. Simon Thompson. Haskell – The Craft of Functional Programming, Addison-Wesley, 1999
19. TIOBE Programming Community Index, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
20. Source code of FC#: http://www.codeplex.com/fcs/