Ciencias Matemáticas, Vol. 33, No. 2, 2019, Pag. 179-188
Recibido 07-2019, Aceptado 09-2019

Decoradores tipo Python para funciones en C# utilizando atributos y la API de Roslyn Python-like function Decorators in C# using Attributes and Roslyn API

Antonio Alejo Combarro[1], José Manuel Espinoza[1], Ludwig Leonard Méndez[1], Miguel Katrib Mora[1]

Resumen Python se ha convertido en los últimos años en uno de los lenguajes de programación más populares. Cuando la capacidad de Python para interceptar llamadas a función se combina con su declaratividad, a través de los decoradores podemos tener una manera perfecta de inyectar funcionalidad adicional a código existente. Implementar decoradores en lenguajes de tipado dinámico como Python es sencillo, pero para los lenguajes con tipado estático como C# o Java, este tipo de sustitución sería imposible en tiempo de ejecución. Los trabajos anteriores para proporcionar a estos lenguajes de esta funcionalidad se basaron en el uso de un enfoque de byte-code weaving. Este trabajo propone en su lugar utilizar un enfoque de source-code weaving en C# utilizando la API de su compilador actual (Roslyn). Siguiendo este enfoque, se presenta un algoritmo para tener el efecto de decoración tipo Python para funciones en C#. Se realizaron experimentos para comparar la eficiencia de la propuesta con respecto a una herramienta comercial obteniéndose buenos resultados.

Abstract Python has become in the past years one of the most popular programming languages. When Python capability to intercept function calls is combined with the functional features then through decorators we can have a seamless way to inject additional functionality to existing code. Implementing decorators in dynamic typed languages like Python is pretty straightforward, but under the compiled model of static typed language like C# or Java, this kind of substitution would be impossible at runtime. Previous works on providing these languages of a decorator feature were based on using a bytecode weaving approach. This paper proposes instead to use a source-code weaving approach in C# by using its current compiler (Roslyn) API. Following this approach, an algorithm is presented to bring Python-like function decoration effect to C#. Experiments were conducted to compare the proposal efficiency with an state art library achieving good results.

Palabras Clave
Metaprogramación, C#, Roslyn, Python, Decoradores

Keywords
Metaprogramming, C#, Roslyn, Python, Decorators

[1] *Programming Department, Havana University, Havana, Cuba, aalejo@matcom.uh.cu, lleonart@matcom.uh.cu, mkm@matcom.uh.cu*

Introduction

Python has become in the past years one of the most popular programming languages. Usually it appears in the top 4 of most popularity rankings regarding different metrics. This is because Python is a major language in some of most exciting technologies today. Python has libraries for many science and cool technologies areas such as Machine Learning, Artificial Intelligence (AI), Big Data, and Robotics. Also Python has his own MVC for web development: Django.

Its dynamic typing character allows Python to support very important features for functional programming constructs, multiple inheritance support and metaprogramming, combined with good syntactic and tons of syntactic sugar. Decorators are one of these syntactic features. When Python capability to intercept function calls is combined with the functional features then through decorators we can have a seamless way to inject additional functionality to existing code.

It's important to remark that this should not be confused with the so called Decorator Design Pattern of the Gang of Four [5].

Python Decorator definition stands [1]:

There's been a number of complaints about the choice of the name 'decorator' for this feature. The major one is that the name is not consistent with its use in the GoF book

[1] `https://www.python.org/dev/peps/pep-0318/`

[http://patterndigest.com/patterns/Decorator.html]. The name 'decorator' probably owes more to its use in the compiler area – a syntax tree is walked and annotated.

In practice, a decorator is any callable Python object that can be applied to a function, method or class definition. A decorator receives the original object being defined and returns a modified one, which is then bound to the name in the definition. An example of a Python's decoration and its equivalent code can be seen in Code 1.

Code 1. Python's equivalent codes applying a generic decorator, with (first foo) and without (second foo) the '@' decoration symbol.

```
@decoratorname
def foo(self):
  pass

def foo(self):
  pass
foo = decoratorname(foo)
```

Implementing decorators in dynamic typed languages like Python is pretty straightforward because of its interpretative behavior, Python uses dictionaries to solve name resolution at runtime. This is also valid for languages like TypeScript, despite its static characteristics, because the code is transpiled to a dynamic language like JavaScript at the end [4]. But under the compiled model of static typed language like C# or Java, this kind of runtime substitution would be impossible at runtime.

Then, any decorator implementation in a static typed language has to find a workaround to avoid the intrinsic runtime limitations of static compilers. A common way to do this and to implement some AOP (Aspect Oriented Programming) approaches is by using code weaving tools, i.e by modifying the generated intermediate code (Java-bytecode or C#-IL) before execution. Previous works based on this approach will be analyzed in the next section.

1. Related Work

One of the most complete bytecode weaving libraries for the .NET eco-system is Fody [2]. This library is free and open-source and is built over Mono Cecil [3]. It offers a plugin architecture in which dozens of third libraries have been coded.

Fody's github page states: "Manipulating the IL of an assembly as part of a build requires a significant amount of plumbing code. This plumbing code involves knowledge of both the MSBuild and Visual Studio APIs. Fody attempts to eliminate that plumbing code through an extensible add-in model."

[2]https://github.com/Fody/Fody

[3]https://github.com/jbevain/cecil

Some of Fody's extensions are related to Decorator implementations [4]:

- MethodBoundaryAspect: Allows to decorate methods and hook into method start, method end and method exceptions (see PostSharp below).
- MethodDecorator: Decorate a method with specific codes that will executed before and after the method invocation.
- MethodTimer: Injects method timing code.

Although Fody reduces programming efforts to code IL modifying plugins, the developer should know several CLR details and the processing and production of IL code in order to program an extension.

Another popular library for .NET is PostSharp [5]. This tool isn't open-source neither free, although it has a limited free edition for testing purposes. PostSharp offers multiples products to improve C# and VB development process. Offering an AOP framework, PostSharp also provides solutions to reduce boilerplate code, to improve threading code security, among other features.

Python decorators and AOP are very related. AOP as a programming technique, tries to provide mechanisms for factoring out those fragments of an application that are not directly related to the central problem domain. In this way, decorators could be seen as a way of implementing some aspect behavior in Python.

PostSharp's AOP framework provides a functionality very similar to Decorators, by adding an aspect to classes, properties or methods via Attributes. With those aspects it can be possible to inject code which well be executed in several specific moments, such as constructor call, method enter, exit or exception throw, among others.

PostSharp AOP framework works similar to Fody using a post-compilation approach to modify Intermediate Language (IL) to include aspect behaviors. [7]

Decorators or Aspects implementations on Fody or PostSharp AOP framework rely on bytecode weaving mechanisms. This was in practice the only solution to implement AOP features in C# because the original black box characteristic of the Visual C# compiler (csc.exe). Other, but unreasonable approach, could be to reproduce a new compiler extending C# with the aspects features.

C# (bad named) attributes could be considered an attempt to achieve such purposes, but unfortunately these attributes[6] have very limited built-in behavior at compiling time. By means of attributes it's possible to insert metadata "documentation" in the generated IL code but to use this at runtime requires a lot of plumbing via reflection programming.

[4]https://github.com/Fody/Home/blob/master/pages/addins.md

[5]https://www.postsharp.net/

[6]We consider that Microsoft's decision to name this as "attribute" was unfortunately

Table 1. Source-code and bytecode weaving techniques comparison

	Source Code Weavers	Bytecode Weavers
Advantages	• Can access output source code • Code can be easier to debug • Plugins could be added to the IDE	• Cross language support for languages using the same bytecode framework
Disadvantages	• Cannot achieve cross-language support	• Harder to debug • Have not an easy way to developed the runtime interaction • Harder to code add-ons or plugins

These weakness could be avoided now with Roslyn [6] arrival and the new open-source approach for C# and VB compilers. Roslyn provides an API that could be accessed from C# code to interact with the compiler (details will be covered in next section). This facilitates the developing of source-code weaving tools.

Advantages and disadvantages of bytecode and source-code weavers are presented in Table 1.

2. Roslyn API

Until first Roslyn CTP (Community Technology Preview) release in 2011 the Microsoft's C# compiler was a black box [9]. It was an unmanaged C++ executable without any accessible API and not consistent with its VB's twin. This and other limitations led C# development team to rethink and redesign C# and VB compilers, Roslyn was the result of this process. [8]

Roslyn also brought an API to access compilers' analysis engines. Exposing parse trees and types, expression binding, and assembly production through an API Roslyn enabled a world of new scenarios including REPL, to execute C# and VB as scripting languages, and more. At the Build 2014 conference in San Francisco April 2014, Microsoft made the "Roslyn" project open-source [3]. Roslyn's first RTM (Release to Manufacturing) was with Visual Studio 2015 [10].

The .NET Compiler Platform SDK exposes the C# and Visual Basic compilers' code analysis by providing an API layer that mirrors a traditional compiler pipeline.

Each phase of this pipeline is a separate component. First, the parse phase tokenizes and parses source text into syntax that follows the language grammar. Second, the declaration phase analyzes source and imported metadata to form named symbols. Next, the bind phase matches identifiers in the code to symbols. Finally, the emit phase emits an assembly with all the information built up by the compiler.

Corresponding to each of these phases the .NET Compiler Platform SDK exposes an object model allowing the access to the information of the phase (see Figure 1). The parsing phase exposes a syntax tree, the declaration phase exposes a hierarchical symbol table, the binding phase exposes the result of the compiler's semantic analysis, and the emit phase is an API that produces IL byte codes.

The next section presents our proposal for declaring a decorator function on C# by mapping required Python features.

3. C#'s decorator declaration

To achieve in C# such decoration effect it's necessary:

1. Functions should be considered first class citizen (passed as parameters and returned from-to other functions).

2. To have the capability of declaring function inside another functions and to use variables being declared in enclosing function scope.

The first requirement is satisfied by C#'s Delegate, Action and Func types. Each of them has its advantages and disadvantages for input and output context. Delegate allows to capture any declared method regardless its signature, including its target instance. By other side, Action and Func provides a way to declare a decorator which only applies for a specific kind of functions depending its signature. So, in the current proposal a decorator functions can receive any of the above Delegate, Action or Func as target.

Nevertheless the return type of the decorated function will be always of type Func<object[], object> because this is the most general way that a function could be adapted to the user desires without forced casts when returning.

To accomplish the second requirement C# provides lambdas, delegates and the recently added local functions. These three resources can capture variables being defined in outer scopes to be used later when they will be invoked.

In Table 2, it can be seen two implementations of Memoize[7] decorator, the first on Python and the second one on C#. C# code illustrates our proposal to decorators declaration on this language.

After proposing a methodology for declaring decorator function, the next step will be defining one to decorate an arbitrary C# method. Next section will cover this topic.

[7]Adds the capability to memorize the return value of functions given the arguments in order to avoid repeated calculations when called again. This behavior is usually desired for pure functions.

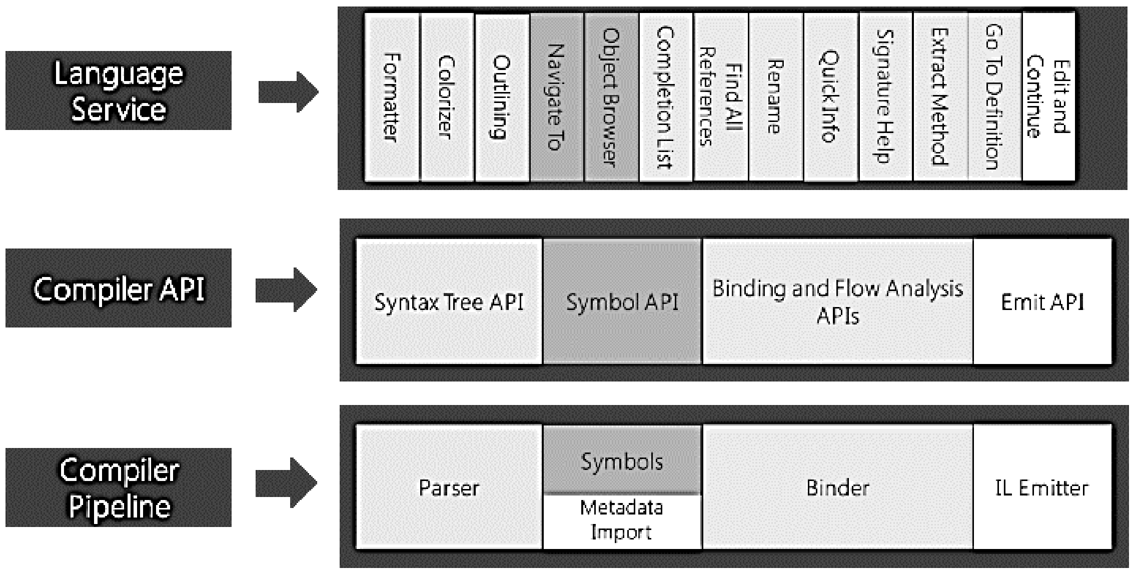

Figure 1. Roslyn's Compiler pipeline, compiler API and Language services layers mapping

4. C# Attributes as Decoration Mark

To doesn't force changes into the C# syntax we used C#'s attributes to simulate the Python's @ decoration syntax, as PostSharp and Fody also uses attributes to "annotate" code.

Unfortunately, C# attributes has several restrictions. For example, they only accept build-in types or array of build-in types as constructor arguments. This constrains will be partially addressed in the present proposal to provide a developer friendly way to define decorators.

In Python any function returning a function or a callable object can be used as decorator because the @ syntax is integrated in the language. But achieving similar expressiveness in C# without changing the language syntax is not possible. In our approach it will be used a mark attribute (@Decorate) and the C# nameof[8] operator [1] because of its Intellisense analysis. An example of a function Fibonacci being decorated with a static function Memoize placed in a DecoratorClass class is shown in Code 2.

Code 2. Decorating a Fibonacci method with Memoize decorator

```
public class SomeClass
{
    ...

    [@Decorate(nameof(Class1.Memoize))]
    public int Fib(int n){
        return n == 0 || n == 1
            ? 1
            : Fib(n-1) + Fib(n-2);
    }
}
```

This solve the syntactic side; the next section will address how to achieve the decoration behavior in a static typed language like C#.

5. Decoration effect by static method substitution

Because its interpretative execution, Python achieve decoration effect changing the method definition at runtime. Such similar runtime method definition substitution for compiled language like C# is an impossible task. Until now libraries simulate decoration doing a heavy plumbing task of injecting and editing IL code in the exe or dll files before running them.

This work proposes a Python-like approach to C#, but instead of performing an error prune runtime method redefinition to execute it make a static redefinition in a pre-compilation step. This pre-compilation substitution or source code weaving step can be done thanks to Roslyn's capabilities.

Roslyn's syntax trees are the fundamental and primary structure used for compilation, code analysis, binding, refactoring, IDE features, and code generation.

In order to provide thread-safe access syntax trees are immutable objects. Then to produce our source code weaving approach we need to produce a new tree.

Roslyn provides several ways to create and to obtain a modified version of a syntax tree: using With* and ReplaceNode methods, by instantiating DocumentEditor class [9] and by inheriting from SyntaxRewriter classes [10].

[8]The effect of the nameof used in this example means that a change of the name DecoratorClass.Memoize will be reproduced in Code.

[9]https://docs.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.editing.documenteditor?view=roslyn-dotnet

[10]https://github.com/dotnet/roslyn/wiki/Getting-Started-C%23-Syntax-Transformation

Table 2. Memoize decorator in Python (left) and C# proposal (right)

Python

```python
def memoize(function):
    memory = {}

    def decorated(*args):
        if args in memory:
            return memory[args]
        else:
            x = function(*args)
            memory[args] = x
            return x

    return decorated
```

C#

```csharp
static Func<object[], object> Memoize(Delegate d)
{
    var memory = new Dictionary<object[],
    object>(new ObjectArrayEqualityComparer());

    return (args) =>
    {
      object res;
      if (memory.TryGetValue(args, out res))
      {
          return res;
       }

        res = d.DynamicInvoke(args);
        memory[args] = res;
        return res;
    };
}
```

Each concrete class that derives from SyntaxNode defines With* methods which you can use to specify changes to its child properties. Additionally, the ReplaceNode extension method can be used to replace a descendent node in a subtree.

Instancing DocumentEditor is another way to "modify" a syntax tree that works by stacking changes (node adding, removing or replacement) to be performed in a specified C# document. In order to register changes, the class provides several methods like: InsertBefore, InsertAfter, RemoveNode, ReplaceNode. After all desired transformations are registered, the user must call GetChangedDocument to retrieve a new instance of the document with changes performed.

Due to nature of the transformations that must be done in order to perform a Python-like decoration we decided to use a SyntaxRewriter approach. SyntaxRewriter is based on Visitor Design Pattern and can be used to apply a set of transformations to multiple types of SyntaxNode wherever they appear in a syntax tree. It must be used by inheriting of it, then overriding the target node type's visitor methods and returning the new transformed node. The C#'s class, CSharpSyntaxRewriter, provides more than 200 visitor methods.

The proposed "decoration" algorithm based on static method definition substitution is divided in the following steps:

1. Find methods marked with the @Decorate attribute.

2. Given a method resulting of step 1, extract all code from it and inject the code in a new private method at the same class.

3. Given the class of the method resulting of step 1, add a static private variable which type is the same of the return value of the decorator passed as parameter to @Decorate. Then, assign to this variable the return of calling the decorator method passing the original method selected from step 1 as parameter.

4. Finally, a return expression which invokes the delegate variable obtained in step 3 by passing the inputs parameters as object array argument will be injected in the original empty method resulting from step 2.

Applying the above algorithm to Code 2 will produce the output shown in Code 3. In such code it can be seen that the method call will execute the special field __decoratedFibonacci__ of type Func which is initialized when the program begins as part of static class constructor execution. Remember that the value of this field is the decorated object returned by the decorator @Decorate attribute applied to Fibonnaci method. In this way all calls to this function will be captured as in Python and the memoization logic will be executed for each of those calls.

In next section it will be exposed some experiments performed in order to validate present algorithm and to compare its performance with other alternatives.

6. Experimental results

To experiment it was used the memoize pattern. This pattern is a very powerful cross cutting feature or aspect. It's easy to implement via decorators in Python (Table 2 left), but it's hard or impossible to implement in a general and factorized way in C# or other static typed language.

Code 3. Resulting code after applying the proposed algorithm to previous Fibonacci method using Memoize decorator

```
public class SomeClass
{
   ...

   static Func<object[], object> __decoratedFibonacci__=
            DecoratorClass.Memoize(newFunc<int,int>(Fibonacci_decorated_Memoize));

   public static int Fibonacci(int n){
      return (int)__decoratedFibonacci__(new object[]{n});
   }

   private int Fibonacci_decorated_Memoize(int n){
      return n == 0 || n == 1 ? 1 : Fibonacci(n-1) + Fibonacci(n-2);
   }
}
```

An inefficient straightforward recursive implementation of Fibonacci was used to test all memoize implementations. This function receives a int and returns a BigInteger [2] (to avoid integer operations overflow issues in our tests). Several memoize variants were implemented to be used to improve Fibonacci succession and avoid repeated calculations:

1. Plain C# with boxing/unboxing: Using a global static Dictionary<object[], object> to memoize. This forces to do boxing and unboxing and the use of the dictionary should hard-wired as a part of the Fibonacci function body.

2. Plain C# with Generic Specialization: The same as (1) but using a static external Dictionary <int, BigInteger>generic specialization.

3. Decorator with boxing/unboxing: Using a Dictionary <object[], object> but weaving it in the Fibonacci decorated function (the approach proposed in this paper).

4. Decorator with Generic Specialization: Using the variant (3) but with generic specialization, i.e substituting the parameters with the corresponding int and BigInteger.

5. PostSharp with boxing/unboxing: PostSharp with boxing/unboxing: Based on Postsharp's MethodInterceptionAspect class [11], it was implemented a memoize aspect (attribute). An implementation alternative avoiding boxing/unboxing isn't possible right now because PostSharp already performs boxing operations for function's arguments and return values.

[11]https://doc.postsharp.net/method-interception

For the experiments[12], some Fody's plugins [13] [14] were tested in order to implement a memoize variant using this library without success. These plugins don't allow to return a value directly from the method call interceptor without calling the target function first, which is the main Memoize pattern objective. Older Fody's versions had a Method Cache plugin that is currently no longer maintained[15] and doesn't work with latest library's versions.

Testing experiment 1 was performed creating a Console Application Project that calculates Fibonacci(n) for $2 <= n <= 10000$ for each variant implementation. The memory was kept while incrementing n and the Console App was executed 40 times in order to get the mean time and the standard deviation for every Fibonnacci(n) calculus.

Figure 2 shows the results of running the previously presented memoize implementations for Experiment 1 and Table 3 shows in Experiment 1 column the mean standard deviation for each implementation given the experiment. As illustrated, PostSharp has the heaviest initialization step. To verify that, it was developed a minor Experiment 2, in which the initialization part of every memoize method was measured 40 times. The results of this experiment confirm the hypothesis and its results are shown in Experiment 2 of Table 3.

Figure 3 illustrates more clearly the Experiment 1 without taking into account the initialization part. As the chart shows, this paper's proposal although it has a very low initialization time gets heavier in time than others with PostSharp's implementation following it. The most interesting part of the experiments were the performance behavior of the Plain C#

[12]All experiments were run in an Intel Core i5-6500 CPU @ 3.20 GHz with 8GB of RAM in a Visual Studio 15.8.2 Release configuration.
[13]https://github.com/Fody/MethodDecorator
[14]https://github.com/vescon/MethodBoundaryAspect.Fody
[15]https://github.com/Fody/Home/blob/master/pages/addins.md

Table 3. Standard deviation average for Experiments 1 and 3 and, mean and standard deviation for Experiment 2

	Experiment 1	Experiment 2		Experiment 3
Memoize Method	Std Deviation avg (ms)	Mean (ms)	Std Deviation (ms)	Std Deviation avg (ms)
PLAIN C# WITH BOXING/UNBOXING	1.173	0.3977	0.0854	0.6863
PLAIN C# WITH GENERIC SPECIALIZATION	0.6753	1.3956	0.1925	0.3981
DECORATOR WITH BOXING/UNBOXING	1.4449	0.5808	0.0804	1.1795
DECORATOR WITH GENERIC SPECIALIZATION	0.555	1.5338	0.1553	0.4087
POSTSHARP WITH BOXING/UNBOXING	1.2	27.0082	1.2231	1.0354

with Generic Specialization and the Decorator with Generic Specialization methods. Although, it could be expected a performance boost when removing boxing/unboxing operations by using genericity, the improve was as optimal as the optimal hard-wired solution implemented.

An additional Experiment 3 was performed in order to test in other slightly different conditions the behavior of the five Memoize implementations. It was performed as stated next: similar to Experiment 1, it was created a Console Application that calculates Fibonacci(n) for $2 <= n <= 500$ for every memoize implementations. In this case, the memory wasn't kept while incrementing n and the Console App was executed 40 times too. It was removed from the results the initial time penalty for each memoize method. The results can be seen in Figure 4.

For Experiment 3, similar results to Experiment 1 were obtained: Memoize Decorator with boxing/unboxing penalty gets heavier while n increases, followed by PostSharp's Memoize Aspect, and the Memoize Decorator without the boxing/unboxing operations gets the same performance as optimal memoize hard-wired memory decorator with specialized generic dictionary.

7. Conclusions

By doing Roslyn metaprogramming, in this work has been presented a proposal to provide a dynamic Python-like decorator functionality to a static compiling language like C#. Previous works in this line have been focused on achieve Decoration effect by using bytecode weaving tools. Instead, the current proposal present an approach based on source code weaving tools thanks to the Roslyn C# compiler API. This approach provides several advantages like that the output code will be available directly in C# and thus easier to understand and to debug.

The solution is focused in declaring First Order Functions in C# which receives either a Delegate, a Func or an Action type parameter and returns a Func<object[], object> as the decorated object. In order to not disrupt the C# syntax was used the Attribute C# feature to express the decoration attempt.

This work serves as an example of the good metaprogramming capabilities that could be achieved using the Roslyn Syntax Tree API and in particular CSharpSyntaxRewriter class methods. It was illustrated in the experimental results section the competitive performance of provided implementation compare with other approaches.

8. Future Work

The presented proposal serves as base to further improvements and extensions. One could be to implement multiple decorator composition support. As experiments showed, it could be a performance boost to implement a mechanism, similar to what C# does in runtime with generic types instantiation, to reduce or remove boxing and unboxing operations in the proposed decoration algorithm.

Another useful area for a further work is to use Roslyn to produce some metaprogramming capabilities. For example, to intersect constructor, method calls and property accesses, also could to provide a mechanism to auto-generates some members in order to reduce manual boilerplate code.

References

[1] Joseph Albahari and Ben Albahari. *C# 7.0 in a Nutshell: The Definitive Reference*, chapter 3, pages 95–96. O'Reilly Media, 2017.

[2] Joseph Albahari and Ben Albahari. *C# 7.0 in a Nutshell: The Definitive Reference*, chapter 6, pages 276–277. O'Reilly Media, 2017.

[3] Jason Bock. *.NET Development Using the Compiler API*. Apress, 2016.

[4] Steve Fenton. *Pro TypeScript Application-Scale JavaScript Development*, chapter 1, pages 76–81. Apress, Berkeley, CA, 2nd edition, 2018.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[6] GitHub. dotnet/roslyn: The roslyn .net compiler provides c# and visual basic languages with rich code analysis apis., 2019.

[7] Joe Kunk. Aspect-oriented programming with postsharp.

[8] Eric Lippert. Hiring for roslyn, 2010.

[9] Neil McAllister. Microsoft's roslyn: Reinventing the compiler as we know it, 2011.

[10] Microsoft. What's new in visual studio 2015, 2014.

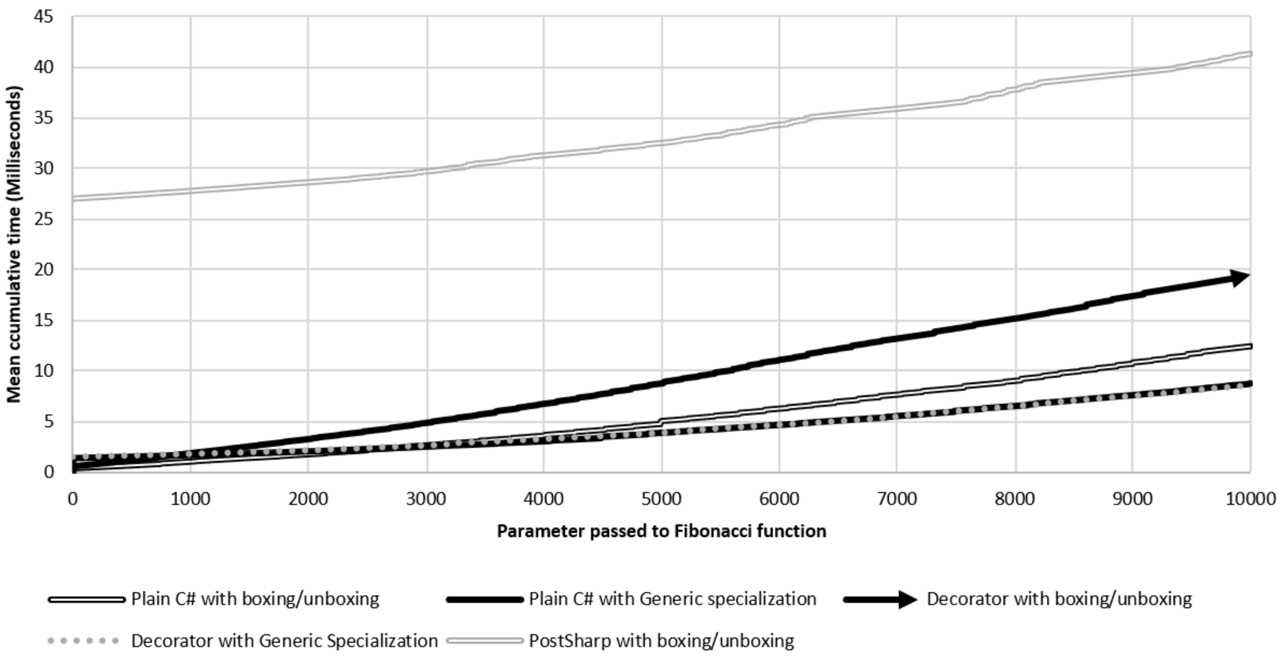

Figure 2. Time measures obtained by calling Fibonacci(n) incrementally with distinct Memoize implementations

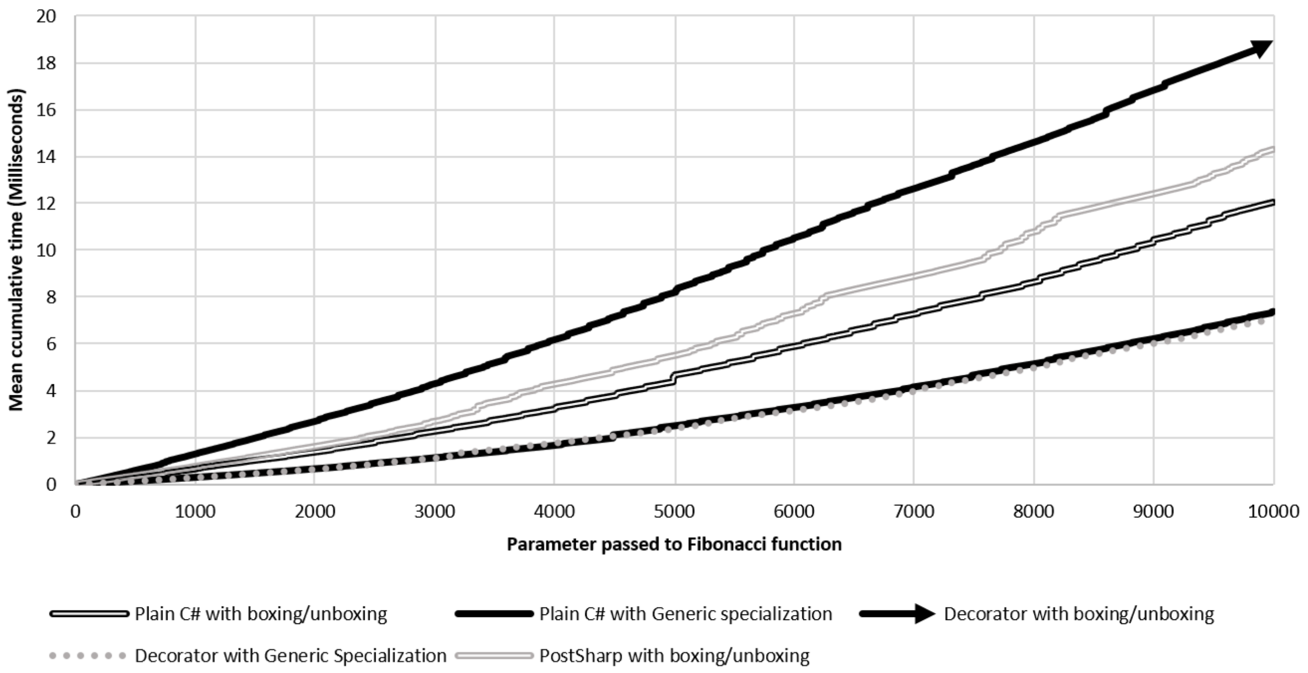

Figure 3. Several C#'s memoize implementations behavior on calling Fibonacci(n) incrementally minus initialization time

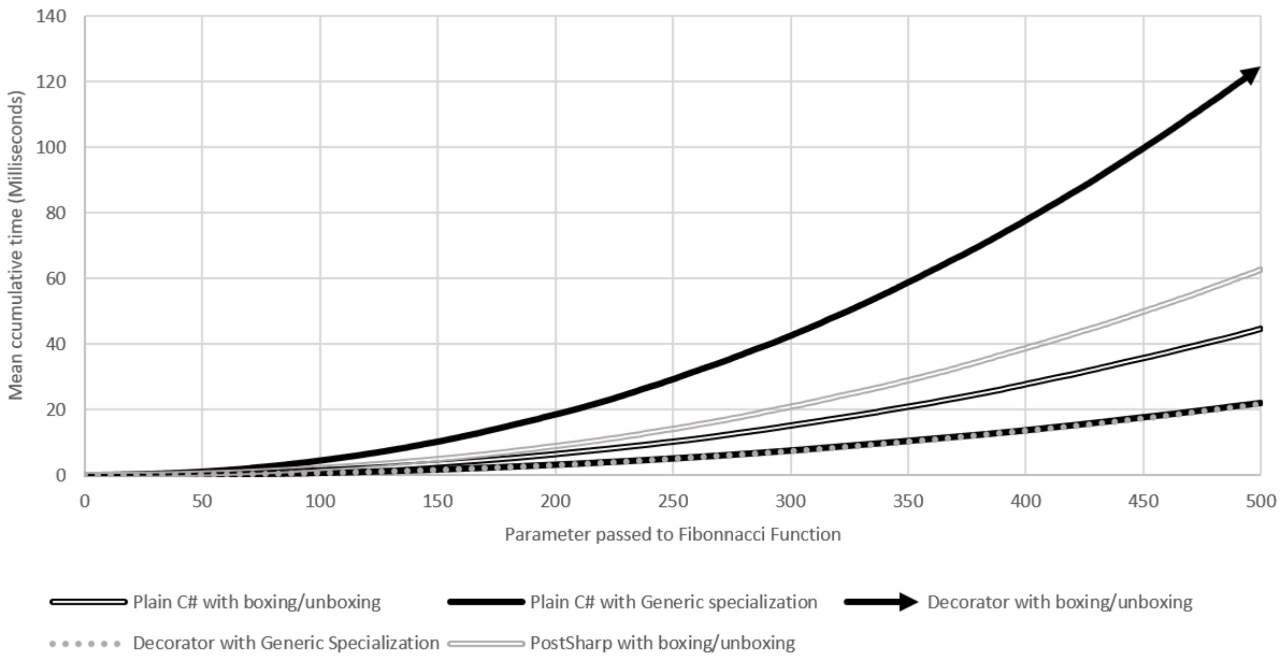

Figure 4. Several C#'s memoize implementations behavior on calling Fibonacci(n) incrementally minus initialization time and resetting the memory before every Fibonacci(n) call