UPPSALA
UNIVERSITET

Master Thesis
May 5, 2020

To what extent is C# being used as a functional programming language?

An evaluation of functional purity in C#

Melker Österberg

Civilingenjörsprogrammet i Informationsteknologi

Master Programme in Computer and Information Engineering

UPPSALA
UNIVERSITET

Institutionen för Informationsteknologi

Besöksadress:
ITC, Polacksbacken
Lägerhyddsvägen 2

Postadress:
Box 337
751 05 Uppsala

Hemsida:
http:/www.it.uu.se

Abstract

To what extent is C# being used as a functional programming language?

An evaluation of functional purity in C#

Melker Österberg

TODO

Handledare: Mikael Axelsson, Erik Löthman
Ämnesgranskare: Konstantinos Sagonas

Sammanfattning

TODO

Contents

1 Introduction

Functional programming is on the rise and becoming more mainstream [6]. Object oriented (OO) programming has been the industry norm for quite some time now, and for a long time functional programming was considered by developers to only be applicable in academic domains. However it is now becoming popular in the IT industry as well. Many mainstream languages like Java, C# and C++ have adopted first-class functions from the functional paradigm [6]. Moreover, functional programming can be used for building web applications and mobile apps [25].

Functional programs have many benefits over purely object oriented ones. Perhaps one of the most useful features of functional programming that the object oriented world of programming could adopt is *functional purity*. Programs with pure functions are generally easier to reason about compared to object oriented ones because they have no *side-effects* [2]. A side-effect is anything that a function does besides producing a return value and that is visible from the function's caller's point of view [18]. Pure functions are also easier to test since all we need to look at are functions' inputs and outputs, which also, for instance, facilitates property-based testing [2]. Moreover, research has shown that pure programs are easier to debug and maintain [18]. For this reason it useful in software engineering to evaluate the level of functional purity in programs [18].

C# is among the top five most popular programming languages [24]. To its core, it is an object-oriented programming language. However, it has features that allow for functional programming [17]. But to what extent is C# being used as a functional programming language by developers? This paper will attempt to answer this by evaluting to what degree functional purity is used in C# programs today.

2 Background

2.1 Definitions

Following are definitions of functional and objected oriented programming respectively.

Functional programming

1. All functions are *pure*, meaning that they do not have any side-effects, and their output depends only on the input [4]. The latter is what Finifter et al. call *determinism*, as mentioned in subsection 5.8 [5].

2. Functions are first-class and can be higher-order, meaning that functions can be passed to functions as parameters, and can be returned by functions [25].
3. Variables are immutable, meaning that their value does not change after being initiated [25].

Object oriented programming

1. Computations are done via *methods* belonging to *objects*, whose structure suits the goal of whatever computation we're doing [4].
2. Objects are based on *classes*, and objects belonging to a class have a shared set of properties [4].
3. Classes can *inherit* from other superclasses, such that a class is also an instance of its superclass [4].

Add a description of the following

- Side-effects and referential transparency, which implies purity [18]
- .NET Core/C#
- Compilers
- Abstract syntax trees (ASTs)
- .NET ASTs [1]
- Objects: fields and properties

Microsoft has an article on refactoring none-pure C# code to pure functions [9].

Things to consider

- Abstract syntax trees
- Input/output
- Recursion

- Monads
- Call-by-value vs. call-by-reference:
- Control flow analysis
- Sometimes we call compiled methods, i.e. methods that we don't have the code for, e.g. the .NET framework base class library (BCL)
- Delegates and anomynous functions.
- Implicit and explicit variable declarations (C# handles both)
- Identifiers are the names of types, members variables or namespaces used in source code. They reference symbols, representing a declared namespace, type, method, field, variable, etc. The compiler's process of associating identifiers with symbols is called *binding* [12].
- Closures and how objects can be considered closures (if fields are `readonly`?), and whether or not they should be considered pure.
- Exceptions.

2.2 Definition of functional purity

As mentioned in section 2, the definition of functional programming consists of three parts. *Purity*, functions being *first-class* and *higher-order*, and variables being *immutable*. To delimit the scope of this thesis it will only focus on purity, mainly because it is the one which seems the most useful for object oriented programming, and as it is the one that is the most emphasized as a benefit of using a more functional style in object oriented programming [2].

As seen in section 5 a lot of related work defines purity only as being synonymous with side-effect free. This definition therefore doesn't require functions' output to depend purely on their input – what Finifter et al. call *determinism*, as mentioned in subsection 5.8 – but allows them to read from variables defined outside of their scope. This definition is less functional. The definition of purity used in this thesis will therefore require pure functions to not only be side-effect free but also to be deterministic. This is the definition of purity that is used by Finifter et al. [5], Pitidis et al. [20] and Alexander [2], the first two who's work is mentioned in subsection 5.1 and subsection 5.8, respectively. Moreover, requiring pure functions to be both side-effect free *and* deterministic does in a way also simplify the analysis because it means that any symbol used

in a function F but defined outside F would make F impure. If we allowed pure functions to be non-deterministic, that would mean that we would have to check each symbol used in F to see if it is being written to or if it is only being read before concluding if F is pure.

Thus, a function is pure if it is side-effect free and deterministic.

2.3 What makes a C# method pure?

Figure 1 and Figure 2 illustrate two very simple examples of pure and impure code, respectively. In Figure 1 the function `addOne()` is impure because it is writing to the variable `number` which was defined outside `addOne()`'s scope, which is a side-effect. Figure 2 illustrates how `addOne()` can be rewritten to a pure function while preserving the program's semantics.

```
int number = 42;
addOne();
Console.WriteLine(number);

void addOne()
{
  number += 1; // this is a side-effect
}
```

Figure 1: A simple example of *impure* code due to a side-effect.

```
int number = 42;
number = addOne(number);
Console.WriteLine(number);

int addOne(int num)
{
  return num + 1;
}
```

Figure 2: This is how `addOne()` from the example in Figure 1 can be rewritten and used as a *pure* function.

Following are more technical requirements for C# programs to be pure that are used as a basis for the implementation of the analysis tool.

- Functions that modify any value outside their scope, e.g. a class variable or global variable, are impure. Modifying a value could be done by assigning a new value to it, or changing a part of it like modifying an element of a list or an object property.
- Functions that read any value outside the function's scope are impure. Reading a value could be done in many ways, including using it as argument in a function, as a condition in an `if`-statement, assigning it to a variable, etc. Basically any non-local variable appearing in a function would imply an impure read (except for if it is assigned to, which is mentioned above).
- Functions that return `void` are assumed to be impure (because in order for them to have an effect on the program, they must have a side-effect).
- This means that any variable read from or assigned to in a function F must have been instantiated inside F in order for F to be pure. If any of F's parameters is an object, that object cannot be modified inside F (at least not for the highest level of purity).
- Functions that call impure functions are themselves also impure.
- Functions that involve input/output (I/O) are impure.

Function parameters preceded with the `in` keyword are passed by reference and read-only inside the function [15]. This means that input parameters marked with `in` cannot be re-assigned inside the function, which may suggest functional purity. Consider the example in Figure 3, where the parameter `number` is preceded by the `in` keyword. Because of this, modifying it inside the function will raise an error.

```
int globalValue = 42;
addOne(globalValue);
Console.WriteLine(globalValue); // value is still 42

void addOne(in int number) // note the 'in' keyword
{
  number += 1; // illegal assignement will raise error CS8332
}
```

Figure 3: Assignment to the parameter `number` which is preceded with `in` raises an error [15].

However, `in` is note a purity guarantee. Consider the example in Figure 4. The `in` keyword before the argument `list` ensures that `list` is readonly. This prevents `list`

from being re-assigned after instantiation, but it doesn't prevent the data structure which `list` refers to from being modified [3]. In C# there are two kinds of types: *value types* and *reference types* [16]. Value types directly contain their data, while reference types – also known as objects – are simply pointers that refer to the location of their data. The `in` keyword therefore only implies purity if applied to a variable of a value type. For the function to be pure all its parameters have to be value types and preceded with the `in` keyword. Even though strings are of the reference type they are immutable, meaning that they cannot be modified after being created [14]. Therefore the same thing that in this case applies to value types also applies to strings.

```
List<int> globalValue = new List<int>{42};
addOne(globalValue);
globalValue.ForEach(Console.WriteLine); // list is now {42, 1}

void addOne(in List<int> list) // note the 'in' keyword
{
  list.Add(1); // this will _not_ raise an error even though
               // Add(1) modifies the list
}
```

Figure 4: The expression `list.Add(1)` which writes a value to `list` is allowed, even though `list` is read-only due to its preceding `in` keyword.

TODO

2.4 The .NET Abstract Syntax Tree and the CodeAnalysis library

In this thesis the term *method* and *function* will be used interchangeably.

Abstract syntax trees (ASTs) are the primary data structure used when analysing source code [10]. It encapsulates every piece of information held in the source code [10]. A syntax tree generated by a parser can be be re-built into the exact same text that was originally parsed [10]. TODO

The abstract syntax tree (AST) generated by the texttttMicrosoft.CodeAnalysis library represents the lexical and syntactic structure of a .NET program [10]. The tree consists primarily of *syntax nodes* which represent syntactic constructs including declarations, statements, clauses and expressions [10]. Each node is derived from the `Syntax-Node` class [10]. Every node is non-terminal, meaning that they always have children - either other nodes or *tokens* [10]. Tokens are the smallest syntactic pieces of the program, consisting of keywords identifiers, literals and punctuation [10].

2.4.1 Microsoft Syntax Analysis API CodeAnalysis

`Microsoft.CodeAnalysis` is a Syntax API developed by Microsoft which allows programmers to convert a C# program to an abstract syntax tree and traverse it [13]. TODO

AST nodes

- The `ClassDeclarationSyntax` node represents the declaration of a class.
- The `MethodDeclarationSyntax` node represents the declaration of a method. Each `MethodDeclarationSyntax` node has a `ParameterListSyntax` - a list of `ParameterSyntax` nodes represents, each representing a method parameter.
- The `LocalFunctionStatementSyntax` node represents a function declared inside a method. Just like the `MethodDeclarationSyntax` it has a `Param-eterListSyntax` representing the local function's parameters.
- The `ReturnStatementSyntax` node represents the `return` token.
- The `VariableDeclarationSyntax` node represents declarations of new variables.
- The `LocalDeclarationStatementSyntax` node represents declarations of new local variables. This node contains a `VariableDeclarationSyntax` node.
- The `AssignmentExpressionSyntax` node represents assignments to already instantiated variables.
- The `IdentifierNameSyntax` node represents symbols used in the code, including variable names and method names.

2.5 C# Events

Events are a way for classes or object to notify other classes or objects when something happens [8]. The class raising the event is called the *publisher* and the class handling the event is called the *subscriber* (there can be more than one subscriber) [8]. When an event is raised the subscriber's handler method is executed. Since events clearly are side-effects, a method that raises events is not considered pure

2.6 Problems with determining purity in object oriented languages

2.6.1 Inheritance

When calling an argument's method, because of inheritance and method overriding we can never be sure of which method implementation will be called. Consider the following example [19]:

```
void f(List<string> x) {
  x.Add("Hello");
}
```

Figure 5: Since `x` can be of any subclass of `List` we can never be sure of `x.Add()`'s implementation.

Because the parameter `x` can be of any subclass of `List` we can not for sure know the implementation of `x.Add()`, nor therefore can we be certain of `x.Add()`'s purity. Thus, we can not determine `f()`'s purity.

One solution to this that David J. Pearce suggests is to demand that pure methods only are overridden by methods that are also pure [19]. Therefore, if a method *m* is overridden by at least one impure method, *m* is assumed to be impure.

This means that in the example in Figure 5, the function `f()` is pure iff all methods that override `List.Add()` are pure.

2.6.2 Modifying a fresh object

If an object *o* is allocated inside the analysed method *m*, the object *o* is said to be *fresh* [19]. To modify *o*'s state we might call a method that looks impure (since that method would have the side-effect of modifying *o*). However, this method should not make *m* impure since *o* is fresh, which means that the modification of *o* is not a side-effect of *m*.

Consider the following example:

```
public List<String> Foo() {
  List<String> list = new List<String>();
  list.Add("hello"); // this changes list's state
  return list;
}
```

Figure 6: TODO

Because `list.Add()` in Figure 6 modifies the state of `list`, which is a side-effect, `Add()` cannot be a pure method. Does that mean that since the function `Foo()` calling `list.Add()` is impure, because it calls a non-pure method? In general functions that invoce impure functions are themselves impure. However, this is not the case for `Foo()`. Recall the definition of purity in subsection 2.2:

A function is pure if it is side-effect free and deterministic.

Because the function `Foo()` only modifies an object exclusively visible inside the function, `Foo()` does not have any side-effect. Nor is the function non-deterministic since it does not read any value outside of the function besides its parameters, which it in this case doesn't have. This means that `Foo()` is pure.

To solve this we introduce the purity level *locally impure*. Any method that is pure except for reading or modifying its object's fields is locally impure.

```
public class List {
  public string[] values;
  public int size;

  public void Add(string s) {
    ... // Increase size of values array

    values[++size] = s; // locally impure action
  }
}
```

Figure 7: TODO

2.6.3 Iterators

This part is probably not relevant because my definition of purity includes determinism. Do I mention this or just delete this part?

As Pearce suggests, an iterator may at first glance look pure [19]. Consider the following (link):

```
bool ListHas(List<int> items, int item) {
   foreach(int i in items) if(i == item) return true;
   return false;
}
```

Figure 8: `ListHas()` may at first glance look pure

2.7 The Analysis

- There are three kinds of purity levels: *pure*, *impure*, and *unknown*.
- Traverse the Abstract Syntax Tree (AST) and build the *dependency set* of each function, i.e. the set of calls inside the function. Each function together with its dependency set is stored in a lookup table where the key is the function identifier f and the values are f's dependency set D_f as well as f's purity level p.
- If any field or property of the currently analyzed method's object is read from or modified, mark it as impure. TODO
- As discussed in subsubsection 2.6.1, the currently analyzed method m is overridden by any impure method, mark m as impure.
- If a method returns `this` it should be considered impure since it is dependent on the state of its object, making it non-deterministic.
- Any method that raises an event or an exception is marked as *impure*.
- If the analysis finds a call or reference to a compiled method or a field, mark its purity as *unknown*.

If we modify a list that was created inside a method m, say by calling the function `list.append()`, m may look like it has a side-effect, even though it hasn't. Discussed in subsubsection 2.6.2.

Solution: if a called method m is (locally) impure, only mark m's callee M as impure if the object which m belongs to was not created inside the callee M. *Does this really work for all cases?*

2.8 Code base

Mostly pure code

- `ImmutableHashMap.cs` - has many methods marked with `[pure]`
- NodaTime has many pure methods. NodaTime's `AnnualDate.cs` for instance has many pure methods not marked with `[pure]`

Mostly impure code

- ImageSharp
- sharpDox

3 Implementation of the analysis tool

Because there was a clear goal for the software and it's requirements, test driven development was used.

Because .NET has many libraries with functions of unknown purity, the purity level "unknown" has to be added.

The code analysis tool is built in .NET core and C#. It can be compiled, run and tested from the command line using the command `dotnet`, followed by `run`, `build` or `test`, respectively.

To find the declaration/definition of symbols (in the form of `IdentifierNameSyntax`) the method `SemanticModel.GetSymbolInfo()` is used, which uses the semantic model of the program.

TODO

4 Results

Static method purity is calculated as the percentage of all methodsin the call graph that are pure [26]. TODO

5 Related work

5.1 Purity in Erlang

In their paper *Purity in Erlang* Mihalis Pitidis and Konstantinos Sagonas develop a tool that automatically and statically analyses the purity of Erlang functions [20]. It classifies functions into being functionally pure, or one of three levels of functional impurity [20]. The three levels of functional impurity they defined are: containing side-effects; containing no side-effects but being dependent on the environment; and containing no side-effects, having no dependencies on the environment but raising exceptions [20].

Their definition of purity uses *referential transparency*, as it implies purity [20]. Referential transparency means that an expression always produces the same value when transparency [20]. This means that a referentially transparent function could always be replaced with it's output without altering the program's behaviour in any way [20].

They store all analysis information in a *lookup table* where the keys are the function identifiers f and the values are the purity level p_f of each f as well as f's *dependency set* D_f [20]. The dependency set is the set of functions being called by f and is constructed by parsing the program's Abstract Syntax Tree [20].

Their analysis starts with Erlang's so called built in-functions (BIFs), which are functions native to the Erlang's virtual machine and are written in C [20]. Impure actions in Erlang can only be done through BIFs, including performing I/O actions or writing to global variables [22]. Because BIFs are written in C they cannot be analysed by their analysis tool, and their purity is assumed to be already known in beforehand by the analysis tool [20]. The analysis propagates the impurity of BIFs to each function which directly or indirectly depends on them.

In short terms, their analysis algorithm works like this: Initialize the purity of all functions in the lookup table to be analyzed to "pure" [20]. Define the *working set* to always equal the set of functions whose purity level is fully determined, i.e. the functions with empty dependency sets [20]. For each function f in the working set, propagate its purity level to functions depending on it and "contaminate" them with f's purity level [20]. Then remove f from the dependency set of each function depending on it [20]. If f has the highest impurity level, remove the entire dependency set of each function depending on f [20]. If the working set gets empty, find a set of functions that are dependent on each other and no other functions, and set their purity level to the purity of the impurest function [20]. Simplify their dependency sets by removing their dependency on each other from their dependency set [20]. Repeat this process until there are no more changes to the lookup table [20].

TODO

5.2 JPure: A Modular Purity System for Java

David J. Pearce built a purity system and analyzer for Java in his paper JPure: a modular purity system for Java [19]. The system uses the properties *freshness* and *locality* to increase the the system's ability to classify methods as pure [19]. An object is fresh if it is newly allocated inside a method [19]. An object's locality is its local state [19]. Their definition of a pure method is one that does not assign (directly or indirectly) to any field that exisetd before the method was called [19].

The system uses uses annotations `@Pure`, `@Local`, `@Fresh`. `@Pure` indicates that a method is pure. `@Local` indicates that a method only modifies an object's locality. `@Fresh` indicates that a method only returns fresh objects. These three annotations are modularly checkable, i.e. one method's purity annotations to be checked in isolation from all other methods.

The system consists of two parts, *purity inference* and *purity checker* [19]. Purity inference adds `@Pure` annotations (and any auxiliary annotations required) to the code and is intended to be run once because it is more costly. The purity checker checks the correctness of all annotations at compile-time, and is intended to be used continuously to maintain the code's purity.

TODO

5.3 Purity and Side Effect Analysis for Java Programs

Similarly, Sălcianu and Rinard presented a method for analysing purity in Java programs, but their definition of purity also only includes side-effects and does not look at the input or output [23]. Their pointer analysis is based on tracking object creation and updates, as well as updates to local variables, and defines methods that mutate memory locations that existed before a method call as impure [23]. Moreover, their analysis can recognize purity-related properties for impure methods, including *read-only* and *safe* parameters [23].

The analysis method presented looks at each program point in each method m, and computes a points-to graph modelling the parts of the heap that method m points to, represented by nodes in the graph [23]. There are three kinds of nodes: *Inside nodes* which model objects created by m, *parameter nodes* which model objects passed to m

as arguments, and *load nodes* modelling objects read from outside *m* [23]. Edges in the points-to graph model heap references [23]. There are two types of edges: *inside edges* which model heap references created by *m*, and *outside edges* modelling heap references read by *m* from outside of it (this includes *m*'s parameters) [23].

The analysis also keeps track of *globally escaped nodes*, which are nodes that may be accessed by unknown code, i.e. passed as argument to a native methods or pointed to static fields [23]. Since globally escaped nodes may be mutated by unknown code, the analysis has to handle them conservatively [23].

To check if a method *m* is pure, the analysis computes the set *A* consisting of nodes reachable from parameter nodes along outside edges [23]. In other words, *A* represents all objects existing before executing *m* [23]. *m* is pure if and only if no node in *A* escapes globally (i.e. is accessed by unknown code) and no fields in any node in *A* is modified [23]. There is one exception to the purity constraint: constructors are allowed to mutate fields of the `this` object [23]. Therefore all mutated abstract fields of `this` are ignored by the analysis [23].

TODO

5.4 Detecting function purity in JavaScript

Nicolay et al. developed a method of detecting function purity in JavaScript using something called *pushdown analysis* [18]. Their definition of functional purity, however, includes only side-effects and does not require functions' output to depend purely on their input [18].

5.5 Writing Pure Code in C#

In his article *Writing Pure Code in C#* Massad defines three levels of pure methods [7]:

1. Pure methods, i.e. methods that do not read or write to instance state variables, or call impure methods.
2. Methods that are pure, and that *read* the state of their containing object, or the state of objects that are passed as parameters or created in the current method.
3. Methods that are pure, and that *read* or *write* to the state of their containing objects, or to the state of objects created in the current method.

5.6 .NET Code Contracts

.NET code contracts are used to define pre- and postconditions, as well invariants for pieces of code – some which can be checked statically and some at runtime [11]. One available code contract is the `[pure]` attribute, which indicates that the method is pure [11]. However, current analysis tools do not enforce that methods marked with `[pure]` actually are pure, and so the attribute does not guarantee functional purity. Microsoft defines pure methods as methods that don't modify an pre-existing state, i.e. methods can only modify objects that were created *after* the method was called. The following code elements are assumed by the code contract tools to be pure [11]:

- Methods or types marked with `[pure]` (for types marked with `[pure]` this should apply to to all the type's methods).
- Property get accessors.
- Operators.
- Any method with a fully qualified name starting with `System.Diagnostics.Contracts.Contract`, `System.String`, `System.IO.Path`, or `System.Type`.
- Any called delegate with the `[pure]` attribute. Delegates are basically function pointers.

5.7 Evolution of Degree of Purity in Programming Languages

Rajasekhara Babu et al. [21].

5.8 Verifiable Functional Purity in Java

In their definition of functional purity Finifter et al. require pure functions to be both side-effect free and *deterministic* [5]. A function is deterministic if any two evaluations of it have the same result [5]. This means that a deterministic function is one that relies purely on its arguments [5]. A function is side-effect free if it only modifies objects that were created during its execution [5].

The language that their analyzer handles is a subset of Java, in which they can prove functional purity [5]. They make the point that if all of method's parameters are im-

mutable, including the implicit `this`, then the method is pure [5]. If its class is immutable it means that a method's global scope has a constant state, and so the only varying state is the one observable through its arguments [5].

Their verifier has a white list of fields and methods from Java libraries that do not expose the ability to observe a global mutable state, or provide access to nondeterminism, and it will reject any reference to a field or method that is not on the list [5].

6 Future Work and Conclusion

6.1 Future work

- Tool can point out exact position of impurity and suggest improvements to increase the purity.

6.2 Conclusion

Functional purity is perhaps the most useful concept from functional programming that object oriented programmers can learn from. The definition of a pure program used in this paper is one that is side-effect free and determinstic.

References

[1] J. Albahari and B. Albahari, "C# in a nutshell," 2003.

[2] A. Alexander, *Functional programming simplified*, 2017.

[3] D. Buchanan, "Common problems with static lists," 2015, accessed 2020-04-02. [Online]. Available: http://dillonbuchanan.com/programming/common-problems-with-static-lists/

[4] J. M. Chambers, "Object-oriented programming, functional programming and R," Tech. Rep., 2014. [Online]. Available: https://projecteuclid.org/euclid.ss/1408368569

[5] M. Finifter, A. Mettler, N. Sastry, and D. Wagner, "Verifiable functional purity in java," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 161–174.

[6] T. Helvick, "Why functional programming is on the rise again," 2018.

[7] Y. Massad, "Writing pure code in C#," 2018, accessed 2020-02-24. [Online]. Available: https://www.dotnetcurry.com/csharp/1464/pure-code-csharp

[8] Microsoft, "Events (C# programming guide)," 2015, accessed 2020-04-21. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/

[9] ——, "Refactoring into pure functions (C#)," 2015, accessed 2020-02-27. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/refactoring-into-pure-functions

[10] ——, "Work with syntax," 2017, accessed 2020-03-10. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/work-with-syntax

[11] ——, "Code contracts," 2018, accessed 2020-02-27. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/debug-trace-profile/code-contracts#purity

[12] ——, "Get started with semantic analysis," 2018, accessed 2020-03-12. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/get-started/semantic-analysis

[13] ——, "Get started with syntax analysis," 2018, accessed 2020-02-27. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/get-started/syntax-analysis

[14] ——, "Strings (C# programming guide)," 2019, accessed 2020-04-03. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/strings/

[15] ——, "in parameter modifier (C# reference)," 2020, accessed 2020-04-01. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/in-parameter-modifier

[16] ——, "Types and variables," 2020, accessed 2020-04-03. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/types-and-variables

[17] H. Mosalla, "Functional programming in C#: A brief guide," 2019, accessed 2020-04-01. [Online]. Available: http://hamidmosalla.com/2019/04/25/functional-programming-in-c-sharp-a-brief-guide/

[18] J. Nicolay, C. Noguera, C. De Roover, and W. De Meuter, "Detecting function purity in javascript," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sep. 2015, pp. 101–110.

[19] D. J. Pearce, "Jpure: a modular purity system for java," in *International Conference on Compiler Construction*. Springer, 2011, pp. 104–123.

[20] M. Pitidis and K. Sagonas, "Purith in erlang," in *Symposium on Implementation and Application of Functional Languages*. Springer, 2010, pp. 137–152.

[21] M. Rajasekhara Babu, A. S. Baby, D. Raveendran, and A. Joe, "Evolution of degree of purity in programming languages," *Procedia Technology*, vol. 6, pp. 354–361, 2012.

[22] K. Sagonas, personal communication, 2020-04-09.

[23] A. Sălcianu and M. Rinard, "Purity and side effect analysis for java programs," in *Verification, Model Checking, and Abstract Interpretation*, R. Cousot, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 199–215.

[24] TIOBE, "TIOBE index for january 2020," 2020, accessed 2020-04-01. [Online]. Available: https://www.tiobe.com/tiobe-index/

[25] J. Wälter, "Functional programming for web and mobile – a review of the current state of the art," Tech. Rep., 2019.

[26] H. Xu, C. J. Pickett, and C. Verbrugge, "Dynamic purity analysis for java programs," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2007, pp. 75–82.