UPPSALA
UNIVERSITET

Master Thesis
October 9, 2020

To what extent is functional purity being used in C# open source programs?

A methodology for measuring 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 functional purity being used in C# open source programs?

A methodology for measuring functional purity in C#

Melker Österberg

This thesis develops a method of statically determining the level of purity in a given C# program. It investigates problems with determining purity in object oriented languages, with a focus on C#. A function is defined to be pure if it side-effect free and deterministic.
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 [26].

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 impure 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 [19]. 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 [19]. For this reason it useful in software engineering to evaluate the level of functional purity in programs [19].

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 [18]. But to what extent is C# being used as a functional programming language by developers? This paper will attempt to answer this by evaluating to what degree functional purity is used in C# programs today.

2 Definitions

Functional programming and object oriented programming are two different programming paradigms. There is no universal definition of either of them. Following is how they have been chosen to be defined in this thesis.

2.1 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. Each object has a unique *object identity* which distinguishes it from all other objects [7].
3. Objects are based on *classes*, and objects belonging to a class have a shared set of properties [4].
4. Classes can *inherit* from other superclasses, such that an object of a class is also an object of its superclass [4].

2.2 Functional programming

1. All functions are *functionally pure*. Functional purity is defined below in subsection 2.3.
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 [26].
3. Variables are immutable, meaning that their value does not change after being initiated [26].

To delimit the scope of this thesis it will only focus on functional purity, mainly because it seems like the most useful one out of the three for object oriented programming.

2.3 Definition of functional purity

Following is the definition of functional purity that will be used in this thesis:

Definition 1 (Functional purity)**.** *A function is functionally pure if it is side-effect free and deterministic.*

Side-effect and *determinism* are defined as follows:

Definition 2 (Side-effect)**.** *A side-effect is any action performed by a function that is visible from the function's caller's point of view [19].*

Definition 3 (Determinism)**.** *A function is deterministic if its output depends purely on its input parameters, i.e. the method must return the same value for the same input regardless of the state of the program [5].*

As seen in section 8 some related work define functional purity only as being synonymous with ”side-effect free”. This definition omits determinism and allows functions to read from variables defined outside of their scope, which 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. [21] and Alexander [2]. 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.

3 The C# language and .NET

C# is a type safe object oriented programming language developed by Microsoft [1]. C# is syntactically quite similar to C, C++ and Java, and includes features like nullable types, enumerations, higher-order functions, and direct memory access [11]. While C# historically has primarily been used for writing code for Windows platforms only, C# has recently spread to most other platforms, including mobile, due to its increased cross-platform support [1].

C# applications run in the .NET ecosystem [11]. There are multiple implementations of .NET, including .NET Core and the .NET Framework [11]. .NET includes a virtual execution environment called the common language runtime (CLR) on which C# programs run, as well as a common set of class libraries [1]. Before execution C# source code is compiled to the so called intermediate language (IL) and stored on disk [11]. Upon execution the IL code is just-in-time-compiled to native machine instructions that can be executed by the operating system [11].

Similarly to many other programming languages, functions in C# are generally referred to as *methods* [1]. These two terms will be used interchangeably in this thesis. Each method belongs to a *class*, which is an encapsulations of data and behaviours [1]. C# also supports anonymous functions [1].

3.1 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.

```
public class Program {
  static int number = 42;

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

  public static void Main()  {
    addOne();
    Console.WriteLine(number); // outputs 43
  }
}
```

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

```
public class Program {
  public static int addOne(int number) {
    return number + 1; // this method is now pure
  }

  public static void Main()  {
    int number = 42;
    number = addOne(number);
    Console.WriteLine(number); // outputs 43
  }
}
```

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

Function parameters preceded with the `in` keyword are passed by reference and read-only inside the function [16]. 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 assignment will raise error CS8332
}
```

Figure 3: Assignment to the parameter `number` which is preceded with `in` raises an error [16].

However, `in` is not a purity guarantee. Consider the example in Figure 4. The `in` keyword before the argument `list` ensures that `list` is read-only. 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* [17]. Value types directly contain their data, while reference types – also known as objects – are simply pointers that refer to the location of their data. Even though strings are of the reference type they are immutable, meaning that they cannot be modified after being created [15]. Therefore, value types and strings are passed to methods by value, which means that a method that modifies a value type parameter only modifies it locally, which doesn't affect its purity.

```
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.

Since the `in` keyword does not entirely prevent methods modifying their input parameters of reference type, and mutations to parameters of value type are not visible outside of methods regardless of the `in` keyword, it does not help us to determine the purity of a method.

3.2 The .NET Abstract Syntax Tree and the CodeAnalysis library

Abstract syntax trees (ASTs) are the primary data structure used when analysing source code [13]. It encapsulates every piece of information held in the source code [13]. A syntax tree generated by a parser can be be re-built into the exact same text that was originally parsed [13].

The abstract syntax tree (AST) generated from a given piece of C# code by Microsoft's code analysis library `Microsoft.CodeAnalysis` represents the lexical and syntactic structure of the C# program [13]. The tree consists primarily of *syntax nodes* which represent syntactic constructs including declarations, statements, clauses and expressions [13]. Each node is derived from the `Syntax-Node` class [13]. Every node is non-terminal, meaning that they always have children - either other nodes or *tokens* [13]. Tokens are the smallest syntactic pieces of the program, consisting of keywords, identifiers, literals and punctuation [13].

3.3 C# Events

Events are a way for classes or object to notify other classes or objects when something happens [10]. 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) [10]. When an event is raised the subscriber's handler method is executed. Since events clearly are side-effects, a method that raises events or handles events is not considered pure

3.4 Impure built-in C# methods

The following methods that are built into C# and are non-deterministic [28]:

- `Console.Read`, `Console.ReadLine`, `Console.ReadKey`, `DateTime.Now` and `DateTimeOffset.Now` depend on the outside world.
- `Random.Next`, `Guid.NewGuid` and `System.IO.Path.GetRandomFileName` give random output.

The following methods that are built into C# and have side-effects [28]:

- `System.Threading.Thread.Start` and `Thread.Abort` mutate states.

- `Console.Read`, `Console.ReadLine`, `Console-ReadKey`, `Console.Write` and `Console.WriteLine` produce console I/O.
- `System.IO.Directory.Create`, `Directory.Move`, `Directory.Delete`, `File.Create`, `File.Move`, `File.Delete`, `File.ReadAllBytes` and `File.WriteAllBytes` produce file system I/O.
- `System.Net.Http.HttpClient.GetAsync`, `HttpClient.PostAsync`, `HttpClinet.PutAsync` and `HttpClient.DeleteAsync` produce network I/O.
- `IDisposable.Dispose` interacts with the program's environment.

The fact that previously mentioned methods are non-deterministic or have side-effects means that we know for sure that they are impure, which means that any function that uses them is also impure.

3.5 .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 [14]. One available code contract is the `[pure]` attribute, which indicates that the method is pure [14]. However, current analysis tools do not enforce that methods marked with `[pure]` actually are functionally pure [9], and so the attribute does not guarantee functional purity. Microsoft defines pure methods as methods that don't modify any pre-existing state, i.e. methods can only modify objects that were created *after* the method was called [14]. The following code elements are assumed by the code contract tools to be pure [14]:

- 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.

4 Problems with determining purity in object oriented languages

Analyzing functional purity in object oriented programming languages yields a number of dilemmas. This section describes them, as well as how they were dealt with.

4.1 Inheritance and method overriding

When calling an object parameter's method, because of inheritance and method overriding we can never be sure of which method implementation will be called. Consider the following example [20]:

```
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 [20]. 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.

4.2 Modifying a fresh object

If an object *o* is allocated inside the analysed method *m*, the object *o* is said to be *fresh* [20]. 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: `List.Add()` has a side-effect because it modifies the list `list`. However, since `list` is fresh `Foo()`'s purity is not affected.

Because `list.Add()` in Figure 6 modifies the state of `list`, which is a side-effect of `Add()`, `Add()` cannot be a pure method. Does that mean that the function `Foo()` calling `list.Add()` is also impure, because it calls a non-pure method? In general functions that invoke impure functions are themselves impure. However, this is not the case for `Foo()`. Recall the definition of purity in definition 1:

A function is functionally 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. The function is also deterministic since it does not read any value outside of the function besides its own 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*:

Definition 4 (Local Impurity)**.** *Any method that is functionally pure except for modifying any of its own object's fields is locally impure.*

Consider the following example:

```
public A Foo() { // pure
  A a = new A();
  a.Increment();
  return a;
}

public class A {
  public int value = 0;

  public void Increment() { // locally impure
    value++;
  }
}
```

Figure 7: Since `Foo()` modifies a fresh object with a locally impure method `Foo()` is still pure.

The method `Increment()` in Figure 7 is locally impure because it modifies its object's field `value` but doesn't have any other side-effects. A function that calls a fresh object's locally impure method *m* is not contaminated by *m*'s local impurity.

There is however one more way to modify a fresh object: to pass it as an argument to a method that alters its state. Consider the following example:

```
public A Foo() { // pure
  A a = new A();
  A.Increment(a);
  return a;
}

public class A {
  public int value = 0;

  public static void Increment(A a) { // parametrically impure
    a.value++;
  }
}
```

Figure 8: Since `Foo()` modifies a fresh object with a parametrically impure method `Foo()` is still pure.

Because the method `Increment()` in Figure 8 modifies its input parameter object it cannot be considered truly pure. However, since `Foo()` uses `Increment()` to modify a fresh object `Foo()` is still pure. I have chosen to categorize methods like this *parametrically impure*:

Definition 5 (Parametrical impurity)**.** *Any method that is pure except for modifying the state of its input is parametrically impure.*

There are two ways for a method to modify its input parameters: either by mutating a value belonging to a reference type input parameter, i.e. an object's field or property, or the cell of an array; or by calling a locally impure method belonging to a parameter object.

The key thing in both the example where a method *m* calls locally or parametrically impure methods is that they modify a fresh object, which therefore doesn't affect *m*'s purity level. So as long as the analyzed method *m* or any of its called methods don't perform any truly impure action like I/O operations (e.g. writing to a file on the file system) or throw exceptions, *m* is pure.

In Python, which also is an object-oriented programming language, the instance object equivalent to C#'s `this` (in Python usually referred to as `self`) always has to be explicitly listed as the first parameter of a method when declaring it, and gets automatically passed as as the method's first argument when the method is called [25]. Similarly, in C# the reference to the instance object `this` is in fact an implicit parameter for all methods [8]. This means that a method *m* that reads the value of a field of its own object still is considered deterministic even though that field has not been explicitly added as a method parameter. Recall from definition 3 that *m* is deterministic if *m*'s output depends purely on its input parameters. This still holds with *m*, since its entire object `this` where that field lives gets implicitly passed as an argument to *m*.

Moreover, the implicit passing of `this` to every method also implies that any method that is locally impure is also parametrically impure. As definition 4 states, a method *m* that is locally impure modifies its object `this`'s fields. Since `this` is always the first parameter of any method, this means that modifying a field in `this` always means modifying a parameter, which implies parametrical impurity.

Although local impurity implies parametrical impurity, the opposite does always necessarily hold since a method could be parametrically impure without modifying a field in `this`. Therefore, distinguishing between local and parametrical impurity could be of value, for instance when implementing other methods in the same class; since locally impure methods modify the state of their object, and those object-local side-effects are visible for methods inside the same object.

4.3 Non-static property pointing to a static field

Properties in C# are special `get` and `set` methods for reading, computing or writing to values of object fields [12]. Properties can, just like other fields or methods be set to `static`, which means that they are assigned to the class rather than any instantiated object. If a method reads from or modifies a static property, that method would be considered to be non-deterministic or to have a side-effect, i.e. it would be impure, because static fields are accessible anywhere in a program.

However, properties can also be non-static but still read or modify static fields. Consider the following:

```
public class Foo {
  public Swede gert = new Swede();

  public void Bar() {
    Swede stina = new Swede();
    stina.Nationality = "Norway"; // assignment to non-static field

    Console.WriteLine(gert.Nationality); // prints "Norway"
  }

  public class Swede {
    static string nationality = "Sweden"; // This field is static

    public string Nationality { // This property is non-static
      get { return nationality; }
      set { nationality = value; }
    }
  }
}
```

Figure 9: What looks like a non-static assignment to an object property does in fact mutate a static field, which is a clear side-effect of `Bar()`.

In the example in Figure 9, the method `Bar()` assigns to the field `Nationality` of the fresh object `stina`. Because the property `Nationality` is non-static it, it might look like a pure action at a first glance. But since the property writes method `get` itself modifies a static field, the `Bar()`'s assignment to `Nationality` is in fact a side-effect, which alters all `Swede` objects' `Nationality`. Therefore, it is not enough to check if a property is static in order to determine whether a read or write is pure, but the property's `get` and `set` have to be checked as well.

4.4 Method overloading

While method overloading is not unique to object oriented programming, it seems appropriate to mention here as well. Method overloading is when there multiple method implementations share the same method name but have different function signatures, i.e. return types, parameter types and/or number of parameters [1]. One could consider all overloads to be the same method *m*, since they share the same name. However, this would make it difficult to calculate *m*'s purity level, since one overload of *m* could be pure, while another overload is impure. Which purity level should then be given to *m*?

Instead, when given an overloaded method named *m* the analysis treats each overload of *m* as a unique method, independent of other overloads. This way overloads with different purity levels can co-exist in the result of an analysis. Because overloads by definition share the same name, in order to differentiate them in this thesis the notation m_1, m_2, m_3, ... is used, where m_1 denotes the first implementation of the method *m*, m_2 denotes the second, and so on.

5 The analysis method

There are five kinds of functional purity levels for: *pure*, *locally impure*, *parametrically impure*, *impure*, and *unknown*. The latter was added for the case where the purity level of a method cannot be determined.

Traverse the Abstract Syntax Tree (AST) and build the *dependency set* for each function, i.e. the set of calls inside a function. If during the traversal a call or reference to a compiled method or a field is found, mark the caller function's purity as *unknown*.

Since object constructors can perform impure actions just like methods, they are in this section included in the term "function".

Because of the problem with inheritance and method overriding discussed in subsection 4.1, when going through the AST and building the dependency set, if any object parameter's method is overridden by any of its subclasses, add all the overridden methods to the calling function's dependency set. 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. The purity of each function is initialized to *pure*, except those that were explicitly marked *unknown*.

Let the *working set W* be the set of all functions with empty dependency sets. Whenever a function's dependency set becomes empty, that function is added to *W*. Calculate the

purity level for each function in *W* as described in the checklist in subsection 5.1 below. Then, for each function *f* in *W*, propagate the impurity of those with purity level *impure* or *unknown* to the functions dependant on *f*, "contaminating" them. Remove *f* from the dependency sets of all functions that depend on *f*, as well as from the working set *W*. Add functions that now have empty dependency sets to *W*. Repeat this process until there are no more changes to the lookup table, in which case the analysis is complete. Each function in the lookup table will now have been marked with its corresponding purity level. Because the purity level of each function was initialized to *pure*, any function who's purity level was unaffected by the analysis will be marked *pure* at the end of the analysis.

Ignoring the purity level *unknown*, the other four purity levels can be ordered in terms of impurity, from least impure to most impure, like so: *pure* $<$ *locally impure* $<$ *parametrically impure* $<$ *impure*. A method *m*'s purity level should during the analysis only be updated if the new purity level is less impure than *m*'s current one. For instance, if in an arbitrary instance during the analysis, *m* has the purity level *parametrically impure*, and a check determines its purity level to be *impure*, it's purity level should be updated since *impure* is more impure than *locally impure*. However, if for instance another check says that *m* is *locally impure*, its purity level should not be updated since *locally impure* is less impure than *impure*.

5.1 Checklist to determine the purity level

The following is how to determine the purity level *p* for the currently analyzed method *m*.

1. If any object field or property of the currently analyzed method *m*'s object is read from or modified, mark *m* as *locally impure*.

2. If *m* calls a locally impure method belonging to *m*'s object (i.e. `this`) *m* is marked *locally impure*.

3. If the method *m* reads or modifies a static field of any class or object, *m* is marked *impure*. This is because reading a static field is a non-deterministic action, and modifying a static field is a side-effect since it mutates the field for all instantiations of that object's class.

4. If the currently analyzed method *m* calls an input parameter's method m_p and m_p is overridden by any locally impure and/or parametrically impure method, m_p is temporarily marked with the impurities of all the overriding methods in the

context of the analysis of the current method *m*. If m_p is overridden by any impure method, *m* is permanently marked as impure.

5. If the analyzed method *m* modifies an input parameter of reference type, mark *m* as *parametrically impure*. This could be done in a couple of ways:
   - By calling an object type parameter's method that has been marked as *locally impure*.
   - By passing an object type parameter as an argument to a method that has been marked as *parametrically impure*.
   - By directly mutating a parameter object's field or property, or the cell of a parameter array.

   If *m* does at least one of the above, mark it as *parametrically impure*.
6. If a method returns `this` or passes it as an argument to a function it marked *locally impure* since it is dependent on the state of its object, making it non-deterministic.
7. Any method that raises an event is marked *impure*.
8. Any method that raises an exception is marked *impure*.
9. Any method mentioned in the two lists of impure built-in C# methods in subsection 3.4 is marked *impure*.

We do not have to explicitly handle the case mentioned in subsection 4.2 where fresh objects are modified with their own locally impure methods due to the fact that our definition of local purity includes determinism. Because if a non-fresh object *o* is modified with its locally impure method, then we must have read the pointer to *o* from a field outside the method, meaning that *m* is non-deterministic and therefore locally impure. This is why we only propagate the purity level of *impure* functions to their callers, and not *local impurity* or *parametrical impurity*. There are however two exceptions to this, which are covered by previously mentioned checks in our analysis:

- The case of *o* being a static field of an instantiated object has to be checked.
- The case of *o* being a parameter to *m* has to be checked because this is parametrical impurity.

5.2 Example

Figure 10 contains a simple implementation of a linked list in C#. In order to illustrate all types of impurities, the implementation contains some odd design choices.

The method `Length()` takes a `LinkedList` as input and returns its length. This method is deterministic since it only depends on its input argument, and it is side-effect free since it does not mutate any value that exists outside its scope, including its parameter object. Therefore, `Length()` is pure.

`Add()` appends an `Object` to the end the list. It is not deterministic since it reads from and mutates the state of `this` in multiple locations. For instance, the first location where it modifies `this` is where it assigns a `new Node` to `head`, which is a field that is visible outside the method. Therefore, `Add()` is locally impure – it depends on and/or mutates the state of its object.

`Remove()` deletes an item at a given index from a `LinkedList` which is passed as an argument to the method. To simplify the method and reduce its size, its input list is assumed to be non-empty and the index value is assumed to be a valid position inside the list. `Remove()` only operates based on its input and is therefore deterministic. However, it does modify fields of the parameter `list` and is therefore parametrically impure.

The first overloading of the method `Concatenate()` modifies the static class field `numberOfConcatenates` and therefore has a side-effect, making it impure. The second overloading of `Concatenate()` depends on the former method, and is therefore also impure. The static field `numberOfConcatenates` was added to illustrate the propagation of impurity.

```
public class LinkedList
{
  private Node head;
  private Node tail;

  // Static counter used by Concatenate()
  public static int numberOfConcatenates;

  // Returns length of list
  public static int Length(LinkedList list)
  {
    Node current = list.head;
    int length = 0;

    while (current != null)
    {
      length++;
      current = current.next;
    }
    return length;
  }

// Removes item at an index from list.
// Assumes that list is non-empty and
// that index is non-negative and less
// than list's length
public static void Remove(int index,
    LinkedList list)
{
  if (index == 0) {
    list.head = list.head.next;
  }
  else {
    Node pre = list.head;

    for (int i = 0; i < index - 1; i++)
    {
      pre = pre.next;
    }
    pre.next = pre.next.next;

    // If index referes to the last element
    if (index == Length(list))
    {
        list.tail = pre;
    }
  }
}

// Appends data to the list
public void Add(Object data)
{
  if (LinkedList.Length(this) == 0)
  {
    head = new Node(data);
    tail = head;
  }
  else
  {
    Node addedNode = new Node(data);
    tail.next = addedNode;
    tail = addedNode;
  }
}

// Concatenates two lists by
// appending a list to the end of
// this list
public void Concatenate(LinkedList
    list)
{
  if (head == null) head = list.head;
  else
  {
    this.tail.next = list.head;
    this.tail = list.tail;
    list.head = this.head;

    // This gives the method a
    // side-effect
    numberOfConcatenates++;
  }
}

// Overloading of Concatenate that
// allows passing both lists as
// parameters
public static void Concatenate(
    LinkedList l1, LinkedList l2)
{
  l1.Concatenate(LinkedList l2);
}

private class Node
{
  public Node next;
  public Object data;

  public Node() { }

  public Node(Object data)
  {
    this.data = data;
  }
}
}
```

Figure 10: Simple implementation of a linked list. For the sake of this example, it contains some odd design choices.

The analysis starts off by building the dependency set for each function and setting all purities to *pure*, as seen in Table 1.

f	D_f	p
`Length()`		*pure*
`Remove()`	`Length()`	*pure*
`Add()`	`Length()`	*pure*
$\texttt{Concatenate}_1()$		*pure*
$\texttt{Concatenate}_2()$	$\texttt{Concatenate}_1()$	*pure*

$W = \{\texttt{Length()}, \texttt{Concatenate}_1()\}$

Table 1 Initial state of the lookup table after computing each function's dependency set. The working set W is the set of all functions with empty dependency sets.

The working set W is the set of all functions in the lookup table with empty dependency sets, which in this case is `Length()` and $\texttt{Concatenate}_1()$ (the first overloading of `Concatenate()` in Figure 10). We now go through each item in the checklist in subsection 5.1 for each method in W, and check which items apply to the methods:

Since none of the items in the checklist apply to `Length()`, its purity level remains *pure*. As for the method $\texttt{Concatenate}_1()$ the items 1, 3 and 5 apply to it. Item 5 since $\texttt{Concatenate}_1()$ modifies the fields `list` and `tail` of its input parameter, item item 1 since it modifies its own `head` and `tail`, and item 3 since `Concatenate()` modifies the static class field `numberOfConcatenates`. Thus $\texttt{Concatenate}_1()$ should marked with the purity levels *locally impure*, *parametrically impure* and *impure*, and since *impure* is the impurest of the three the method gets marked *impure*.

Since $\texttt{Concatenate}_1()$'s purity level was marked *impure* its purity level is propagated to its callers, which in this case is $\texttt{Concatenate}_2()$. $\texttt{Concatenate}_1()$ is then removed from the dependency set of $\texttt{Concatenate}_2()$, and `Length()` is removed from `Add()` and `Remove()`'s depenedency sets. The methods `Length()` and $\texttt{Concatenate}_1()$ are now removed from the working set W, and `Remove()`, `Add()` and $\texttt{Concatenate}_2()$ are added to W, as their dependency sets now are empty. At this point the state of the lookup table is as shown in Table 2.

f	D_f	p
`Length()`		*pure*
`Remove()`		*parametrically impure*
`Add()`		*pure*
$\texttt{Concatenate}_1()$		*impure*
$\texttt{Concatenate}_2()$		*impure*

$W = \{\texttt{Remove()}, \texttt{Add()}, \texttt{Concatenate}_2()\}$

Table 2 `Remove()` and $\texttt{Concatenate}_1()$'s purity levels have been updated, and after analysing them they are removed from the dependency set of their callers. Also $\texttt{Concatenate}_1()$'s purity level has been propagated to $\texttt{Concatenate}_2()$.

Now we perform the same actions again with the new working set $W = \{\texttt{Remove()}, \texttt{Add()}, \texttt{Concatenate}_2()\}$, starting with `Remove()`. Out of the items in the checklist in subsection 5.1, item 5 applies since `Remove()` directly mutates its input parameter `list` in multiple locations. Therefore `Remove()` is marked *parametrically impure*.

As for `Add()` item 1 applies to it since `Add()` modifies its object in multiple locations, for instance by assigning to the field `tail`, and so `Add()`'s purity level is set to *locally impure*.

Looking at $\texttt{Concatenate}_2()$, none of the items in the checklist apply, and so it remains its purity level *impure* that was propagated to it from $\texttt{Concatenate}_1()$.

`Remove()`, `Add()` and $\texttt{Concatenate}_2()$ are then removed from W.

f	D_f	p
`Length()`		*pure*
`Remove()`		*parametrically impure*
`Add()`		*locally impure*
$\texttt{Concatenate}_1()$		*impure*
$\texttt{Concatenate}_2()$		*impure*

$W = \{\}$

Table 3 `Add()`'s purity level has been updated to *locally impure*. Since W is now empty, this is the final result of the analysis.

At this point the working set W is empty, and so there can be no more changes to the lookup table. Therefore the analysis stops here. The final result is thus what is shown in the lookup table in Table 3. Looking at the table we can see that `Length()` is the

only truly pure method out of the five. Thus we can conclude that `LinkedList`'s total purity level is 1/5 = 20% functionally pure.

6 Implementation of the analysis tool

Because there was a clear goal for the software and it's requirements, test driven development was used when implementing the analysis tool in C#.

Because .NET has many libraries in the form of pre-compiled assemblies, the source code of the functions used in these libraries is not always available. Because of this, the purity level "unknown" has to be added for when we cannot compute a function's purity level.

The code analysis tool is built using .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. Upon calling the program it can be provided either with a single file to be analyzed, or a directory. In the second case, all C#-files in the directory and all sub-directories are analyzed as one program.

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.

The tool is not able to determine the purity of methods that invoke delegate functions. Delegate functions are higher-order functions that invoke methods passed to them as arguments. Invoking delegate functions is done relatively sparsely in C# and handling them would require a disproportional amount of extra work. The delegate functions themselves however are analyzed just like regular methods.

Something that was realized while implementing the tool is that the C# compiler does not perform tail call recursion. Recursion is a very common technique when it comes to functional programming. The initial implementation of the program built the lookup table by calculating each function's dependency set recursively. When analyzing larger code bases the program crashed due to an immense memory usage. Therefore, its recursive implementation had to be changed to an iterative one which worked C# better.

TODO

7 Results and discussion

The implemented analysis program works backwards through the analyzed program's function dependencies and for each function visited its functional purity level is calculated. The purity levels *impure* and *unknown* also get propagated from callee to caller. Out of all nine items in the checklist in subsection 5.1, the following three were implemented:

- item 3: "If the method *m* reads or modifies a static field of an object, *m* is marked *impure*".
- item 8: "Any method that raises an exception is marked *impure*".
- item 9: "Any method mentioned in the two lists of impure built-in C# methods in subsection 3.4 is marked *impure*".

The reason why three out of the total nine checks were implemented was due to the sheer amount of time that the implementation took. After implementing the whole lookup table and all of its functionality, calculation of each method's dependency set, propagation of impurities from callee to caller, etc. there was not time left to implement every item in the checklist in a reasonable amount of time, and so the three items above were chosen. Because of this, the program is unable to detect local and parametrical impurity, and can only detect full impurity caused by side-effects.

Due to the choice of including determinism in the definition of functional purity, which is not included in some definitions, the analysis could be somewhat simplified. The main advantage to including determinism is that we don't have to explicitly check whether potentially global (i.e. static) object fields are read from or modified by a function f, in order to determine f's purity. As a long as a static variable appears in f we know that f is non-deterministic since it depends on a global state. On the other hand, if determinism were ignored we would have to check whether the static field is modified by f or not – or by some method called by f – in order to determine f's purity, which would be even trickier.

The analysis does not handle recursion, which if a recursive program is analyzed could cause it finish before all functions' purities have been calculated, because no more functions will be added to the working set. Support for recursive functions can be added by searching for independent strongly connected components [21].

The analysis uses a blacklist of built-in impure C# functions in order to spot functions that handle impure actions such as randomness and I/O. There may be other impure built-in similar functions that are not in the blacklist.

An alternative way to compare programs' total purity levels, i.e. the ratio between number of pure methods and the total number of methods, could be to profile each program and weight the purity level of each method depending on how often that method is called. For example, let's say that only 10% of program *A*'s methods are pure compared to program *B* where 20% of its methods are pure. Now say that *A*'s pure methods make up 90% of the total function calls during execution and *B*'s pure methods only are called 10% of the times during execution. In this case, one could argue that *A* is purer than *B* since it makes more calls to pure methods than impure ones compared to *B*, even though *B* has a higher ratio of the number of pure methods.

Because most methods use built-in C# methods for which the source code is not necessarily public, a majority of the analyzed methods get the purity level *unknown*. One way of tackling this was to search online for the source code of classes like `List`, run the analyzer on them and permanently store the calculated purity level for each method in a dictionary in the program, so that whenever a call to a built-in `List` method is made its purity level (if it is not *unknown*) can be retrieved from the dictionary. However, this relies on the fact that the source code for a particular built in method is available online, and the process of searching for its code, running the analyser on it and adding any known result to the program's dictionary is cumbersome.

Since the implementation is unable to detect local and parametrical impurity it cannot for certain determine the purity of any program. The implementation does however identify true impurity – the impurest purity level – and does therefore compute a lower bound to a program's impurity, which is equivalent to an upper bound to the program's purity level. The implementation of the analysis does not however give a definite lower bound to a program's purity level. This is due to the fact that all methods' purity levels are initialized to *pure*, and the purity level only changes if the analysis detects a non-pure trait based on the checklist in subsection 5.1. Since not every item in the checklist is implemented, we cannot know if a method was marked *pure* in the result of an analysis because it truly was pure, or because it had a non-pure trait that the implementation cannot detect.

One could argue that if choosing between computing an upper or a lower bound to a program's purity level, a lower bound is more interesting to a programmer. However, due to the nature of the analysis method this requires fully implementing the method.

Another approach to handling called methods with unknown purity – as opposed to marking their purity level *unknown* – would be to simply assume that any unknown method's purity level always is *impure*, or to assume that it is always *pure*. The more conservative approach of assuming that any unknown method is impure would mean that the upper bound for the program's total possible purity level would be moved down, which could potentially give a lower bound that is too low, since some of the unknown

methods that were marked *impure* could have been pure. Since the incomplete implementation does not give a lower bound, the assumption of impurity will have no effect on the lower bound.

For the same reason, assuming that any unknown method is *pure* will also not affect any lower bound of the program's total purity level. The upper bound for the program's total purity level will in this case be the same as the *unknown* approach. However, this assumption will be misleading with a complete implementation since it will potentially give false positives that indicate that some methods are pure when they are not. Therefore, since it clearly indicates to the user where the purity level could not be determined, the purity level *unknown* seems to be the best choice here.

7.1 Results from scanning implementation of linked list

Following is the result of running the analyzer on the linked list program in Figure 10. By looking at the result from running the implementation of the analysis and comparing it to the expected result in subsection 5.2 we can evaluate its performance.

```
METHOD                      PURITY LEVEL
-------------------------------------
LinkedList.Length           Pure
LinkedList.Add              Pure
LinkedList.Remove           Pure
LinkedList.Concatenate      Impure
LinkedList.Concatenate      Impure
```

Figure 11: Result from running the analyzer on the implementation of linked list in Figure 10.

As seen in Figure 11 the analyzer correctly identifies the impurity of both overloads of `Concatenate()`, who's implementations can be seen in Figure 10. The analyzer has thereby successfully spotted the first overload's side-effect and propagated its impurity to the second overload, which depends on the first.

However, the three other analyzed methods `Length()`, `Add()` and `Remove()`'s purity levels – which are always initialized to *pure* in the lookup table – were unchanged by the analysis. As seen in Table 3, `Length()`'s purity level should be unaffected and remain *pure*. However, `Add()`'s and `Remove()`'s purity levels are not identified, since the implementation of the analysis tool only can detect methods with maximum impurity level *impure* – i.e. item 3 in the checklist – and not local or parametrical impurity.

Therefore, what we can say for sure about the linked list implementation based on the

result in Figure 11 is that its purity level is *at most* three pure methods out of a total of five methods, i.e. it is $\leq 3/5 = 60\%$ functionally pure. Since the true purity level of 20% that we expect from subsection 5.2 is less than or equal to 60%, the upper bound is correct.

TODO

8 Related work

8.1 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 [21]. It classifies functions into being functionally pure, or one of three levels of functional impurity [21]. 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 [21].

Their definition of purity uses *referential transparency*, as it implies purity [21]. Referential transparency means that an expression always produces the same value when evaluated [21]. This means that a referentially transparent function could always be replaced with it's output without altering the program's behaviour in any way [21].

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 [21]. The dependency set is the set of functions being called by f and is constructed by parsing the program's Abstract Syntax Tree [21].

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 [21]. 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 [21]. 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" [21]. 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 [21]. 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 [21]. Then remove f from the dependency set of each function depending on it [21]. If f has the highest impurity level, remove the entire dependency set of each function depending on f [21]. 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 [21]. Simplify their dependency sets by removing their dependency on each other from their dependency set [21]. Repeat this process until there are no more changes to the lookup table [21].

The foundation for the analysis method used in this paper is based off the method developed by Pitidis and Sagonas: The approach with using a lookup table to store the intermediate states of the analysis, dependency sets and propagation of impurity from callee to caller using dependency sets. However, the analysis by Pitidis and Sagonas is intended for programs written in Erlang, which is a functional programming language. Also, their analysis gets facilitated by Erlang's BIFs since impure actions can only be performed through them, and so they only need to propagate the impurity of the called BIFs to the callers, and therefore do not need to perform any intermediate analysis on other functions. Since no equivalence to BIFs exists in C#, our analysis can't only comprise of propagating impurities from callee to caller but also needs to check each function analysed to see if that function performs any impure action.

8.2 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 [20]. The system uses the properties *freshness* and *locality* to increase the system's ability to classify methods as pure [20]. An object is fresh if it is newly allocated inside a method [20]. An object's locality is its local state [20]. Pearce's definition of a pure method is one that does not assign (directly or indirectly) to any field that existed before the method was called [20].

The system uses uses the purity annotations `@Pure`, `@Local` and `@Fresh` [20]. The annotation `@Pure` indicates that a method is pure [20]. `@Local` indicates that a method only modifies an object's locality [20]. `@Fresh` indicates that a method only returns fresh objects [20]. These three annotations are modularly checkable, i.e. one method's purity annotations to be checked in isolation from all other methods [20].

The system consists of two parts, *purity inference* and *purity checker* [20]. 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 [20]. 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 [20].

Pearce's definition of functional purity does not include determinism, unlike the one used in this paper. The purity level *locally impure* used in this thesis is based on Pearce's `@Local` attribute. Moreover, the solution to method overriding used in this paper in subsection 4.1 is also based on the ideas developed by Pearce. However, Pearce's approach requires that the program is modularly checkable, i.e. that each method's purity can be evaluated independently from all other methods, and uses annotations to achieve this [20]. Pearce does not introduce an *unknown* purity level for called methods (referred to as methods from external packages) that are not analyzed, but assumes conservatively that such methods are always impure [20].

8.3 Purity and Side Effect Analysis for Java Programs

Similarly to Pearce, Alexandru Sălcianu and Martin 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, i.e. does not include determinism [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].

Similarly to this thesis, Sălcianu and Rinard explicitly mark parts of a method that are potentially accessed by unknown code, and deal with them conservatively. Their analysis uses a points-to graph to compute the purity of each method, unlike the method in this thesis that uses a lookup-table and checklist.

8.4 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]. If a method is written in this subset of Java and its parameters are immutable, 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]. This is similar to the approach with a list of impure built-in C# methods mentioned in subsection 3.4.

Finifter et al. use the same definition of functional purity that is used in this thesis. The main disadvantage with Finifter et al.'s approach is that it requires the code to be written in a subset of Java code in order to be able to analyze the code.

8.5 Dynamic Purity Analysis For Java Programs

[27] TODO

9 Conclusion and Future Work

9.1 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 deterministic. There are many benefits of using functionally pure methods in object oriented programming. Pure methods are generally easier to reason about due to their lack of side effects. Moreover, pure methods are easier to test, debug and maintain. The ability to automatically determine the level of purity in a given C# program can help programmers to write less impure code, and thereby reap previously mentioned benefits.

9.2 Future work

- Tool can point out exact position of impurity and suggest improvements to increase the purity. Because purity is determined by passing the purity level of impure functions to their callers, this could be implemented by also passing the identifier of the impure function to the caller.

References

[1] J. Albahari and B. Albahari, "C# 6.0 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] S. N. Khoshafian and G. P. Copeland, "Object identity," *ACM SIGPLAN Notices*, vol. 21, no. 11, pp. 406–416, 1986.

[8] M. Michaelis, *Essential C# 4.0*. Addison-Wesley Professional, 2018.

[9] Microsoft, "PureAttribute class," accessed 2020-10-07. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/debug-trace-profile/code-contracts#purity

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

[11] ——, "Introduction to the C# language and .NET," 2015, accessed 2020-09-28. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework

[12] ——, "Properties (C# programming guide)," 2017, accessed 2020-05-12. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties

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

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

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

[16] ——, "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

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

[18] 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/

[19] 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.

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

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

[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] University of Cape Town and individual contributors, "Object-oriented programming in Python," 2014, accessed 2020-09-28. [Online]. Available: https://python-textbok.readthedocs.io/en/1.0/Classes.html

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

[27] 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.

[28] D. Yan, "C# functional programming in-depth (13) pure function," 2019, accessed 2020-05-15. [Online]. Available: https://weblogs.asp.net/dixin/functional-csharp-pure-function