

# **Type Annotation Analysis Using the .NET Compiler Platform**

by

**Theodore Sill**

A Project Report Submitted  
in  
Partial Fulfillment of the  
Requirements for the Degree of  
Master of Science  
in  
Computer Science

Supervised by

Dr. Matthew Fluet

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences  
Rochester Institute of Technology  
Rochester, New York

May 2017

# Dedication

To my wife Aileen,  
for believing in me when I did not believe in myself.

# Acknowledgments

I am grateful for the wonderful faculty and staff at RIT who spend countless hours curating, mentoring, and motivating. I am indebted to Paychex for providing tuition assistance, and allowing me to alter my schedule to accommodate graduate school. Finally, I am thankful for my classmates and coworkers who have shared insights, commiserated, and inspired.

# **Abstract**

## **Type Annotation Analysis Using the .NET Compiler Platform**

**Theodore Sill**

**Supervising Professor: Dr. Matthew Fluet**

Programming language type and runtime systems provide powerful guarantees about the behavior of a program when it is executed. However, they do not always ensure that a program will have the desired runtime characteristics, or that the outcomes will align with the intent of the programmer. It is often necessary to provide additional assurances that a program is correct in this sense. Type annotation analysis frameworks are static analysis tools that allow programmers to add additional information in the form of type annotations, and thus express their intent in such a way that it may be automatically verified.

In the past, creating a type annotation analysis tool would have been a large undertaking. Most compilers were black boxes which accepted source files as input and produced an executable as output. Those wishing to make use of a program representation, such as an Abstract Syntax Tree (AST), were forced to construct their own. Microsoft opened this black box when they delivered the .NET Compiler Platform (code named “Roslyn”), which exposes several APIs.

In this work, we will explore these offerings and describe how they were leveraged to build a type annotation analysis tool for C#. We call this tool “Sharp Checker” in homage to the Checker Framework, which is a full-featured solution for Java. The contribution of

this work is to translate the mechanisms of annotation processing at work in tools like the Checker Framework to the Visual Studio IDE, where users receive feedback immediately as they type and upon compilation. Sharp Checker may be installed as a NuGet package, and the source code is available on GitHub. We have demonstrated Sharp Checker's extensibility and usefulness by implementing the Encrypted, Nullness, and Tainted annotated type systems and applying them to the Sharp Checker source code, as well as publicly available applications. In the process, we discovered and corrected several bugs, while gaining insights into the properties which may be enforced by type annotation analysis.

# Contents

<b>Dedication</b> . . . . .	<b>ii</b>
<b>Acknowledgments</b> . . . . .	<b>iii</b>
<b>Abstract</b> . . . . .	<b>iv</b>
<b>1 Introduction</b> . . . . .	<b>1</b>
1.1 Type Annotations . . . . .	1
1.2 Static Analysis . . . . .	3
1.3 Roadmap . . . . .	5
<b>2 Background</b> . . . . .	<b>6</b>
2.1 Project Motivation . . . . .	6
2.2 Type Annotation Analysis . . . . .	10
2.3 Checker Framework . . . . .	12
2.4 JQual . . . . .	14
2.5 .NET Compiler Platform . . . . .	15
<b>3 Design</b> . . . . .	<b>18</b>
3.1 Checker Framework . . . . .	18
3.2 Design Decisions . . . . .	20
<b>4 Implementation</b> . . . . .	<b>23</b>
4.1 Framework Features . . . . .	23
4.2 Annotated Type Systems . . . . .	25
4.3 Architecture . . . . .	29
4.4 Challenges . . . . .	30
<b>5 Analysis</b> . . . . .	<b>32</b>
5.1 Framework Introspection . . . . .	32

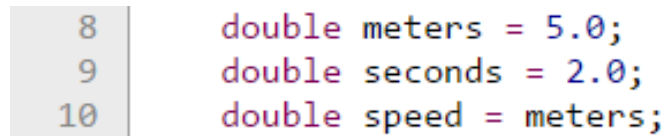
5.2	Applying Sharp Checker . . . . .	32
<b>6</b>	<b>Conclusions . . . . .</b>	<b>36</b>
6.1	Current Status . . . . .	36
6.2	Future Work . . . . .	37
6.3	Lessons Learned . . . . .	38
	<b>Bibliography . . . . .</b>	<b>39</b>
<b>A</b>	<b>UML Diagrams . . . . .</b>	<b>41</b>
<b>B</b>	<b>Code Listing . . . . .</b>	<b>44</b>
<b>C</b>	<b>User Manual . . . . .</b>	<b>47</b>
C.1	Use Existing Checkers . . . . .	47
C.2	Build Your Own Type System . . . . .	49
C.3	Extending Sharp Checker . . . . .	51

# Chapter 1

## Introduction

### 1.1 Type Annotations

Type systems aid us in avoiding trivial blunders, and in building and enforcing our own abstractions. Type annotations and analysis frameworks effectively strengthen a type system by rejecting additional programs which are believed to contain semantic errors. A more detailed description of the rationale for using type annotation analysis is presented in Section 2.2. Let us first consider a concrete example which will provide context for the abstract treatment. Figure 1.1 presents a Java code snippet intended to calculate velocity.



```
8      double meters = 5.0;  
9      double seconds = 2.0;  
10     double speed = meters;
```

Figure 1.1: Incorrectly calculating velocity in Java

Did you notice the logical error? We have forgotten to divide by “seconds”. We could prevent this type of mistake by introducing a new type. In Figure 1.2 we see a “Velocity” class. It has two fields which hold the “meters” and “seconds” values, and a “GetVelocity” method which will perform the required computation when invoked. This prevents us from making the aforementioned mistake, but what happens when we want to compute acceleration? If we apply this approach broadly, we may end up with a large collection of classes which serve no purpose other than enforcing simple constraints.



```

8      Velocity vel = new Velocity(5.0, 2.0);
9      double speed = vel.GetVelocity();
10     }
11
12     public class Velocity{
13         private double meters = 0.0;
14         private double seconds = 0.0;
15         public Velocity(double m, double s)
16         {
17             this.meters = m;
18             this.seconds = s;
19         }
20         public double GetVelocity()
21         {
22             if(seconds != 0.0)
23             {
24                 return meters / seconds;
25             }
26             return -1.0;
27         }
28     }

```

Figure 1.2: Using Java types to enforce a velocity calculation

Another approach is to add additional information in the form of type annotations. The Checker Framework is an existing type annotation analysis framework for Java which recognizes and enforces type annotations. A more detailed discussion of the offerings of the Checker Framework is presented in Section 2.3. In Figure 1.3 we have made our intent clear, and the Checker Framework presents an error indicating that there is a mismatch between our intent and the way in which the program will be executed. This is much more concise than creating a “Velocity” class. We also avoid the potential performance degradation resulting from allocating objects on the heap, instead of using primitive types which are maintained on the stack. With type annotations, the verification is occurring at compile time, so there is no runtime cost.

In Figure 1.4 we have corrected the logical error and the compilation is now successful.

8	@m double meters = 5.0 * UnitsTools.m;
9	@s double seconds = 2.0 * UnitsTools.s;
10	@mPERs double speed = meters;

Executed command: javac -processor org.checkerframework.checker.units.UnitsChecker afile.java

No.	Type	Description
1	error	Error: [assignment.type.incompatible] incompatible types in assignment. found : @m double required: @mPERs double

Figure 1.3: Catching a logical mistake in a velocity calculation

As an added benefit, our intent remains embedded in the code. When editing occurs in the future, we can simply execute the analysis again to ensure that we have not introduced a regression defect. Unlike comments, which may quickly become outdated, we benefit from being able to automatically verify our intent. Unlike unit tests, we do not have to look elsewhere for the verification code. It is immediately evident, and often makes our code more readable.

8	@m double meters = 5.0 * UnitsTools.m;
9	@s double seconds = 2.0 * UnitsTools.s;
10	@mPERs double speed = meters / seconds;

Figure 1.4: Correct velocity calculation with type annotations

## 1.2 Static Analysis

Type annotation analysis frameworks fit broadly into the category of static analysis tools. There are many reasons why an organization may wish to enforce stricter code quality rules than those established by compilers. For example, you may have standards which dictate style guidelines or idiomatic approaches to expressing certain operations. Static analysis tools can help to enforce these domain or application specific rules.

There are existing static analysis tools for C# such as Microsoft's FxCop and SonarSource's SonarQube. These tools generally present a report of violations and code metrics

after developers have committed their code to a source control repository, and a build has been executed. In Figure 1.5, we see sample output from SonarQube where a warning is presented about the exact comparison of floating point numbers. As a result of the internal representation of floating point numbers, such a comparison may have unexpected results, but it is permitted by the C# type system. Existing static analysis tools generally present warnings about constructs which fall into this gray area. Unlike type annotation analysis, no additional information may be provided by the programmer; the tool simply warns that the use is often bad practice. These tools are at the full featured end of the spectrum. They provide the capability to toggle specific rules and even craft your own. There are also many lightweight tools which generally operate at the level of syntax. These “linting” tools do not require an initial investment of time, but will generally present a large number of false positive warnings.

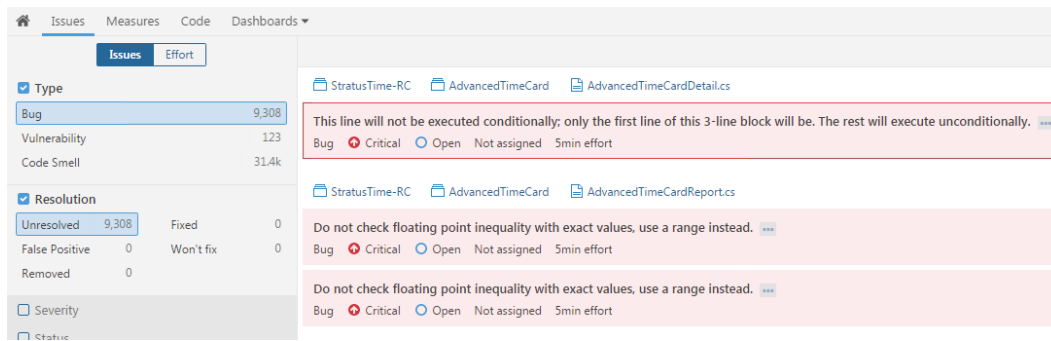


Figure 1.5: SonarQube results executing on commercial codebase

A recent addition to the static analysis space is the Diagnostic API offerings of the .NET Compiler Platform. This allows for the enforcement of rules, like those that are incorporated into SonarQube, and gives feedback immediately in the Visual Studio IDE. The .NET Compiler Platform is discussed in greater depth in Section 2.5.

Type annotation analysis frameworks for Java, like the Checker Framework and JQual, were proven to be useful to programmers and type system designers [11] [13]. We hypothesized that a similar system for C# would be beneficial. Existing static analysis tools for

C# suffer from a lengthy feedback loop, discussed in Section 2.1, as well as imprecision which manifests as false positives. Sharp Checker improves upon these tools by leveraging the .NET Compiler Platform to present feedback immediately in the IDE, and by allowing programmers to express their intent so that their code may be verified with greater precision.

## 1.3 Roadmap

In the following chapters, you will find a description of the design, implementation, and analysis of Sharp Checker. In Chapter 2, we establish the context in which this tool was conceived by providing information about type annotations analysis in general, and by enumerating the capabilities and design elements of the Checker Framework and JQual. We focus on characteristics which are reflected in Sharp Checker. In Chapter 3, we discuss the architecture of Sharp Checker, and in Chapter 4 we describe the implementation of this design. We then explore several sample applications and type systems, and present an analysis of the efficacy of this tool in Chapter 5. Finally in Chapter 6, we offer conclusions and propose future work.

# Chapter 2

## Background

In this chapter we will go into greater detail regarding background material and previous work which made the creation of Sharp Checker possible. Most notably, the research which produced the Checker Framework and JQual provided much of the theoretical foundation for this work, empirical evidence that pluggable type systems are useful in practice, and reference implementations. The .NET Compiler Platform (“Roslyn”) furnished the tools necessary to translate these ideas to C# and to provide immediate feedback within the Visual Studio IDE.

### 2.1 Project Motivation

Many professional software developers are acutely aware of the inherent trade-offs between meeting aggressive deadlines and producing robust, efficient, and maintainable software. There is no shortage of tools available which promise to help users achieve an optimal balance. However, many fail to deliver for a simple reason: the target demographic for these products are those who are short of time. Many of the tools require a significant investment of time to understand the offering, install and configure the software, and evaluate and leverage the results.

Take for instance a static analysis tool such as Microsoft’s FxCop or SonarSource’s SonarQube. These software products are generally configured to execute on a build agent. They present a report of violations after developers have committed their code to a source control repository and a build has been executed (see Figure 1.5 for sample output). Builds

may be triggered by changes to source code or run at scheduled times. In either case, there is enough of a delay that the individual who made a code change will have moved on to another task before the feedback is issued (see Figure 2.1 for a sequence diagram depicting this process). It is generally the responsibility of the developer to review the results and determine if the code which they added has resulted in new violations. If so, they must also determine if the violations represent legitimate issues with the code or if they are false positives resulting from the bias or configuration of the tool. Most developers strive to produce high quality code, and truly appreciate this type of feedback. However, if their managers and coworkers are primarily concerned with the timely delivery of new features, it is improbable that code quality will be given top priority.

It is also common for there to be organizational mechanisms which discourage code changes. Many businesses now leverage the Agile project management methodology. When using this process, teams generally deliver new functionality in short cycles (1-3 weeks) called sprints or iterations. Requirements are given to developers who must deliver the associated functionality with enough time for quality assurance teams to verify that it meets the requirements specified prior to the end of a sprint. It is also expected that some level of regression, integration, and performance testing occur before changes are released to production. Every time a developer makes a code change the validation must be repeated. It is understandable that product owners who advocate for the business, which assumes some level of risk every time a change is made, and project managers, who are primarily concerned with deadlines, would discourage code changes.

So how can the conscientious software developer hope to leverage tools which suggest making code changes? The solution here is again quite simple: feedback must be immediate or nearly so. If the developer is presented with code quality feedback in their IDE or upon building locally, then they may tune their code to meet their personal, team, or organizational standards. This short feedback loop allows for these changes to be made without incurring a business cost which is not commensurate with the long term benefit. This process is distinct from the more protracted processes described above and the two are

contrasted in Figure 2.1.

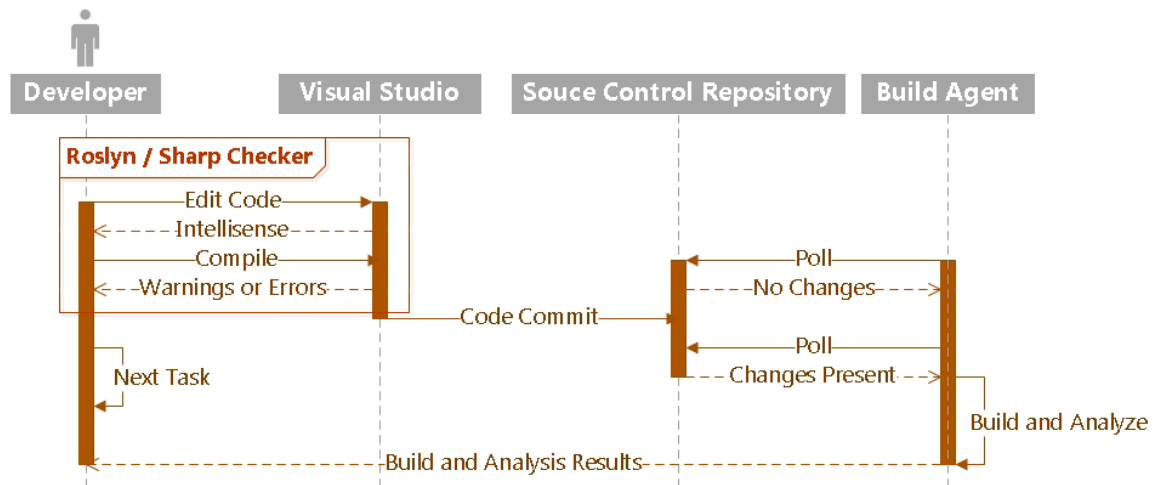


Figure 2.1: Contrasting the traditional static analysis life-cycle with immediate feedback

The need for feedback earlier in the development cycle has been recognized by industry leaders and the advent of the .NET Compiler Platform is an excellent example of an effort to satisfy this demand. Great care has been taken to make tools accessible that enforce general or domain specific rules immediately. Given this platform, we now have an opportunity to evaluate old static analysis tools - while considering their usefulness as reflecting by adoption rates - and determine if they may be translated to this new paradigm.

Two of the common components of the Diagnostic API, which the .NET Compiler Platform provides, are diagnostic analyzers and code fixes. When an analyzer detects a violation it will present a green (warning) or red (error) underline in Visual Studio (VS). When a code fix is associated with this diagnostic, the user may click a light bulb icon to preview the proposed change and optionally apply it. More information about the APIs which the .NET Compiler Platform exposes can be found in Section 2.5.

To become more familiar with these mechanisms we created an analyzer and code fix which validate that the appropriate number of arguments are present in a `String.Format` method invocation (see Figure 2.2). Since the first argument is a string containing tokens to

be replaced by subsequent arguments, the C# type system is unable to verify the appropriate number of arguments have been provided. When too few arguments are present, a runtime exception results. When this diagnostic analyzer was executed on a commercial codebase, two genuine violations were reported. This provided anecdotal evidence that this type of analysis is worthwhile in a real-world setting.

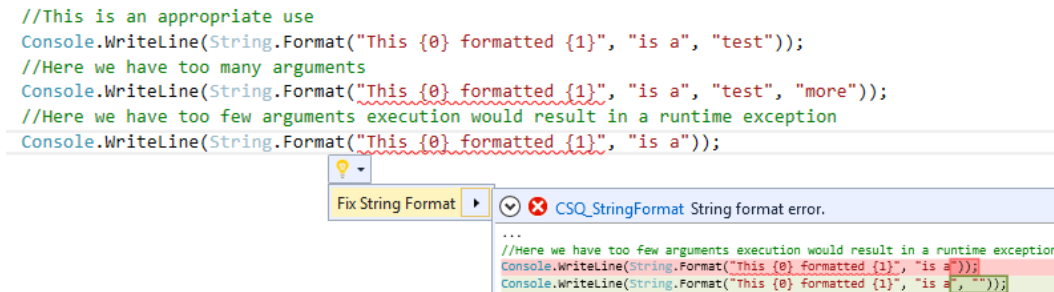


Figure 2.2: Roslyn diagnostic analyzer presenting errors and code fix suggestion in VS

One of the primary benefits of this approach, is that you can leave the rules in place to catch future violations in real time. These may be thought of as a general form of unit test or automatic code review. In other words, when an issue is encountered which might be remediated in a formulaic way, then a new analyzer and code fix may be created. Along with the industry shift to Agile development, has come a focus on continuous delivery. With short iterations and frequent deployments to production, tools for automatically verifying code are becoming ever more essential.

The C# language team at Microsoft leveraged this mechanism, and ported a subset of the original FxCop static analysis rules, further demonstrating their support for this approach to ensuring code quality. They started with those rules which they believed to be the most widely applicable and have prioritized the implementation of others which they feel are still useful. This is one area where they have publicly encouraged community involvement, both to identify important rules and to implement the associated analyzers and code fixes.



## 2.2 Type Annotation Analysis

Most programmers are familiar with the mechanics of using types, but they may not know the significance of type systems. It is not uncommon for a novice programmer to complain about the rigidity of a compiler's type checking when it has rejected their program and returned a type error. What they fail to appreciate, is that most of the time this error is beneficial because it prevents the compiler from generating machine instructions which, when executed, may have unintended or unpredictable consequences.

Occasionally, a program is rejected even though it would exhibit the desired behavior. This follows from the fact that most type systems are sound and not complete. When a program is accepted by a sound compiler, the programmer has a guarantee that the program will move from one meaningful state to another indefinitely, or until it completes and returns a value. A complete type system on the other hand, provides a guarantee that no program which could exhibit the desired runtime characteristics will be rejected, and as a result permits some unsafe programs. Based on the acceptance and use of programming languages, it seems that the programming community has decided that it is generally worth surrendering some expressive power in order to achieve the guarantees afforded by type safety.

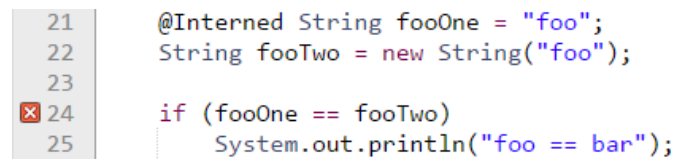
An annotated type system strengthens a type system in the sense that even more programs will be rejected. The annotated types add constraints which must be satisfied in order for the compilation to proceed. This imposes some additional implementation and maintenance costs, but in return we achieve additional confidence that the program we have written will manifest our intent.

It is important to note that annotated type systems operate in a somewhat different way than type systems. Compilers don't support an unchecked mechanism to assert that an object of one type is actually another type. With casting there is runtime checking, so it is never the case that the program is trusted to identify safe conversions. However, this is common with annotated type systems. If you provide a utility method which does a conversion (for example converting dollars to pesos), the analysis framework will trust the

annotations. In general, the properties of the annotated type system may only be enforced if the annotations themselves are correct.

One major feature of type annotation analysis frameworks is that they are generally extensible and multi-purpose. You could encode the properties of an annotated type system with Java or C# types as we saw in Section 1.1, but the implementation would be limited to the specific use, and could not easily be reused. These frameworks are lightweight in that they may be applied without impacting the runtime behavior of the code. This cannot be said of adding new classes to enforce the desired constraints, which may not be an option when dealing with a large existing legacy codebase.

We previously saw the Units annotated type system. To build on our understanding of the features of annotated type systems, let us examine the “Interning” type system. Depending on the programming language, comparison with an equality operator may be interpreted as a reference comparison, and only return true when both variables reference the same location in memory.



```

21  @Interned String fooOne = "foo";
22  String fooTwo = new String("foo");
23
24  if (fooOne == fooTwo)
25      System.out.println("foo == bar");

```

Figure 2.3: Error presented when comparing variables with different type qualifiers, generated with the Checker Framework live demo [8]

In Figure 2.3 we see an example of a type annotation: “@Interned”. This annotation provides additional information about the variable “fooOne”. The Checker Framework, which we will discuss at length in Section 2.3, is equipped to understand this additional information and warn us when we have contradicted ourselves. In this case, comparing an interned string to a string object is unlikely to be what we intended. There are myriad other type systems which may be added to existing type safe languages in this fashion. Another example, the “Nullness” type system, which helps programmers avoid null reference exceptions, will be discussed in the next section.

## 2.3 Checker Framework

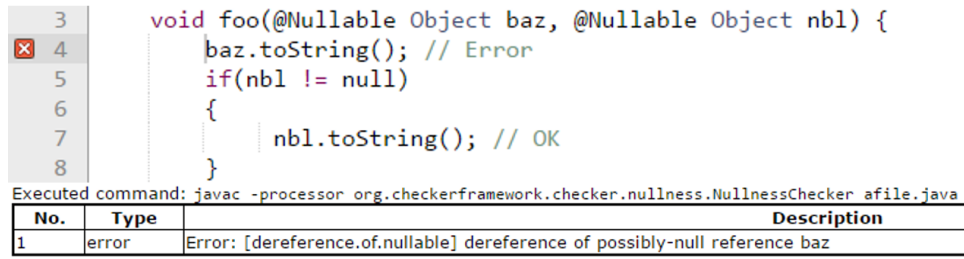
The Checker Framework is a tool for Java which is executed as a batch annotation processor. It was designed to facilitate the implementation of annotated type systems or “checkers” which enforce certain invariants in the target code. The framework comes with a collection of these “checkers” which may be deployed immediately. After instrumenting the target code with annotations and executing the appropriate “checker” the user is presented with feedback which indicates adherence to a specific type system.

Annotations in Java are syntactically represented with an “@” prefix. In the subsequent text we will present them as they appear in code - *i.e.*, without surrounding quotes - to avoid unnecessary clutter. The “javac” Java compiler has a mechanism for annotation processors to plug into the compilation pipeline, and optionally issue warnings or errors. In the case of the Checker Framework, the annotations represent metadata specifying type qualifiers.

One of the checkers distributed with the Checker Framework enforces the “Nullness” type system. Having uninitialized objects is a common source of runtime exceptions in object-oriented languages. The Nullness type system attempts to prevent these exceptions by considering explicit annotations and inferring reasonable defaults. For example, if an assignment statement has a right hand side with an expression to initialize a new object, then that variable will be assigned an annotated type indicating it is not null.

In Figure 2.4 we present a sample execution of the Nullness checker. On line 4, an error is presented when dereferencing a object which may contain a null value. In this case the programmer has explicitly indicated that the variable may be null by annotating the parameter with `@Nullable`. A method invocation might not be worthy of inspection in another type system, but in the case of the Nullness type system, this represents a potentially dangerous operation.

Support for type qualifiers in all the syntactic locations necessary for the full functionality of the Checker Framework was added to Java 8 with JSR 308. Previously there were places where type annotations needed to be enclosed in comments to avoid compilation errors, and in some cases this is still recommended to preserve backward compatibility [9].



```

3      void foo(@Nullable Object baz, @Nullable Object nbl) {
4          baz.toString(); // Error
5          if(nbl != null)
6          {
7              nbl.toString(); // OK
8          }

```

Executed command: `javac -processor org.checkerframework.checker.nullness.NullnessChecker afile.java`

No.	Type	Description
1	error	Error: [dereference.of.nullable] dereference of possibly-null reference baz

Figure 2.4: Demonstrating an unsafe dereference and type refinement with the Checker Framework live demo [8]

The Checker Framework has a stub processing component which allows for more accurate analysis involving code which was compiled without annotations. Without the stub file, conservative assumptions must be made, and false positives in the form of additional warnings and errors will result. As an alternative to stub files, the authors of the Checker Framework also recommend using third party tools such as Nit, JastAdd, and Cascade for inferring annotations in source code [9]. Also, the Checker Framework makes appropriately annotated versions of common libraries available to supplement the stub file functionality.

The Checker Framework provides the capability to create new type annotations and specify their ordering (*i.e.*, subtype relationship) and introduction rules declaratively. Introduction rules define the default type qualifiers for unannotated types in various contexts, as mentioned above in the description of the Nullness type system. The creators' general goal was to deliver a framework which would make defining simple type systems easy, while also granting enough power to make it possible to create cutting-edge type systems [14].

The last major component of the Checker Framework is the mechanism which enforces type rules. In the simplest case, a partial order representing a subtyping relationship is enforced. The Checker Framework also performs local type inference and flow sensitive type refinement. For example, when using the Nullness checker, if a variable has been compared to “null” with the not-equal operator, then the type of that variable may be refined to @NonNull within the block of code guarded by the inequality check (See line 7 in

Figure 2.4).

By the type introduction rules of the Nullness type system, if a literal value is assigned to a variable then `@NonNull` is inferred. However, when an assignment is made to that variable or any method is called, that refined type must be abandoned. This follows from the fact that the assignment could result in a null value, and the method call might have side effects which result in this variable being set to a null value. However, the Checker Framework provides annotations which allow the programmer to indicate that a method should be considered `@SideEffectFree`, `@Deterministic`, or both: `@Pure`. When executing a `@SideEffectFree` or `@Pure` method, the refined types in the context of the invocation may be preserved [9].

The Checker Framework has been under active development for over ten years, and is clearly mature and full featured. We analyzed the source code (which is publicly available), user manual, and published papers in order to guide the direction of Sharp Checker’s development.

## 2.4 JQual

JQual differs from the Checker Framework in that it performs type inference instead of type checking [14]. That is to say, it qualifies all of the types which are being analyzed with an explicit qualifier, when one was specified by the programmer, and otherwise introduces a type qualifier variable. It then performs an analysis to determine if there is a set of qualifiers which may be assigned to the qualifier variables which will satisfy the constraints established by the program. JQual is less reliant on annotated libraries because it performs interprocedural type inference. To contrast this with the Checker Framework, a “checker” visits nodes of the AST and verifies invariants such as those dictating that assignments and method invocations comply with the annotations present or locally inferred.

Despite the differences, JQual does share some of the same architectural components as the Checker Framework. For example, it support similar expressive power for specifying the relationship among type qualifiers. JQual uses a lattice file, the format of which was

established in earlier work on the CQual system [3]. JQual and the Checker Framework were both shown to support many of the same type systems, and their use from an end user perspective is quite similar.

One major contribution of JQual is the implementation of an opt-in flow sensitive analysis. When enabled, instead of conflating the object instances of a class, fields are tracked separately. This allows for different instances to have different type qualifiers. Also, in an effort to further refine the accuracy of their analysis, they introduce context sensitivity to take into consideration the context in which a method is called. This is encoded as a context free language reachability problem [13].

There doesn't seem to be much alignment with the features offered by Roslyn and those required to perform this analysis. Also discouraging the pursuit of this approach in the system we created, was the assertion of Dietl, et al. that JQual does not scale to legitimate real world codebases [11]. However, it may be that these techniques simply do not translate well to Java. Foster et al. make strong claims as to the real world applicability of their earlier system CQual for the C programming language [12].

## 2.5 .NET Compiler Platform

The C# programming language was announced in July 2000, with a compiler implemented in C++. As the language evolved, it became clear that complexity and technical debt present in the compiler codebase was slowing the delivery of new features. Additionally, the landscape had changed since the compiler had initially been conceived. Contemporary workflows demanded immediate feedback in IDEs given only program fragments, language services such as “find all references”, and hooks to facilitate extensibility [10]. This motivated a decision around 2010 to start anew, in the hopes that a redesign would allow the project team to better address future demands. The .Net Compiler Platform (code named “Roslyn”) was implemented in C# and became open source in 2014.

Roslyn exposes the Compiler API, Workspace API, and Diagnostic API. The Compiler API allows developers direct access to the compiler pipeline. Using this API one may

parse source text, and access the resulting AST, or create a complete compilation on the fly, emitting an executable or loading it directly into memory. The Workspace API exposes an object model representing a solution in VS. This is necessary for operations where we must take into account all of the available namespaces or assemblies for a given compilation. For example, to perform a rename operation we must be able to locate all of the references to a given identifier. As described in Section 2.1, the Diagnostic API supports the creation of analyzers which recognize potential code issues and may additionally provide recommended fixes. To leverage this functionality one subscribes to actions which occur during the compilation process. For example, it is common to subscribe to syntax node actions which will be triggered each time a specified type of syntax node is visited. This API exposes the compilation process to a high degree of granularity, which gives clients great flexibility, but also presents a challenge in that this client must handle a large amount of detail. In general, these APIs do not offer a simple mechanism to create general rules like those delivered by many static analysis tools. This is why we feel that there is a niche for Sharp Checker, which abstracts some of the low level detail, and eases the implementation burden on those who wish to enforce such rules. We leverage C# attributes which appear with surrounding square brackets in source - *i.e.*, [NotNull] - and serve the same purpose as the aforementioned Java annotations.

Microsoft has clearly committed to making Roslyn accessible to the general population of developers. They have furnished an “Analyzer with Code Fix” template in the project templates within Visual Studio (VS). When this template is selected, a solution is created with three projects. One is a cross-platform .NET Core project which will contain the analyzer and code fix. This also comes with a “.nuspec” file that is necessary to bundle the functionality as a NuGet package. NuGet is the standard package management solution for C#, and makes sharing assemblies and managing dependencies quite straightforward. More detail will be given about how Sharp Checker leverages NuGet in Chapter 4. The solution also contains a VSIX project. When selecting this project as the startup project and debugging, an experimental hive will be created. This is essentially a second copy of

VS. This second instance has your analyzer and code fix installed, and makes debugging quite easy. The final project that comes preloaded, is a unit test project that contains some helpful boiler plate code for verifying your analyzer and code fix. This template served as the starting point for the implementation of Sharp Checker, which is described in detail in Chapter 4.



# Chapter 3

## Design

### 3.1 Checker Framework

The design for Sharp Checker was heavily influenced by that of the Checker Framework. Having covered the high level components of the Checker Framework in Chapter 2, we will now go into greater detail regarding the internal flow of the analysis. The entry point for the analysis that the Checker Framework performs is the “Process” method of a class which implements the “Processor” interface. The implementing class must also provide a definition of the following methods: “init”, “getSupportedAnnotationTypes”, “getSupportedOptions”, and “getSupportedSourceVersion”. This is fulfilled by an abstract base class called “AbstractTypeProcessor”, which forms the top of the Checker Framework class hierarchy. The abstract class “SourceChecker” extends this class, and “BaseTypeChecker” in turn extends “SourceChecker”. The checkers which verify adherence to a particular type system extend “BaseTypeChecker” or one of its subclasses. Figure 3.1, from the Checker Framework manual, presents a high level view of the major components [9]. The box labeled “Checker Framework” would contain “AbstractTypeProcessor” and “SourceChecker”, and “BaseTypeChecker” would fall into the “Base Checker” box.

The main purpose of these levels of abstraction, is to provide default behavior which is generally sufficient at each level. In accordance with good object-oriented design, the behavior that is least likely to be overridden, and is applicable to many classes, is implemented near the top of the class hierarchy. Case in point, the default behavior of an annotation processor is not to visit all of the nodes of the AST, but instead only public elements [9]. Since

Subtyping Checker	Nullness Checker	Mutation Checker	Tainting Checker	...	Your Checker	
Base Checker (enforces subtyping rules)						Type inference    Other tools
Checker Framework (enables creation of pluggable type-checkers)						<a href="#">Annotation File Utilities</a> (.java ↔ .class files)
<a href="#">Type Annotations</a> syntax and classfile format (“JSR 308”) (no built-in semantics)						

Figure 3.1: A presentation of the components of the Checker Framework from the Checker Framework Manual [9]

this is insufficient for the Checker Framework, once control has been passed to the framework, it performs its own traversal of the AST. This enables the analysis of the body of methods for example. This transition occurs at the level of “AbstractTypeProcessor”, and is never overridden in derived classes. We similarly attempted to push shared functionality into Sharp Checker base classes, to ease the implementation burden on those adding new type systems.

The base functionality furnished by the Checker Framework class hierarchy is quite extensive. In the extreme case a new type system can be created by declaring a new class which extends “BaseTypeChecker”, and is appropriately annotated with the type annotations it will be used to evaluate. In Figure 3.2 you will find an example of this in the Tainting checker distributed with the Checker Framework. This declarative syntax represents a convenience afforded by the framework. However, by the creators’ own admission, they pursued procedural mechanisms first, then only when the use cases justified the effort did they add these conveniences [14]. We have taken this tact with Sharp Checker pursuing the procedural mechanism, and leaving the declarative as future work.

```

package org.checkerframework.checker.tainting;
import org.checkerframework.common.basetype.BaseTypeChecker;
import org.checkerframework.framework.source.SuppressWarningsKeys;
@SuppressWarningsKeys({"untainted", "tainting"})
public class TaintingChecker extends BaseTypeChecker {}

package org.checkerframework.checker.tainting.qual;
//Imports and some repetitive statements elided for brevity
@Documented
@PolymorphicQualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
public @interface PolyTainted {}

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
@DefaultQualifierInHierarchy
@SubtypeOf({})
public @interface Tainted {}

@SubtypeOf(Tainted.class)
@ImplicitFor(literals = {LiteralKind.STRING, LiteralKind.NULL})
@Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@DefaultFor(TypeUseLocation.LOWER_BOUND)
public @interface Untainted {}

```

Figure 3.2: The complete declaration of the Checker Framework Tainting checker

## 3.2 Design Decisions

Early in the process of developing Sharp Checker we faced a decision about how closely we would follow the pattern set forth by the Checker Framework. Their analysis is largely a manual process. Creating their own AST and type annotation mirror data structure, to make the annotations readily available during the analysis, they perform several passes to refine types and support other features. With Roslyn, operating within the confines of the Diagnostic Analyzer API, we would not be able to perform multiple passes in the same way. Roslyn uses data structures which are immutable to enable concurrent processes. It does not provide a mechanism to modify the context it passes to the methods which you have plugged into the pipeline, such that those changes will be preserved in later stages of analysis. That is to say, you can create and modify a copy of the AST, perhaps by adding annotations to syntax nodes, but you cannot pass this back or otherwise overwrite

the analysis context.

Another challenge results from the fact that Roslyn’s Diagnostic API is used to provide real time IDE feedback instead of batch processing output. The experience of the developer is the focus of this design. As a result, a compilation and analysis triggered by a change in VS may be preempted when another modification makes the results obsolete. The Roslyn documentation contained a warning about allocating resources during an analysis with the intent of releasing them later. Since an analysis may be canceled before completion, the later stages of the analysis might never be executed, and you could introduce memory leaks [4].

There are however many benefits to operating within the context of the Diagnostic API. For example, when we subscribe to the action associated with the “InvocationExpression” syntax node, we are notified whenever such a node is visited. This means that our analysis will apply to an invocation expression which occurs at the top level, within the argument to another method invocation, or buried at arbitrary depth within a syntax tree representing an expression (See Figure 5.3 for some samples of this nesting). We also wanted to preserve the immediate feedback within the IDE, without having to carefully manage state throughout the analysis life cycle. For these reasons, we decided to diverge somewhat from the example set forth by the Checker Framework.

JQual also influenced the design of Sharp Checker. Some of the primary contributions of JQual have to do with field and context sensitivity. With Sharp Checker, we are keeping track of the effective annotations at each syntax node, and we are not performing type inference. As a result, we do not have a need for these features. For this reason, those mentioned in Chapter 2, and because we were not able to retrieve the source code, we based the design of Sharp Checker less on JQual than the Checker Framework. However, both provided excellent information regarding annotated type systems and their application.

We thus conceived the design for Sharp Checker. Our goal was to provide a similar extensible class hierarchy to that furnished by the Checker Framework. Reviewing the implementation of the Checker Framework allowed us to identify some key components and

mechanisms. For instance, it became clear that we would need a logical space between recognizing the attributes explicit or inferred, and verifying that they were respected. Operations like type refinement, based on context and assertions, operate within this space. We also identified some of the key design elements of Roslyn itself, and concluded that by allowing Roslyn to manage much of the infrastructure, we benefit from optimizations built into the platform. In the next chapter we go into detail regarding how this design was implemented.

# Chapter 4

## Implementation

### 4.1 Framework Features

The features of the Sharp Checker framework may be grouped into several categories. One category is that of flow properties. These properties will generally hold for all pluggable type systems, and as a result, are implemented within the base classes. For example, when you are invoking a method, if the formal parameters are annotated with a qualifying type, then we need to ensure that at each invocation, the arguments passed have the appropriate qualified type. When an assignment is made to a variable with an annotated type, we need to ensure that the result of the expression on the right hand side of the assignment will have the appropriate annotated type. When a method is overridden in a subclass and the parent implementation had an annotated return type, or annotated parameters, we need to ensure that the overriding method respects these annotations. Finally, when a method is decorated with a return attribute, we need to ensure that the value returned in the body of that method has the appropriate annotated type.

A second major feature of the framework is support for subtyping among annotated types. In the previous paragraph when we mention the “appropriate annotated type”, we are generally referring to an exact match or a subtype. “Subtype” is a somewhat overloaded term. We use it here to communicate that a subtype may be safely substituted for its supertype. We are not referring to the object-oriented concept of inheritance.

There are situations where the properties described above (or those enforced by specific annotated type systems) do not hold, but the author of the code has investigated the use

and deemed it to be safe. This regularly occurs in the methods which accept a variable of one annotated type, and return that value or a modified product of that value with a different annotated type. For instance, you may take a string and encrypt it, then return an array of bytes with the “Encrypted” type annotation (see Figure 4.1). For these use cases, Sharp Checker provides a mechanism to assert the annotated type. A statement like “`Debug.Assert(true, “variableName:AnnotatedType”);`” will result in the effective annotated type “AnnotatedType” being assigned to “variableName” in the current scope.

```
[return:Encrypted]
public byte[] Encrypt(string text)
{
    byte[] rtn = null;
    using (RijndaelManaged myRijndael = new RijndaelManaged())
    {
        myRijndael.GenerateKey();
        myRijndael.GenerateIV();
        rtn = EncryptStringToBytes(text, myRijndael.Key, myRijndael.IV);
    }
    Debug.Assert(true, "rtn:Encrypted");
    return rtn;
}
```

Figure 4.1: The assertion displayed above imparts the “Encrypted” type annotation to “rtn” and prevents an error resulting from the disagreement with the return attribute

Another small feature of the framework, is a “Not Implemented” warning which is used when we fall through all cases which Sharp Checker is currently able to handle. This indicates to the user that there is a gap in the framework’s implementation, and that they cannot rely on it to enforce the type annotations at that location.

As Sharp Checker was created, test driven development was used whenever possible. Since the framework is meant to be modified and extended, we felt that having a suite of unit tests was essential. This allows for opportunistic refactoring, and confident delivery of new versions of the framework. It also provides a good model for those wishing to build their own type systems. You may start with basic functionality, and write associated tests, then build more advanced features iteratively.

## 4.2 Annotated Type Systems

Sharp Checker was designed to be an extensible framework. The base functionality may be sufficient to enforce certain properties, but often times users will need to extend it. In this section, we will describe the implementation and function of the Encrypted, Tainted, and Nullness type systems, which are distributed with Sharp Checker. It is important to note that with all of these type systems, only the information given can be enforced. If a variable is not annotated properly, and the framework cannot infer the annotated type, then it will not be checked. As you read this section, you may find it helpful to refer to the UML class diagram presented in Appendix A.

The Encrypted type system is quite simple in that it features only one attribute. Values which are not known to be encrypted may not be assigned to those which are marked as “Encrypted” (Figure 4.2 shows a sample error). The Encrypted type system did not need to override any of the analysis provided by the “SCBaseAnalyzer”. It was also unnecessary to override any of the verification taking place in the “SCBaseSyntaxWalker” class. This is reflected in the absences of a “EncryptedSytnaxWalker” class in Figure 4.7. The implementation of the “EncryptedAnalyzer” class, which simply identifies the rule and attribute associated with the analysis, is available in Figure B.1.

On the other hand, values which are properly encrypted may be assigned to a variable with an “Encrypted” annotated type, and a variable with no type annotation will accept an assignment to an encrypted value. We also permit these values to flow as described in Section 4.1 above. In Figure 4.3, we see that the return type of a method is matched with the expected annotated type of an argument.

The Tainted type system is slightly more complex, in that it involves multiple type annotations, and thus a type hierarchy. To accomplish this the “GetAttributesToUseInAnalysis” method was overridden in the “TaintedAnalyzer” class (See Figure B.2). It is here that the type hierarchy is established with “Untainted” as a subtype of “Tainted”.

The Tainted type system is considered somewhat general purpose. It may be used for a host of source-sink problems. For example, you may want to ensure that tainted



```

[Encrypted]
public string EncryptedText { get; set; }
void SetText()
{
    //This causes the diagnostic to fire because the return type of the method
    //doesn't have the appropriate attribute
    EncryptedText = RemoveSpecialChars(plaintext, 3);
}
public string RemoveSpecialChars(string original, in
{

```

Attribute application error Encrypted  
Show potential fixes (Alt+Enter or Ctrl+.)

Figure 4.2: Assigning unencrypted value to variable with “Encrypted” type annotation results in an error

```

    //An [Encrypted] value is returned from the Encrypt method
    //so this may safely be passed to SendOverInternet which
    //expects an [Encrypted] argument
    SendOverInternet(Encrypt(RawText));
}
public string RawText { get; set; }
public int SendOverInternet([Encrypted] byte[] msg)
{
    return Transfer(msg);
}
[return:Encrypted]
public byte[] Encrypt(string text)

```

Figure 4.3: A value with the “Encrypted” type annotation is returned from the Encrypt method, so we can safely pass this as an argument to “SendOverInternet”

values are not passed to methods which execute database queries. This could help protect against SQL injection attacks (see Figure 4.4). You might also consider form values entered on a webpage to be tainted in that they may contain cross-site scripting attacks. Prior to displaying these values on a webpage, you would ensure that they were appropriately cleaned, resulting in a value with the “Untainted” annotated type.

The Nullness type system is more complex than either Encrypted or Tainted. A lot of this complexity arises from the types of properties which the Nullness system must enforce. For example, if you attempt to dereference a variable which may contain a null value, then

```

[Untainted]
private static string GetCustomers = "Select * from dbo.Customers";
static void Main(string[] args)
{
    var dbAccess = new DatabaseAccess();
    dbAccess.ExecuteNonQuery(GetCustomers);
    dbAccess.ExecuteNonQuery(ReadUserInput());
}
[return: Tainted]
public static string ReadUserInput()
{

```

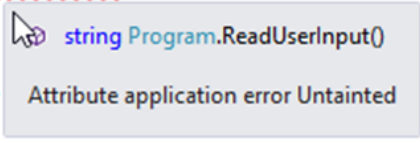


Figure 4.4: Attempting to pass an unsafe “Tainted” value where “Untainted” is expected results in an error

an error should be presented (see Figure 4.6). This means that, unlike many other type systems, the analysis must consider every expression where a variable is dereferenced. At a high level, this is what the Nullness type system is intended to prevent: null reference exceptions.

There are also some challenges associated with accounting for all of the ways in which a variable might be proven to be non-null. One case, which is appropriately handled by Sharp Checker occurs when a variable is explicitly compared to the null literal (see Figure 4.5). We might also consider when a variable has been dereferenced previously. If we get past the first dereference, then we know the variable cannot be null, so long as we are not executing other threads which may unset the value. Another common idiom is comparing parameters to null at the top of a method, and returning or initializing them to a default value if any are null. In Figure 4.6 we clearly see that dereferencing “txt” is safe, but the framework is currently unable to deduce this from the context. These represent more opportunities for type refinement, and make sense to pursue as future work.

```

[MaybeNull]
public string MaybeNullProp { get; set; }
[NotNull]
public string NonNullProp { get; set; }
public string RawText { get; set; }
public void SendOverInternet([NotNull] String msg) { }
void SendText()
{
    //This is ok
    SendOverInternet(NonNullProp);
    // Refine the type of RawText to [NotNull]
    if (RawText != null)
    {
        SendOverInternet(RawText);
    }
    //--Error Cases--//
    SendOverInternet(String.Empty);
    SendOverInternet(MaybeNullProp);
    NonNullProp = null;
    SendOverInternet(null);
    SendOverInternet(RawText);
}

```

string Program.MaybeNullProp { get; set; }

Attribute application error NonNull

Figure 4.5: A variety of use cases for the Nullness type system

```

public int GetSize([MaybeNull] string txt)
{
    if (txt == null)
        return 0;
    return txt.Count();
}

```

Figure 4.6: Currently the framework cannot deduce that “txt” is guaranteed to be non-null

## 4.3 Architecture

In this section, we will describe the implementation of Sharp Checker in terms of the architectural components. Figure 4.7 gives an overview of Sharp Checker, and the relationship to the .NET Compiler Platform. It also highlights the type systems to which Sharp Checker was instantiated in order to demonstrate its usefulness: Nullness, Tainted, and Encrypted. The UML class diagram in Appendix A may be helpful to understanding the material presented in this section.

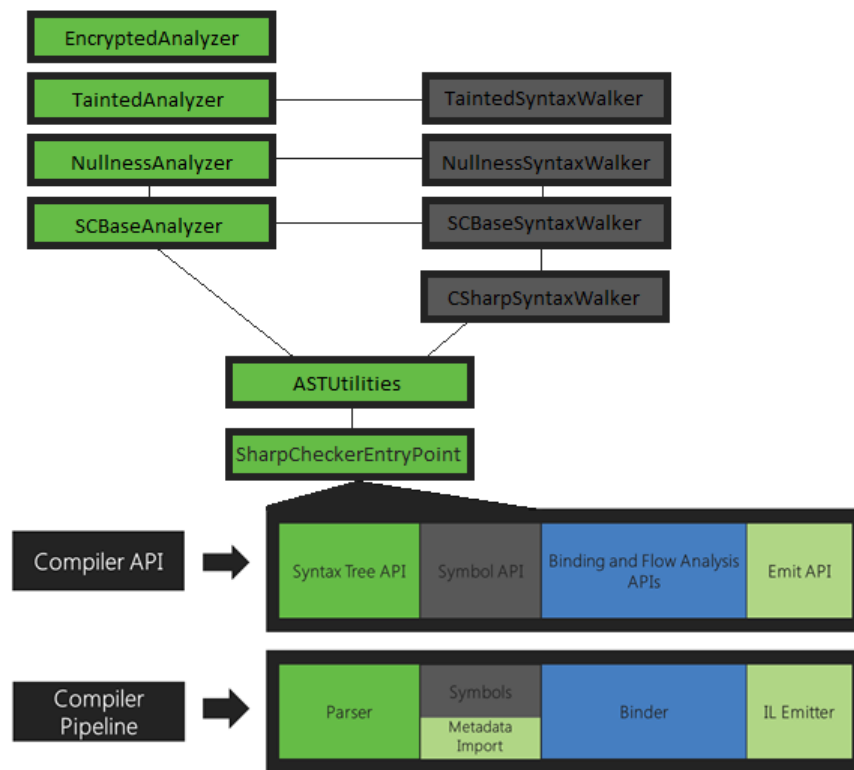


Figure 4.7: Relationship of Sharp Checker to the .NET Compiler Platform

The lopsided pyramid on top of the “Compiler API” sequence gives an indication as to the parts of the compilation with which Sharp Checker is concerned. We plug into the analysis by subscribing to the “CompilationStartAction” within the “SharpCheckerEntryPoint”

class. It is here that we initialize the “ASTUtilities” class which holds our collection of analyzers, and our global symbol table which maps syntax nodes to their type annotations. The target application is searched for an xml file called “checkers.xml”. The analyzers identified in this file will cause the associated “Analyzer” classes to be initialized and applied.

At a high level, the “Analyzer” classes record the attributes associated with various syntax nodes. Often times these are explicit. In other cases, there isn’t sufficient information contained in the syntax. For example, if a method is annotated with a return attribute, the attribute syntax is present at the method declaration, not at the method invocation. When we analyze an assignment where the value being assigned is the result returned from a method invocation, we need to lookup the symbol associated with that method in order to determine the returned attribute. In cases such as this, we are reaching into the Binding API, thus the triangle extends to this stage in the pipeline. The “SyntaxWalker” classes are those which verify the expected type annotations are present, refine types when possible, enforce subtyping relationships, and present diagnostics in the IDE via side effecting method invocations. In Appendix C you will find instructions for how one may add a checker to the Sharp Checker framework. This showcases the essential components of a new pluggable type system, as well as the initial implementation of the Tainted type system.

## 4.4 Challenges

C# does not support attributes on local variables. Ideally, these type annotations would be inferred to ease the annotation burden on the programmer. Nevertheless, at times it would be convenient to have this capability. Currently, we circumvent this limitation with the assertion functionality mentioned in Section 4.1.

Initially, it was hoped that we could hang annotated types on the AST itself. Roslyn exposes a “WithAnnotations” factory method for adding metadata to a copy of an AST. However, we were not able to find a good mechanism for preserving this information. As

previously mentioned, the immutability of the data structures used by Roslyn, and the life-cycle of the analysis, made this approach infeasible.

The Checker Framework makes certain assumptions about the correctness of the program being analyzed with regard to the Java type system. As a result of the fact that they are plugging into the compilation pipeline after type checking has occurred, they may safely assume type correctness. On the other hand, Sharp Checker operates on a programs which are currently being edited, and may not be correct or complete. Roslyn does a good job of predicting what tokens are missing, so this is generally not a problem for our analysis, but it did result in several interesting use cases arising during the course of the project.

Sound compilers provide a guarantee that a program will be judged to be well typed or rejected within a reasonable amount of time. As we build upon the C# compiler we must be cognizant of this guarantee and avoid any logic which might diverge at compile time. Since we are generally operating on a finite AST, it is reasonably straightforward to abide by this constraint. Walking up the parent hierarchy, or down to descendants, should be monotonically increasing or decreasing with no risk of infinite recursion. We also must remain aware of the Roslyn design emphasis on user experience. Performing analysis which degrades this experience would make Sharp Checker unusable in practice.

# Chapter 5

## Analysis

### 5.1 Framework Introspection

One of the primary goals of the Sharp Checker project is to create an extensible framework. Indeed, one of the main reasons why a programmer might choose to use an annotated type system, instead of leveraging traditional types, is that these frameworks tend to be light weight and extensible. Rather than requiring intimate knowledge of the code being verified, and having to change it, one who is well versed in applying a framework such as Sharp Checker might work with a subject matter expert to exercise the framework without invasive changes.

To establish some measure of the extensibility of the framework, we have counted the lines of code in the base classes, and those in classes specific to each annotated type system. As you can see in Figure 5.1, a fair amount of the logic is inherited from these base classes. We believe this indicates that future annotated type systems will also benefit from the base functionality, and that the implementation effort associated with their addition will be proportional to the unique characteristics of that system.

### 5.2 Applying Sharp Checker

To demonstrate the usefulness of the Sharp Checker framework, we applied the annotated type systems described in Section 4.2 to several target applications. In Figure 5.2 we present a summary of the experimental results. Our hope is to give the reader an idea of how well this type of analysis scales, and provide some evidence as to the efficacy of

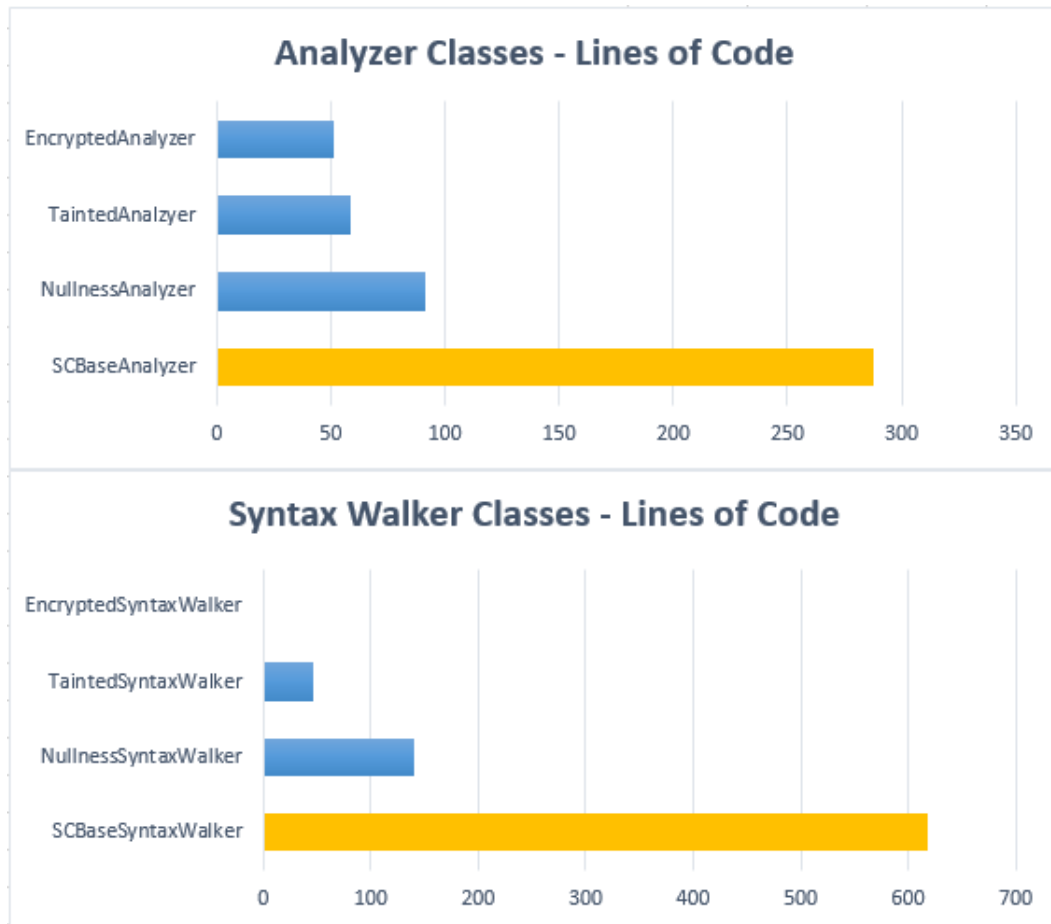


Figure 5.1: The base classes (rendered in gold) of the framework contain a substantial amount of code by comparison to their descendants (blue)

the tool. Unfortunately, it is impossible to establish ground truth in many cases. If we retrieve publicly available code from an online repository as we did with the “EventCloud” application, then we have only the source code itself and the comments contained therein as sources of the author’s intent. We may interpret something to be a bug that is instead an unsupported use case. As a result, quantitative results regarding bugs discovered must be viewed as somewhat imprecise. We can observe only internal consistency with the expectations of the individual annotating the target application, and the ability of the framework to carry out the analysis with respect to their intentions so expressed.



Checker	Target Application	Lines of Code	Annotations	Assertions	Limitations	Bugs Discovered
Encrypted	RijndaelEncryption	100	2	2	0	0
Nullness	SharpChecker	4368	10	12	3	2
Nullness	EventCloud	268144	15	2	1	0
Tainted	EventCloud	268144	10	20	3	0

Figure 5.2: Experimental Results

With regard to target code which we have authored, we have more insight into the intent, but we suffer from the biases of our own style. Applying the analysis to the Sharp Checker source code for example, provided a good mechanism to verify the ability of the framework to enforce our intent as expressed in type annotations. Applying the analysis to code which we did not author, helped us make the framework more robust as a result of the different language features which we encountered in code written by others.

Each of the type systems has supporting unit tests, and a demo application created to exercise the features of the type system. In Figure 5.3 we show some of the use cases from the demo application used to test the Encrypted type system. These demo applications are available along with the Sharp Checker source [7]. Although these are useful to validate basic functionality, to truly exercise the framework we needed to apply it to real world codebases.

We applied the Nullness type system to the Sharp Checker source. As you can see in Figure 5.2, this resulted in the discovery of two bugs. We had reason to believe that there were gaps in the null checks present because a null reference exception had been logged when applying the framework to another target codebase. Through the application of Nullness annotations we were able to find the source of the issue. This was due to an assumption which was made about the possible syntax in the target application. We also discovered several limitations of the framework, and corrected these along the way.

We applied the Nullness and Tainted type systems to the EventCloud application that was retrieved from GitHub [5]. This application was generated using “aspnetboilerplate” which is a template for a web based .NET application [1]. It makes use of many common

```

//--Acceptable Cases--//
//This should be an allowed usage because Ciphertext has the [Encrypted] attribute
//At this call site we need to determine that the method expects an value with an attribute,
//then determine if the value being passed has this attribute (or eventually a subtype attribute).
SendOverInternet(Ciphertext);
//This is ok because the return type of the 'Encrypt' method has the [Encrypted] attribute
SendOverInternet(Encrypt(plaintext));
//This should be allowed because both branches of the conditional return a value with the
bool yep = true;
SendOverInternet(yep ? Encrypt(plaintext + " ending") : Encrypt("testing"));
//--Error Cases--//
//This should generate an error because 'RawText' does not have the [Encrypted] attribute
SendOverInternet(RawText);
//These are also unacceptable
SendOverInternet(this.RemoveSpecialChars(Encrypt(plaintext), 0));
SendOverInternet("");
SendOverInternet(String.Empty);
SendOverInternet(RemoveSpecialChars(Encrypt(plaintext), 1));

```

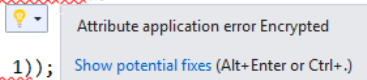


Figure 5.3: Use cases in the demo application used to test the Encrypted type system

components such as an Entity Framework data access layer, and an Angular.js front end. It is featured as one of the trending C# applications on GitHub, which may explain why we found it to be robust with regard to its treatment of null values. However, we did confront several limitations of the Sharp Checker framework. In one case, the result of an await expression is passed as an argument to a method which we attempted to decorate with “NotNull”. Sharp Checker was unable to dig into the await expression to determine if the result of the invocation would have the appropriate annotated type. In another case, we discovered that a central component contained a method that we wanted to annotate. Unfortunately, this component was a third party library, and the source was not available. Without stub functionality, or something similar, we were forced to move the annotations to the calling methods. This resulted in some duplication.

Finally, we located a sample encryption program called “RijndaelEncryption” [6]. This was relatively small, and annotating with the Encrypted type system did not yield any interesting results.

# Chapter 6

## Conclusions

This project was undertaken within the temporal confines of a one semester course. As a result, we narrowed our focus to those features essential for a proof of concept. However, we have made substantial progress, and are able to exercise the framework on real world applications. In the process we found opportunities to improve that code. We have thus provided evidence that Sharp Checker is a viable tool, and that Roslyn provides the appropriate infrastructure for developing these types of tools.

An important question, but one which we did not have the means or time to assess, is whether this type of analysis is worthwhile. Given that there is no ground truth with regard to program correctness, it is difficult to assess the value of these types of analyses. However, given a specific application domain one may readily weigh the initial investment of time against the cost of defects discovered later in the development process, or when the software has been delivered to customers. As with any tool, an initial investment may reap long term benefit if the tool may be applied repeatedly.

### 6.1 Current Status

Sharp Checker delivers the core functionality of a pluggable type system, and was designed with extensibility in mind. It enables the programmer to quickly add type annotations to their target code, and toggle analyses as needed using an xml file included in their project. It also furnishes a mechanism by which they can override the default behavior to issue an error or warning by asserting that a particular use is acceptable.

From the perspective of those who would add their own type systems to the framework, there are several important features. The class structure is such that it is quite straightforward to add to the subtyping and flow constraints enforced by the base classes, or replace this functionality entirely. We have demonstrated context sensitivity or type refinement in the Nullness type system when explicitly checking for null. There are many opportunities to extend this concept to more use cases.

## 6.2 Future Work

We would like to implement a mechanism for inserting type annotations into existing code, annotating the stubbed public interfaces, or performing inter-procedural type inference. These would certainly be worthwhile pursuits, and would benefit from the work accomplished in this project, in that the annotations added may be verified. Indeed, it may be argued that type annotation verification is a prerequisite capability to inference for this reason.

With regard to inserting type annotations, we would like to implement something like the tool for Eclipse called “Cascade” [2]. This would allow for the automatic insertion of attributes where recommended, and would make the process of instrumenting code much easier. This seems like it is in line with the Code Fix functionality which is exposed by Roslyn.

We would also like to instantiate Sharp Checker to categorically different type systems such as Units, Interning, and Lock. This may expose shortcomings of the current design and once addressed, result in a more robust general purpose framework. Expanding upon the warning suppression behavior to allow for ignoring or targeting projects, namespaces, or classes would also ease adoption. With a large legacy codebase, it may be difficult to annotate large swaths of the code and resolve all resulting warnings and errors. Given the ability to target a subset of the code, the task may be broken into more manageable pieces.

It may also be useful to create a mechanism for assigning attributes to syntax elements which do not accept C# attributes. The Checker Framework accomplished this goal by

embedding type annotations in comments, and we believe something similar could be accomplished with Roslyn. This bears some risk because one loses the safety associated with attributes as first class citizens within the C# language. It may also result in performance degradation since comments will have to be parsed manually. This cost would however be paid during development and compilation, rather than runtime, so it may be worth the expressive power it grants.

Lastly, as the Sharp Checker framework grows we may recognize procedures which are repeated, or which appear as one of several variations repeatedly. In these cases, it would be nice to create a declarative mechanism to select the appropriate behavior, as an alternative to the procedural implementation. This is something which the Checker Framework does, and it makes the creation of new type systems seem like a more manageable undertaking.

## 6.3 Lessons Learned

Over the course of this project we learned a great deal about what may be expressed as invariants in code, and the challenges inherent in reasoning about the runtime behavior of a program at compile time. Applying type annotations is nuanced, and requires an investment of time not required by more lightweight analysis, but the results come with guarantees. The analysis is conservative, and there will be false positives. Indeed, there is always room to permit more use cases which a human would judge to be safe. Nevertheless, given an understanding of the features and limitations of the framework, we have seen that implicit understanding may be made explicit, and verified using Sharp Checker. This resulted in the discovery of real bugs, even at this scale and stage of development. We have thus provided evidence that implementing a pluggable type system using the .NET Compiler Platform is viable and worthwhile.

# Bibliography

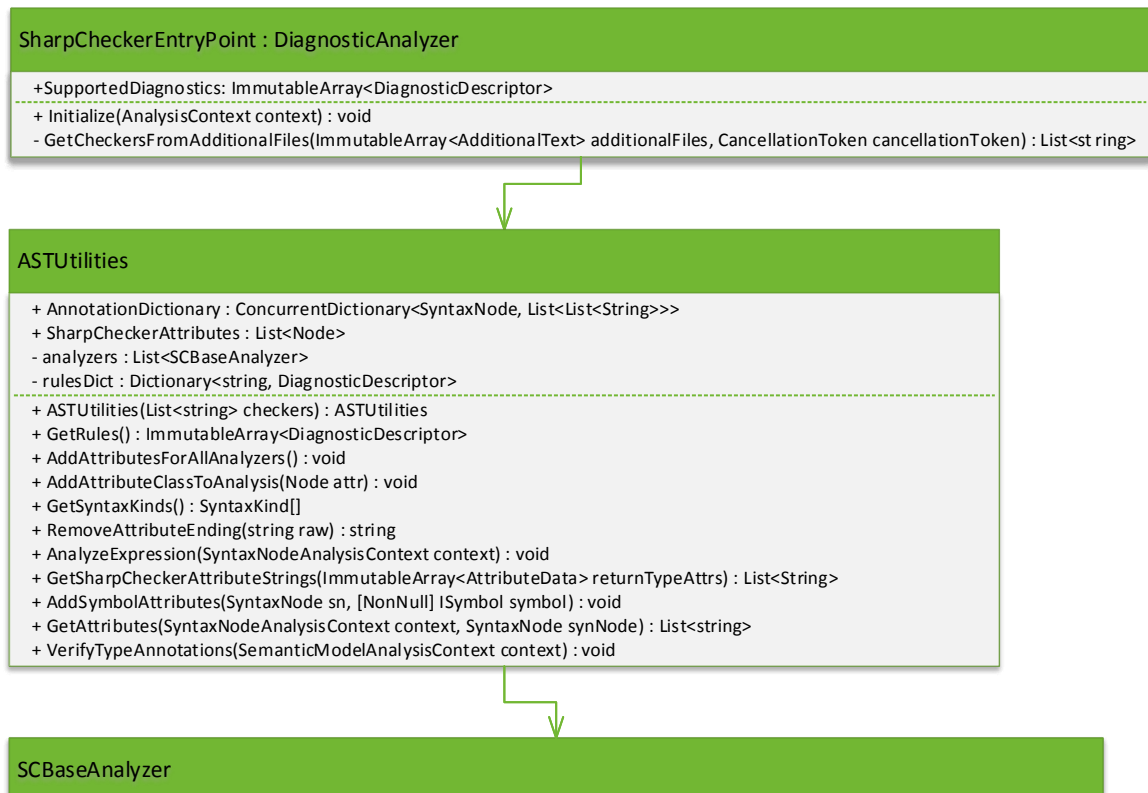
- [1] ASP.NET Boilerplate - Application Framework. <https://github.com/aspnetboilerplate/aspnetboilerplate>. [Online; accessed 5-May-2017].
- [2] Cascade - A Universal Inference Tool for Type Qualifiers in Java 8 (JSR-308). <https://github.com/reprogrammer/cascade>. [Online; accessed 28-Jan-2017].
- [3] JQual User's Guide. [https://www.cs.umd.edu/projects/PL/jqual/users\\_guide](https://www.cs.umd.edu/projects/PL/jqual/users_guide). [Online; accessed 13-Feb-2017].
- [4] Roslyn Analyzer Actions Semantics. <https://github.com/dotnet/roslyn/blob/master/docs/analyzers/Analyzer%20Actions%20Semantics.md>. [Online; accessed 12-Mar-2017].
- [5] Sample SaaS (Multi Tenant) Event Management Application. <https://github.com/aspnetboilerplate/eventcloud>. [Online; accessed 5-May-2017].
- [6] Sanfoundry Technology Education Blog. <http://www.sanfoundry.com/>. [Online; accessed 5-May-2017].
- [7] Sharp Checker Source Code. <https://github.com/tcs1896/SharpChecker>. [Online; accessed 1-May-2017].
- [8] The Checker Framework Live Demo. <http://eisop.uwaterloo.ca/live/>. [Online; accessed 5-May-2017].
- [9] The Checker Framework Manual: Custom pluggable types for Java. <https://checkerframework.org/manual/>. [Online; accessed 11-Feb-2017].
- [10] Jason Bock. *.NET Development Using the Compiler API*. Apress, 2016.

- [11] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu, and Todd W. Schiller. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 681–690, New York, NY, USA, 2011. ACM.
- [12] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28(6):1035–1087, November 2006.
- [13] David Greenfieldboyce and Jeffrey S. Foster. Type qualifier inference for java. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, OOPSLA '07*, pages 321–336, New York, NY, USA, 2007. ACM.
- [14] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis, ISSTA '08*, pages 201–212, New York, NY, USA, 2008. ACM.

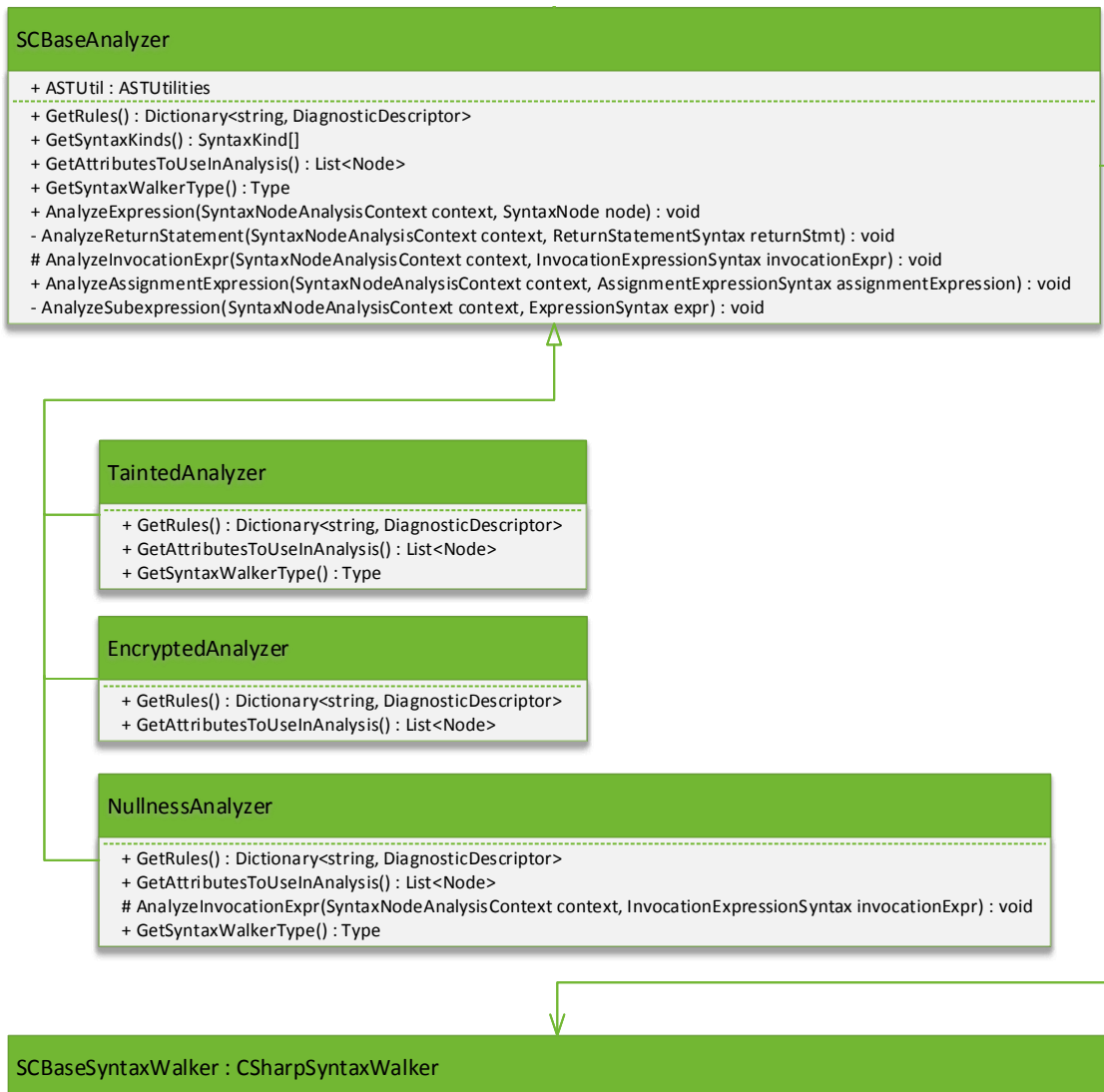
# Appendix A

## UML Diagrams

The following UML class diagram presents the major components of Sharp Checker. The diagram was segmented for presentation here, but the unadulterated version is available with the Sharp Checker source [7]. When only the heading of a class is presented, look to the next page for the complete listing. We have notated access as (+) public, (-) private, or (#) internal.







### SCBaseSyntaxWalker : CSharpSyntaxWalker

```
# rulesDict : Dictionary<string, DiagnosticDescriptor>
# AnnotationDictionary : ConcurrentDictionary<SyntaxNode, List<List<String>>>
# context : SemanticModelAnalysisContext
# attributesOfInterest : List<Node>
+ SCBaseSyntaxWalker(Dictionary<string, DiagnosticDescriptor> rulesDict, ConcurrentDictionary<SyntaxNode, List<List<String>>>
  annotationDictionary, SemanticModelAnalysisContext context, List<Node> attributesOfInterest) : SCBaseSyntaxWalker
+ VisitAssignmentExpression(AssignmentExpressionSyntax node) : void
+ VisitInvocationExpression(InvocationExpressionSyntax node) : void
+ VisitMethodDeclaration(MethodDeclarationSyntax node) : void
+ VisitReturnStatement(ReturnStatementSyntax node) : void
- VerifyReturnStmnt(ReturnStatementSyntax node) : void
# VerifyMethodDecl(MethodDeclarationSyntax methodDecl) : void
# ReportDiagsForEach(Location location, List<string> expectedAttributes, List<string> actualAttributes) : void
- RemoveAllInHierarchy(List<string> expectedAttributes, Node actualNode) : void
# GetSharpCheckerAttributeStrings(ImmutableArray<AttributeData> attrDataCollection) : List<String>
# VerifyAssignmentExpr(AssignmentExpressionSyntax assignmentExpression) : void
# VerifyInvocationExpr(InvocationExpressionSyntax invocationExpr) : void
- RefineTypesBasedOnAssertion(InvocationExpressionSyntax invocationExpr, MemberAccessExpressionSyntax memAccess) : void
# VerifyExpectedAttrsInSyntaxNode([NotNull] List<string> expectedAttributes, [NotNull] SyntaxNode node) : void
# GetDefaultForStringLiteral() : string
# GetDefaultForNullLiteral() : string
```

### TaintedSyntaxWalker

```
+ TaintedSyntaxWalker(Dictionary<string, DiagnosticDescriptor> rulesDict, ConcurrentDictionary<SyntaxNode, List<List<String>>>
  annotationDictionary, SemanticModelAnalysisContext context, List<Node> attributesOfInterest) : TaintedSyntaxWalker
# GetDefaultForStringLiteral() : string
# GetDefaultForNullLiteral() : string
```

### NullnessSyntaxWalker

```
+ NullnessSyntaxWalker(Dictionary<string, DiagnosticDescriptor> rulesDict, ConcurrentDictionary<SyntaxNode, List<List<String>>>
  annotationDictionary, SemanticModelAnalysisContext context, List<Node> attributesOfInterest) : NullnessSyntaxWalker
# VerifyInvocationExpr(InvocationExpressionSyntax invocationExpr) : void
# VerifyExpectedAttrsInSyntaxNode([NotNull] List<string> expectedAttributes, [NotNull] SyntaxNode node) : void
# GetDefaultForStringLiteral() : string
# GetDefaultForNullLiteral() : string
```

# Appendix B

## Code Listing

The entirety of the Sharp Checker source code, sandbox applications, and real-world target applications are available on GitHub [7]. Some samples are included here so that they may be easily referenced.

```
class EncryptedAnalyzer : SCBaseAnalyzer
{
    //Define the diagnostic for the Encrypted type system
    private const string DiagnosticId = "EncryptionChecker";
    private const string Title = "Error in attribute applications";
    private const string MessageFormat = "Attribute application error {0}";
    private const string Description = "There is a mismatch between the effective attribute and the one expected";
    private const string Category = "Syntax";
    private static DiagnosticDescriptor EncryptionRule = new DiagnosticDescriptor(DiagnosticId, Title, MessageFormat,
        Category, DiagnosticSeverity.Error, isEnabledByDefault: true, description: Description);

    /// <summary>
    /// Get the rules associated with this analysis
    /// </summary>
    /// <returns>Encrypted</returns>
    [return: NonNull]
    public override Dictionary<string, DiagnosticDescriptor> GetRules()
    {
        var dict = new Dictionary<string, DiagnosticDescriptor>
        {
            { nameof(EncryptedAttribute).Replace("Attribute", ""), EncryptionRule }
        };
        Debug.Assert(dict != null, "dict:NonNull");
        return dict;
    }

    /// <summary>
    /// This method is called during the init phase of an analysis and should be used to
    /// register attributes for analysis.
    /// </summary>
    /// <returns>Encrypted</returns>
    public override List<Node> GetAttributesToUseInAnalysis()
    {
        return new List<Node>() { new Node() { AttributeName = nameof(EncryptedAttribute) } };
    }
}
```

Figure B.1: The Sharp Checker EncryptedAnalyzer class

In Figure B.1 we present the complete implementation of the EncryptedAnalyzer class.

The behavior is entirely dictated by the base implementation, so here we only override the methods which determine the rules and attributes which we are interested in analyzing.

In Figure B.2 we present a portion of the “TaintedAnalyzer” class. This is similar to “EncryptedAnalyzer”, except for the fact that there are multiple attributes, and we have specified a “TaintedSyntaxWalker” which overrides some of the default verification behavior inherited from the base implementation.

```

/// <summary>
/// Get the rules associated with this analysis
/// </summary>
/// <returns>Tainted and Untainted</returns>
[return: NonNull]
public override Dictionary<string, DiagnosticDescriptor> GetRules()
{
    var dict = new Dictionary<string, DiagnosticDescriptor>
    {
        { nameof(TaintedAttribute).Replace("Attribute", ""), TaintedRule },
        { nameof(UntaintedAttribute).Replace("Attribute", ""), TaintedRule }
    };
    Debug.Assert(dict != null, "dict:NonNull");
    return dict;
}

/// <summary>
/// This method is called during the init phase of an analysis and should be used to
/// register attributes for analysis.
/// </summary>
/// <returns>Tainted and Untainted and the associated hierarchical relationship</returns>
public override List<Node> GetAttributesToUseInAnalysis()
{
    Node tainted = new Node() { AttributeName = nameof(TaintedAttribute).Replace("Attribute", "") };
    return new List<Node>() { tainted, new Node() { AttributeName = nameof(UntaintedAttribute), Supertypes = new List<Node>() { tainted } } };
}

/// <summary>
/// This is the link between the TaintedAnalyzer class and the TaintedSyntaxWalker class which will
/// verify the associated attributes.
/// </summary>
/// <returns>The type TaintedSyntaxWalker</returns>
public override Type GetSyntaxWalkerType()
{
    return typeof(TaintedSyntaxWalker);
}

```

Figure B.2: A portion of the Sharp Checker TaintedAnalyzer class

In Figure B.3 we present the entry point of the Sharp Checker analysis. This demonstrates the mechanism by which we register for compilation actions, and manage the state of the analysis - which is maintained in the ASTUtilities object.

In Figure B.4 we present a sample unit test. These tests were constructed such that the common portions of the sample program are defined globally, and only the segment under test is defined within the test itself. This allows for the creation of many use cases with minimal overhead.

```
[DiagnosticAnalyzer(LanguageNames.CSharp)]
public class SharpCheckerEntryPoint : DiagnosticAnalyzer
{
    /// <summary>
    /// Get our list of diagnostics from the Checkers
    /// </summary>
    public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics { get { return ASTUtilities.GetRules(); } }

    /// <summary>
    /// The entry point of the analysis. This fires once per session, which in a batch processing
    /// mode, corresponds to one compilation.
    /// </summary>
    /// <param name="context">The analysis context</param>
    public override void Initialize(AnalysisContext context)
    {
        context.RegisterCompilationStartAction(compilationContext =>
        {
            //Retrieve the checkers which the target code has identified as active
            List<string> checkers = GetCheckersFromAdditionalFiles(compilationContext.Options.AdditionalFiles, compilationContext.CancellationToken);

            //Perform any setup necessary for our analysis in the constructor
            var analyzer = new ASTUtilities(checkers);

            //Subscribe to be notified when syntax node actions are fired for the types of syntax nodes which we will analyze
            compilationContext.RegisterSyntaxNodeAction<SyntaxKind>(analyzer.AnalyzeExpression, analyzer.GetSyntaxKinds());

            //Register an end action to report diagnostics based on the final state. There is some risk
            //in using this action because it is not gauranteed to fire after all of the syntax node actions.
            //The CompilationEndAction was initially used, which does provide this gaurantee, but does not
            //fire when "full solution anlysis" is not enabled in Visual Studio. This is not enabled by default.
            compilationContext.RegisterSemanticModelAction(analyzer.VerifyTypeAnnotations);
        });
    }
}
```

Figure B.3: The Sharp Checker analysis plugging into the .NET Compiler Platform

```
[TestMethod]
public void OverridingMethodHasNoReturnAttribute()
{
    var overridingClass = @"
    class Graduate : Student
    {
        // The method which is being overridden has the [return:Encrypted] attribute,
        // so a diagnostic is presented
        public override double GetGPA()
        {
            double average = (grades.Sum() / grades.Count()) * 1.05;
            return average;
        }

        public override void AddGrade([Encrypted] double grade)
        {
            double inflation = grade + 1;
            grades.Add(inflation);
        }
    }";
    var test = String.Concat(baseClass, overridingClass);
    var diagLoc = new[] { new DiagnosticResultLocation("Test0.cs", 52, 44) };
    VerifyDiag(test, diagLoc);
}
```

Figure B.4: A sample unit test ensuring that overriding methods have the appropriate return attributes

# Appendix C

## User Manual

### C.1 Use Existing Checkers

In this section we will describe the basics of applying type annotations to a codebase and exercising the Sharp Checker framework.

1. The first step is to add a reference to the Sharp Checker NuGet package which is available on nuget.org. Once installed you will see the associated analyzer appear in the “Analyzers” node within the project.
2. Now that we have access to Sharp Checker, we need to enable the analysis by adding a “checkers.xml” file to the target project. To do so, right click on the project and select “Add/New Item” (see Figure C.1).

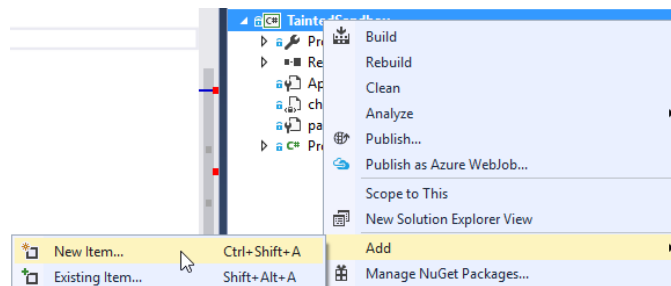
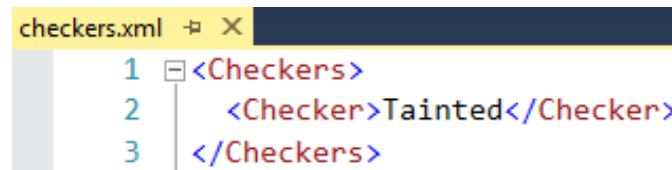


Figure C.1: Add a New Item to the target project

3. Add an xml file called “checkers.xml”.

4. Open the file, and create a parent “<Checkers>” node with subnode “<Checker>”. The content between the opening and closing “Checker” element is the name of the checker. This should match the Analyzer class name without the Analyzer suffix (see Figure C.2).



```

1 <Checkers>
2   <Checker>Tainted</Checker>
3 </Checkers>

```

Figure C.2: Populate the checkers.xml file

5. Now right click on the project and select “Unload Project”.
6. Right click the project again and select “Edit <project>.csproj” (see Figure C.3).

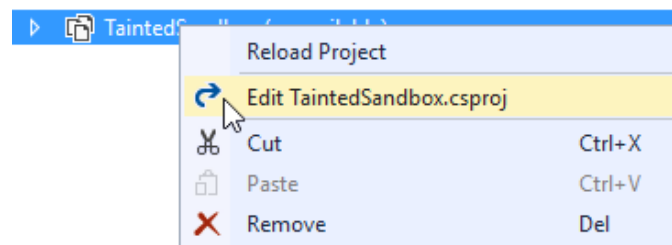


Figure C.3: Edit the proj file

7. Scroll down until you find the “checker.xml” node, and change it from a “Content” node to an “AdditionalFiles” node (see Figure C.4).

```

<ItemGroup>
  <AdditionalFiles Include="checkers.xml" />
</ItemGroup>

```

Figure C.4: Edit the entry to AdditionalFiles

8. Save this change, then right click on the project and select “Reload Project”.
9. Now you are ready to begin applying attributes. Sharp Checker will executed as you type, and also upon building a project. The feedback is presented within the Visual Studio editor, and in the console output.

## C.2 Build Your Own Type System

The following describes the steps necessary to add a new type system to Sharp Checker. This is just a starting point, and you can add, expand upon, or replace functionality as you see fit.

1. Define the attributes with associated hierarchy (see Figure C.5).

```
namespace SharpChecker.Attributes
{
    [SubtypeOf("Tainted")]
    [AttributeUsage(AttributeTargets.All, Inherited = true, AllowMultiple = true)]
    public class UntaintedAttribute : Attribute
    {
    }

    [AttributeUsage(AttributeTargets.All, Inherited = true, AllowMultiple = true)]
    public class TaintedAttribute : Attribute
    {
    }
}
```

Figure C.5: Defining attribute classes

2. Define an Analyzer class which extends “SCBaseAnalyzer”. Within this class declare one or more diagnostic descriptors. Override “GetRules”, “GetAttributesToUseInAnalysis”, and “GetSyntaxWalkerType” (see Figure C.6).
3. Define a “SyntaxWalker” class which extends “SCBaseSyntaxWalker”. Override the constructor, and call the base implementation passing the required parameters. Optionally, you may override methods which are used to retrieve default values at



```

class TaintedAnalyzer : SCBaseAnalyzer
{
    //Define the diagnostic for the Tainted type system
    private const string DiagnosticId = "TaintedChecker";
    private const string Title = "Error in attribute applications";
    private const string MessageFormat = "Attribute application error {0}";
    private const string Description = "There is a mismatch between the effective attribute and the one expected";
    private const string Category = "Syntax";
    private static DiagnosticDescriptor TaintedRule = new DiagnosticDescriptor(DiagnosticId, Title, MessageFormat, Category,
                                                                              DiagnosticSeverity.Error, isEnabledByDefault: true, description: Description);

    /// <summary>
    /// Get the rules associated with this analysis
    /// </summary>
    /// <returns>Tainted and Untainted</returns>
    [return: NonNull]
    public override Dictionary<string, DiagnosticDescriptor> GetRules()
    {
        var dict = new Dictionary<string, DiagnosticDescriptor>
        {
            { nameof(TaintedAttribute).Replace("Attribute", ""), TaintedRule },
            { nameof(UntaintedAttribute).Replace("Attribute", ""), TaintedRule }
        };
        Debug.Assert(dict != null, "dict:NotNull");
        return dict;
    }

    /// <summary>
    /// This method is called during the init phase of an analysis and should be used to
    /// register attributes for analysis.
    /// </summary>
    /// <returns>Tainted and Untainted and the associated hierarchical relationship</returns>
    public override List<Node> GetAttributesToUseInAnalysis()
    {
        Node tainted = new Node() { AttributeName = nameof(TaintedAttribute).Replace("Attribute", "") };
        return new List<Node>() { tainted, new Node() { AttributeName = nameof(UntaintedAttribute), Supertypes = new List<Node>() { tainted } } };
    }

    /// <summary>
    /// This is the link between the TaintedAnalyzer class and the TaintedSyntaxWalker class which will
    /// verify the associated attributes.
    /// </summary>
    /// <returns>The type TaintedSyntaxWalker</returns>
    public override Type GetSyntaxWalkerType()
    {
        return typeof(TaintedSyntaxWalker);
    }
}

```

Figure C.6: Defining the analyzer class

the appropriate time in the flow analysis. The “SCBaseSyntaxWalker” extends the “CSharpSyntaxWalker” class, which implements the visitor pattern. There are methods that may be overridden for each syntax element which may be visited in the AST. One traversal is performed per analysis (see Figure C.7).

4. Add the analyzer class to the array defined in “ASTUtilities.GetRules()” (see Figure C.8).
5. Now execute the VSIX project distributed with the Sharp Checker source. This will launch an experimental hive instance of Visual Studio. Load your target application, or create a new test project.

```

class TaintedSyntaxWalker : SCBaseSyntaxWalker
{
    /// <summary>
    /// Pass the arguments along to the SCBaseSyntaxWalker constructor
    /// </summary>
    /// <param name="rulesDict">A dictionary which maps strings used as attributes to their associated rules</param>
    /// <param name="annotationDictionary">The global symbol table which maps syntax nodes to the associated attributes</param>
    /// <param name="context">The analysis context which Roslyn provides</param>
    /// <param name="attributesOfInterest">The attributes which have been registered for analysis</param>
    public TaintedSyntaxWalker(Dictionary<string, DiagnosticDescriptor> rulesDict, ConcurrentDictionary<SyntaxNode, List<List<String>>> annotationDictionary,
        SemanticModelAnalysisContext context, List<Node> attributesOfInterest) :
        base(rulesDict, annotationDictionary, context, attributesOfInterest)
    { }

    /// <summary>
    /// Get the default attribute which should be applied to string literal expressions
    /// </summary>
    /// <returns>Tainted</returns>
    internal override string GetDefaultForStringLiteral()
    {
        return nameof(TaintedAttribute).Replace("Attribute", "");
    }

    /// <summary>
    /// Get the default attribute which should be applied to null literal expressions
    /// </summary>
    /// <returns>Tainted</returns>
    internal override string GetDefaultForNullLiteral()
    {
        return nameof(TaintedAttribute).Replace("Attribute", "");
    }
}

```

Figure C.7: Defining the syntax walker class

6. Follow the steps described above in Section C.1 to add the checkers.xml file to activate your new type system.
7. You will inherit the flow and subtyping functionality of the Sharp Checker framework. From here you may build additional functionality or override it as needed.

## C.3 Extending Sharp Checker

In this section we will provide detail which may be disregarded unless you wish to extend the Sharp Checker framework. We only present those few items which were not included elsewhere in this document, and which we feel may be helpful to those wishing to extend the framework.

First, we would like to note our observations about the way in which Sharp Checker interfaces with Roslyn. Only certain actions are guaranteed to execute in a particular order [4]. For example, the “CompilationStartAction” will execute first, and the “CompilationEndAction” will execute last. However, we discovered that this does not mean that

```

/// <summary>
/// Returns a collection of diagnostic descriptors to the DiagnosticAnalyzer which drives the analysis.
/// Type system creators should add to the list of analyzers below.
/// </summary>
/// <returns>The rules that we will enforce</returns>
public static ImmutableArray<DiagnosticDescriptor> GetRules()
{
    List<DiagnosticDescriptor> descriptors = new List<DiagnosticDescriptor>();
    //If you add a new analyzer class, make sure to add it to this list. We can't discover these classes using the
    //AdditionalFiles mechanism because we don't have the compilation context when assigning the SupportedDiagnostics
    //property of our DiagnosticAnalyzer instance
    var analyzerClasses = new SCBaseAnalyzer[]
    {
        new SCBaseAnalyzer(),
        new EncryptedAnalyzer(),
        new NullnessAnalyzer(),
        new TaintedAnalyzer()
    };
};

```

Figure C.8: Adding the analyzer to the GetRules method

“CompilationEndAction” will execute. In fact, it is only executed when “full solution analysis” is enabled within Visual Studio. Since this is a setting in the user’s IDE, there is no way to enable this setting upon adding the Sharp Checker NuGet package. Rather than force Sharp Checker users to take an additional setup step, we used the “SemanticModelAction”. This executes once the semantic model has been constructed. It stands to reason that this could only happen after the syntax nodes have been processed, so enforcing type annotations at this time seems to be an acceptable solution. There very well may be other ways to integrate with Roslyn. In particular the code block actions show promise.

You may notice that a Node struct was created to allow for multiple supertypes of a type annotation. Alternatively, we could have used C# inheritance to establish subtyping among type annotations. However, given that only single inheritance is supported by C#, this would have been somewhat limiting.