Gradual Pluggable Typing in Java

by

Daniel Brotherston

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathemathics
in
Computer Science

Waterloo, Ontario, Canada, 2016

© Daniel Brotherston 2016

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Gradual typing provides the ability to safely mix untyped or dynamically typed code with statically typed code while maintaining, within the statically typed portion, the guarantees claimed by the static typing. It is motivated by the idea that different amounts of typing are valuable at different times during a software projects development. Less typing is useful at the prototyping stage, where requirements change frequently, and rapid development is important. More typing is useful in more mature projects where maintenance, documentation, and correctness become critically important. Gradual aims to enable a project to gradually move from one side of this spectrum to the other.

Pluggable typing and pluggable typing frameworks aim to allow a language to support many different type systems, that can be developed by type system designers and plugged into the existing compiler infrastructure. The Checker Framework is built on the OpenJDK™ Java compiler, and provides a framework to develop additional type systems that can be applied to Java code. It has been tested and proven using many other type systems. However, not all code will adopt all type systems, so a developer who wishes to check their project with a given type system may still link against other components that are not checked with that type system. Additionally, a developer may wish to adopt a new type system, but be unwilling to spend the effort necessary to annotate all the existing code for the new type system.

The gradual extension to the Checker Framework aims to improve type safety of partially applied Checker Framework type systems, using gradual typing techniques. A prototype was built, and the gradual nullness type system was designed to demonstrate it.

Defining the checked-unchecked boundary is crucial. Even though the use case is limited to individual files being either entirely annotated or entirely unannotated, because Java is an object-oriented language with inheritance, many different boundary conditions present themselves. This work analyses each boundary condition in detail, and presents options for runtime tests to ensure that values crossing the boundary meet the static type requirements on the other side. The nature of the tests, and the method of passing the static type to the runtime environment is discussed.

This work presents the design of the prototype, implemented in the Checker Framework and OpenJDK™. It discusses the design of the OpenJDK™ compiler, the Checker Framework, and how they work together. It demonstrates how the abstract syntax tree can be modified in order to effect insertion of the runtime tests. It discusses the specific implementation details needed to implement each boundary condition runtime test in Java.

An evaluation of the prototype is completed looking at correctness, both through a set of synthetic tests, and by inserting artificial errors into an existing real world program; performance, by running configurations of real world programs designed to model the motivating examples; and applicability, by comparing the gradual framework with other options for improving type safety in partially annotated programs.

Related work on both gradual typing and pluggable typing are discussed. Final remarks and future work concludes the work.

Acknowledgements

This work was supported by the University of Waterloo, by NSERC through the Alexander Graham Bell Canada Graduate Scholarship, and by the David R. Cheriton Graduate Scholarship. I would like to express my most sincere thanks to my Supervisors, Ondrej Lhotak and Werner Dietl, for guiding me through this research, and to my readers, Gregor Richards and Peter Buhr for their time and helpful feedback. Also, to my lab mates, who put up with long and detailed conversations that were always enlightening to me. And especially, my partner Tessa, who has supported me unwaveringly through this experience.

Dedication

I dedicate this work to my parents, who always supported and encouraged my interest in computer science, even when they had no idea what I was talking about.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements v

Dedication vi

List of Tables x

List of Figures xi

1 Introduction 1

1.1 Gradual Typing 1

1.2 Pluggable Typing 3

1.2.1 Checker Framework 4

1.2.2 Unchecked Code in the Checker Framework 5

1.2.3 The Nullness Type System 8

1.3 Gradual Pluggable Typing 8

1.3.1 Target Use Cases 9

1.4 Contributions 10

2 Solution Description **11**

2.1 Type Hierarchy and the `@Dynamic` Qualifier 11

2.2 Checked-Unchecked Boundary 12

2.2.1 Runtime Tests 13

2.3 Defining the Boundary 16

2.3.1 Return Values and Parameters 17

2.3.2 Fields 19

2.3.3 Inheritance 23

3 Implementation **36**

3.1 OpenJDK™ Java Compiler 36

3.2 Checker Framework 38

3.3 Runtime Transformations 39

3.3.1 Runtime Checks 40

3.4 Constructors 42

3.4.1 Checking Parameters 45

3.5 Virtual Dispatch Runtime Check 46

4 Evaluation **48**

4.1 Correctness 48

4.1.1 Setup 49

4.1.2 Results 49

4.2 Performance 50

4.2.1 Setup 51

4.2.2 Results 51

4.3 Usability 52

5 Related Work **54**
5.1 Pluggable Typing . . . 54
5.1.1 Checker Framework . . . 55
5.1.2 Checker Framework Type Systems . . . 56
5.2 Formative Work in Gradual Typing . . . 57
5.3 Gradual Type Systems . . . 59

6 Conclusion **62**
6.1 Future Work . . . 63

References **65**

List of Tables

4.1 Compilation Times (in seconds) 51
4.2 Runtime (in seconds) 52

List of Figures

1.1 Java Annotations in Variable and Method Declarations. 4

2.1 Type Hierarchy With the Dynamic Type 12

2.2 Runtime Check Method (part of the gradual checking framework) 18

2.3 Unchecked Method: A.Java 18

2.4 Calling Unchecked Method: B.Java 18

2.5 Safe Calling Unchecked: B.Java 19

2.6 Unchecked Return Value 20

2.7 Checked Method: A.Java 21

2.8 Calling Checked Method From Unchecked Code: B.Java 21

2.9 Method Transformation to Achieve Safe Calling: A.Java 22

2.10 Unchecked Argument Value 28

2.11 Unchecked Portion: A.Java 29

2.12 Checked Portion: B.Java 29

2.13 Checked Portion After Transformation: B.Java 30

2.14 Checked Portion: A.Java 30

2.15 Unchecked Portion: B.Java 31

2.16 Checked Portion after Simple Transformation: A.Java 31

2.17 Checked Portion after Full Transformation: A.Java 32

2.18 Inheriting from Unchecked 33

2.19 Checked Portion: A.Java 34

2.20 Unchecked Portion: B.Java 34
2.21 Checked Portion: A.Java 35
3.1 Phases of the OpenJDK™Java Compiler. Grey boxes indicate compilation phases during which the gradual checking prototype modifies the Java AST. 37
3.2 The Interface For Runtime Test and Error Functions 41
3.3 Checked Portion: A.Java 42
3.4 Unchecked Portion: B.Java 42
3.5 Code Added to the Gradual Checking Framework 44
3.6 Checked Portion A.Java 44
3.7 Helper Method for Checking Arguments To a Method Call 46
3.8 Using the Method Call Helper to Runtime Check a Method Call 46
3.9 Virtual method Dispatch Runtime Check 47

Chapter 1

Introduction

By combining the disciplines of gradual typing with pluggable typing, this thesis seeks to introduce a gradual, pluggable, type system framework. This framework allows gradual type systems to be implemented easily. Gradual typing further reduces the burden of adding a pluggable type system to an existing project, as well as increases the type safety provided by existing pluggable type systems. It can also make errors easier to find as they are caught at the point of interaction between checked and unchecked code, instead of propagating further into the program.

This section introduces the project. Section 1.1 provides background on gradual typing, section 1.2 discusses pluggable typing. Section 1.2.1 introduces the checker framework and discusses type annotations in section 1.2.1, and how it handles unchecked code in section 1.2.2. Section 1.3 introduces gradual pluggable typing along with several motivating examples and target use cases. Section 1.4 concludes the chapter with the main contribution, in addition to the organization of the remainder of this thesis.

1.1 Gradual Typing

Traditionally, gradual typing is the ability to combine static and dynamic typing in a single program while maintaining type safety. This is to say, some parts of the program may contain type annotations and be statically typed, and are checked for correctness by the compiler. Other parts of the program do not have static types assigned at compile time, but is instead typed dynamically, and checked for correctness at runtime by the runtime system.

Type checking some expressions at compile time, and some expressions at runtime is desirable in order to achieve a balance between the complementary strengths of dynamic and static checking. Dynamic typing provides a flexibility to the program which enables very fast prototyping and rapid development cycles. This is ideal for some stages of the development cycle, but as a project becomes larger and more mature, the safety and performance provided by static typing becomes more attractive. Traditionally, languages supported either static or dynamic typing, and switching between the two would require either complex integration of several languages or rewriting of parts of the program. Gradual typing aims to allow a programmer to seamlessly increase or reduce the amount of static typing (through explicit type annotations) in a program simply by adding or removing type annotations. Gradual typing achieves this by adding runtime type checks at the boundary between the typed and untyped portions of the program. These type checks ensure that at runtime, the types that were used at compile time in the typed portion hold true, thus ensuring type safety within the typed portion.

Gradual typing was formalized by Siek and Taha by introducing a calculus named $\lambda^{?}_{\rightarrow}$ and proving the following properties, which they considered important for gradual typing [20].

- When all functions in the program have types fully applied, that is, the program is fully annotated, the type system is equivalent to the simply-typed lambda-calculus, and as such all type errors are caught at compile time.
- Portions of the program that are assigned static types at compile time do not incur the performance penalty of runtime checks or value boxing.
- Type safety is maintained at the boundary between static parts and dynamic parts of the program through the use of boxed values and runtime checks.

The type system presented by Siek and Taha implements gradual typing by adding an "unknown type" to the simply-typed lambda-calculus, denoted as '?'. This type represents a dynamic type which is determined at runtime. Siek and Taha defined a type consistency relation between types. It handles the case of complex types where different parts might be known or unknown. For example, a function type $? \rightarrow t$ can be considered to be consistent with another type $t \rightarrow ?$. The consistency relation is equivalent to the mathematical construct of partial function consistency where the type ? is considered to be an undefined portion of the partial function. As a result, the consistency relation is reflexive and symmetric but not transitive, which is necessary for type soundness.

Siek and Taha then provide a translation from this enhanced calculus to the simply-typed lambda-calculus with typecasts. They also provide runtime semantics for the simply-typed lambda-calculus typecasts.

Siek and Taha later expanded this gradual type system formalization to an object calculus [19]. They expanded their definition of consistency to include subtype relations. This more complex consistency relation can be thought of simply as a right angle triangle. Two types, A and B are subtype consistent, $\mathtt{A}\ \widetilde{<:}\ \mathtt{B}$, if there exists a third type, C which is either a subtype of A and consistent with B, that is, $\mathtt{A}\ <:\ \mathtt{C}\ \&\&\ \mathtt{C}\ \sim\ \mathtt{B}$ or is a super type of B and consistent with A, that is, $\mathtt{A}\ \sim\ \mathtt{C}\ \&\&\ \mathtt{C}\ <:\ \mathtt{B}$. In both cases, the relationship can be seen to form a right angle triangle in the field where one dimension represents consistency, and the other dimension represents the subtype relationship. In both cases, the type C must vary from A and B in only a single dimension.

1.2 Pluggable Typing

One goal of type systems is to provide confidence that code is free from certain types of errors. Most languages provide a fairly standard set of features in their type system. Typically but not always this includes protection from calling methods that do not exist, calling constructs that are not callable, or calling with arguments that do not meet the requirements of the parameters they are passed too. While type systems are often flexible enough to allow some correctness guarantees to be modeled with types to allow the compiler to provide correctness guarantees, there are limitations on what can be checked in this manner.

Pluggable type systems on the other hand seek to provide a framework for which additional type systems can be designed and 'plugged-in' to the existing compiler infrastructure to allow different classes of errors to be checked for. In effect these additional type systems further restrict the set of programs that will be accepted by the compiler. This framework allows researchers to easily experiment with new forms of type checking, and allows real world software developers to selectively turn on type checking for different classes of errors across specific regions of code. In the context of the Checker Framework [5], there are two groups of end users: type system designers build a type system using the Checker Framework and end developers who use the new type system within their own code.

```
@Annotation int foo = 2;
void method(@Annotation List<@Annotation Integer> param);
```

Figure 1.1: Java Annotations in Variable and Method Declarations.

1.2.1 Checker Framework

The Checker Framework is a concrete implementation of a pluggable type system built for the Java programming environment using the OpenJDK™ compiler. It is designed to be an easy to use extensible system allowing both developers and researchers to easily build and test type systems using standard Java code that can be compiled with any standard Java compiler.

Annotations

The Checker Framework uses a language feature called Java annotations, which provides an extensible syntax to allow developers to add annotations within a program. These annotations can be processed by an annotation processor that executes as a phase of the compilation process. A developer of Java annotation develops a system of permitted annotations, and an associated annotation processor. An end user then writes annotations within their code, and use the associated annotation processor to process the annotations for whatever purpose they are intended. Annotations provide Java with an extensible syntax, where custom language features can be implemented as annotations and annotation processors. Annotations have been used in several ways in the Java language from providing interface deprecation and versioning information, to implementing automatic property and data class specifications.

Figure 1.1 illustrates adding an annotation named `Annotation` and denoted as `@Annotation` in Java code, to several different type-use expressions in Java.

The Checker Framework uses annotations in two ways. First, type system designers can define a number of annotations that end developers can use in their code to specify types. These annotations are referred to as qualifiers.

Second, the Checker Framework provides a number of built in meta-annotations to allow type system designers to specify their type system. For example, there are subtype relationship annotations that type system designers use to specify subtype relationships

between their type qualifier annotations. This allows a declarative style for defining parts of the type system.

The actual Checker Framework type checker is an annotation processor library, which allows each type system designer to easily define a customized type checking annotation processor for their type qualifier annotations. The language designer only needs to implement the parts of their type system that differ from the default provided by the Checker Framework. For example, they must provide a list of supported type qualifiers, and can customize subtyping relationships, default types, etc. The Checker Framework processor libraries provide many useful features to provide common reusable type system behaviour. These include: type hierarchy systems, with subtype relationship tests; a type checking algorithm, which checks Java constructs for type correctness and requires only a simple type assignment test to be implemented. Developing the framework using the annotation syntax provided by Java provides many benefits:

- Compatibility with standard Java is maintained, since the compiler simply ignores (or stores) annotations which are not being processed.
- Type systems are easy to build, because a framework for checking already exists; a developer must only define a set of qualifiers, and a system for checking specific special cases that their type system depends on.
- Finally, type systems are easy for developers to use, as developers must only add annotations in their code to specify additional type information, and then add an annotation processor phase to their build process.

The Checker Framework has been tested extensively by Dietl et al. [5][9][16]. This extensive testing has shown the framework to be useful both in type system research and for finding actual errors in real-world code. The researchers were able to quickly develop several novel type systems for Java, and then apply them to real programs to find previously unknown errors. Sampson et al. were also able to use the system in developing a research type system for experiments using static typing to ensure correctness in hybrid low power systems with high error-rates [17]. Their project included modification of the Java Abstract Syntax Tree (AST) at compile time.

1.2.2 Unchecked Code in the Checker Framework

The end developer has a program consisting of their source code, as well as various libraries for which source code maybe not be provided. Parts of this source code and parts

of the libraries may be annotated with types from the Checker Framework. Any part of either the end developer's source code, or any of the additional libraries, may be checked or not checked by the Checker Framework. This flexibility is possible because the Checker Framework functions as phase of the OpenJDK™ Java compiler, and produces compatible class files, so an end developer may compile their source code and use the Checker Framework to check it, but still link against class files that were not checked using the Checker Framework. Section 3.1 describes the phases of the OpenJDK™ Java compiler in more detail.

Currently, the Checker Framework applies default types to symbols which do not have any type annotations. Symbols that lack a type annotation can originate from two places. From source code, where the developer has not written a type annotation, and from bytecode, where the Checker Framework never processed the original source code. Defaults are specific to the particular type system, but in general, type systems apply a defaulting policy that aims to reduce the number of type annotations required in order to reduce the burden for users adopting the type system. In some systems, this involves selecting the bottom type as default. However, for symbols only in bytecode, where the checker framework does not see the code, and therefore cannot verify that the bottom type is safe, the top type can be chosen as a default. This approach is a conservative default in that it should ensure some type safety, but may require additional work on the part of the developer.

In many cases this defaulting mechanism is effective; the nullness checker for example, defaults all types in unannotated but checked code to `@NonNull`, and it defaults unannotated and unchecked code to `@Nullable`. This serves to minimize the number of type annotations required in Java as in practice few references end up containing null values. It still maintains some type safety however, as the conservative nature of unchecked defaults ensures that the type system accommodates code that could produce a null value. When combined with dataflow-sensitive type refinement, this leads to a low annotation overhead [5].

The Checker Framework also provides a framework for advanced dataflow-sensitive analysis for type refinement that allows the type system designer to restrict the type depending on the code structure. In the case of nullness checking, when a reference value is checked against null, the type system can now safely assume the value is non-null. However, this only applies to code which the Checker Framework actually processes.

For code checked by the Framework, a pluggable type is assigned to every value or expression in an end developer's code, and the Checker Framework annotation processor verifies that the type is valid. That is, the type checking algorithm is applied to the AST

to verify that the type determined by the annotations, by defaults, or by flow analysis is correct.

The second category of code that the Checker Framework must deal with is code where the Checker Framework only sees the bytecode and the original source code is never processed by the Checker Framework. This scenario could result from using a proprietary library, where only a compiled version is available, or from using a different compiler for a different part of the project.

In this case, types are still applied to the methods and fields of each class or interface. These types may be derived from defaults (possibly specifically for unchecked code, as described above), or, developers, either end developers, or type system designers may provide a stub file to specify the Checker Framework types that should apply to various methods. For this, a developer must read through the documentation for the source code and determine which type is appropriate for each method.

Since the Checker Framework never sees the actual implementing source code, no type checking algorithm is run on it. Thus, it is never verified that the default, or the developer specified type is correct, or that the method is free from errors, which may violate the type. This introduces an element of weakness in the type system. These values from unchecked code may permeate through the typed portion of the program meaning any value or expression may actually at runtime contain a value, which violates its type.

Using conservative defaults in unchecked code can reduce this weakness. This ensures that in all cases, untyped methods and fields return or contain values that are valid given the type. However, it complicates integrating the Checker Framework, as it increases the overhead in terms of the number of value checks (i.e., null pointer checks), or type annotations a developer is required to write. In some cases, these may in fact be useful in catching bugs, but in many cases are simply extra overhead for the developer. Additionally, since stub file specifications for library code override default types, conservative defaults for unchecked code do not entirely eliminate the risk of type errors.

A more flexible option is to use the techniques from gradual typing to insert runtime checks at the boundaries between the statically checked portions of the code and unchecked portions. In this way, end developers could slowly add a pluggable type system to their code without being forced to add annotations or value checks at every location where checked code interacts with unchecked code, and yet still maintain type safety within the checked portion of the code.

1.2.3 The Nullness Type System

The Checker Framework comes with several type systems available out of the box. The Nullness type system is one of the oldest and most mature available. It provides a practical and useful Nullness type system within Java. It includes a KeyFor type system to statically determine when a Java Collections Framework map object is guaranteed to contain a value for a given key, thus reducing the need for superfluous nullness checks on calls to retrieve values from map objects that can be statically shown to be not null.

The type hierarchy in the Nullness type system includes the top type `@Nullable`, the bottom type `@NonNull`, and a polymorphic type `@PolyNull`. `@PolyNull` allows polymorphism in the type system so, for example, a method can accept a `@PolyNull` parameter, and the return value depends on the type of the argument passed to the method.

The Nullness type system secures the system against `NullPointerExceptions` (NPEs). It ensures that derefences only occur on `@NonNull` values, and it ensures that assignments to `@NonNull` values are always `@NonNull`. In this way, it conservatively prevents NPEs from occurring. However, the other weaknesses described in pluggable typing mean NPEs are still possible.

The Nullness type system uses data-flow sensitive type refinement to allow `@Nullable` values to flow into `@NonNull` values after they have been checked for nullness. Papi et al. found that the NonNull Except Locals (NNEL) defaulting policy, when combined with data-flow sensitive type refinement, resulted in a reduced annotation burden for end users [16]. This policy defaults all program values, except local variables, to `@NonNull`. Local variables default to `@Nullable` but are often either assigned to or checked for nullness within a local block.

1.3 Gradual Pluggable Typing

Different type systems in the Checker Framework may prevent different types of runtime errors. In the example of a nullness checker, a null value used where a non-null value is expected may cause a NPE. This error is something that is caught at runtime by the Java Virtual Machine (JVM), but may occur at a point in the program that the developer does not expect. Using a gradual type system, this error is detected at the point the object crosses the static/dynamic checking boundary, which can help the developer localize the issue. Additionally, because the gradually typed nullness type system provides guarantees to the runtime that a value cannot be null, a customized runtime system could optimize this code by removing the nullness checks.

Some type systems, however, do not protect from errors that readily result in runtime exceptions. Instead they may protect from errors which cause incorrect results, or even corruption of the runtime system. For example, Graphical User Interface (GUI) frameworks often require that certain methods are only executed on a specific thread, often referred to as the GUI thread. However, these frameworks, in order to remain responsive, generally require that heavyweight processing be conducted on a different thread, called a worker thread. If a worker thread invokes a GUI method directly, some frameworks will immediately throw an error. However, others may not check for this error and instead it may result in corruption of the runtime state of the program, which may cause errors later on, or may simply result in incorrect rendering of the GUI. This error is very difficult to debug since it can occur very far from where it originated. A type system was developed to check for this, but since it isn't entirely safe, it may still result in errors. Using gradual typing to insert checks into the checked-unchecked boundary, the error may be caught earlier and a difficult to track down transient problem can be turned into a runtime type error, at a specific boundary site.

1.3.1 Target Use Cases

A developer wishing to use a pluggable type system in their project maintains type safety while calling out to third party libraries that do not have source code available. In this case, using gradual pluggable typing would insert checks at the boundary between the library and the code, and thus verify that neither the user's code, nor the library, violates any type systems that are only applied to one or the other.

If a developer wishes to add a new pluggable type system to their existing project, they must add some number of type annotations to their existing project in order to satisfy the new type system [5]. This number may be low, if the type system has reasonable defaults and flow analysis, but it is not always the case. This initial effort may discourage developers from adopting an additional type system in their code, especially if they are unsure of the benefit.

By providing a gradual typing system, the developer could gradually add types to their program in order to statically check the parts of the program they wish to validate, while still benefiting from the dynamic checks added to verify behaviour at runtime. This helps the developer isolate the cause of problems in their code and provide indications of where to add more annotations.

1.4 Contributions

This thesis contributes a specification of a prototype for a gradual, pluggable, type system framework built upon the existing Checker Framework pluggable type system framework. The thesis is organized as follows:

- Chapter 2 explores and highlights the theoretical challenges to implementing a gradual type system in a pluggable type system context, on top of a real world object oriented language like Java. Specifically, the Chapter investigates the nature of the checked-unchecked boundary in the context of an object oriented language with inheritance and interfaces.
- Chapter 3 explores the specific implementation details required to build a gradual checking system on top of the Checker Framework and the OpenJDK™ Java compiler. It includes details of the compiler, the annotation processing phase, and on modifying the AST within the Java compiler during the annotation processing phase, what runtime tests actually look like, and how to handle constructors.
- Chapter 4 provides an evaluation of the framework, and the sample gradual nullness type system by evaluating the performance of both the compiler and the compiled code using an existing project annotated for the Checker Framework nullness type system.
- Chapter 5 discusses previous work on pluggable type systems, work on the formalized foundations of gradual typing, and on other examples of gradual typing systems.
- Chapter 6 discusses possible future work, and summarizes the contributions of the thesis.

Chapter 2

Solution Description

This chapter discusses the nature of the boundary between the checked and unchecked portions of the code, how it arises, and the possible type errors and solutions to detect the type error, that are present along the boundary. First, section 2.1 describes the modification to the type hierarchy required to allow dynamic as a type, as well as the `@Dynamic` annotation and type qualifier. Section 2.2 describes the general nature of the boundary, and how it can arise in the development of software using a Checker Framework type system. Section 2.2.1 describes the design of the runtime checks that will be added to the final program. Section 2.3 describes in detail where the boundary exists within an object oriented pluggable type system as used within the Checker Framework. It describes how unchecked values might enter checked code at specific points in the program.

2.1 Type Hierarchy and the `@Dynamic` Qualifier

In previous work on gradual typing systems, the type system was expanded with a new type representing the unknown or dynamic values originating from unchecked code. The same direction is taken here. In the Checker Framework, this type is represented by the `@Dynamic` type qualifier, which is built into the gradual typing portion of the Framework. It is up to the type system designer to incorporate the type into the type hierarchy.

In the Nullness type system in the prototype, dynamic is added to the type hierarchy as shown in figure 2.1. It is a subtype of the top type, Nullable, and a supertype of the bottom type, NonNull. It is beside PolyNull. However, incorporating the Dynamic type into the type hierarchy is only necessary to allow it to be used in the type system. The supertype

or subtype relationships defined to not define the Dynamic type's compatibility with other types. In general, type systems incorporate the Dynamic type into their type hierarchy below the top type, and above the bottom type, but with no particular relationship with other types.

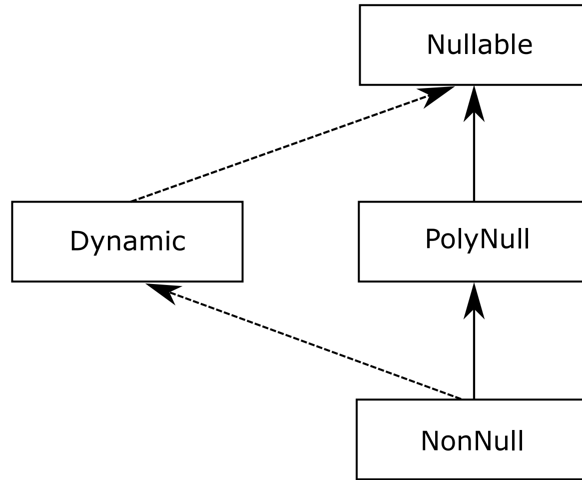

Figure 2.1: Type Hierarchy With the Dynamic Type

As described in section 1.1, Siek and Taha define the notion of type consistency in their formative paper on gradual typing [20]. This prototype adopts the same consistency relation by adding an additional test to the framework to determine if two types are consistent. The type hierarchy in the Nullness type system is quite simple, and types are never function types, nor structural types. This simplicity leads to the situation where all types are consistent with the Dynamic type. However, it would be possible in the Checker Framework to build a more complex type system, which would need to support the more complex version of the consistency relation as described by Siek and Taha. Providing the additional type consistency test in the framework allows a type system designer to implement this correctly for their type system.

2.2 Checked-Unchecked Boundary

When using a pluggable type system in a software project, since the type system is optional, some portions of the program may be checked by this type system, while other parts may not be. This means that a pluggable type system may not see the source code for the

entire program, and must make assumptions about the Checker Framework types in the interface of the unchecked portion of the program.

This could be the result of a software team deciding to adopt a pluggable type system within an existing project, at which point, the team may not want to incur the overhead of adding type annotations to the entire project at the same time. They may wish to only use the type system for particular portions of the program, or to gradually roll out the type system to the entire project.

Alternately, a software project may make use of a number of third party libraries which do not use all of the type system used by the main project, or those libraries may make use of a type system that the main project does not use. In both cases, circumstances arise where the type systems applied are different in the libraries and the main program.

In either of the above situations, a pluggable type system does not know, or cannot guarantee which pluggable type a certain expression has. Therefore, pluggable type system must make assumptions about the type of certain expressions. When considering applying gradual typing to a pluggable type system, runtime tests are inserted to verify that the runtime value conforms to the assumed compile-time type.

2.2.1 Runtime Tests

Implementing a runtime test involves determining if a runtime value conforms to a given type. How to actually implement this test is an important question. The framework should be general enough to allow any type system to implement its test, but simple enough that tests are easy to implement for type systems with simple tests. The framework must also be flexible enough so that highly optimized tests can be written. There are two considerations for implementing the test, the runtime value, and the actual type.

Runtime Value

In Siek and Taha's work [20] on gradual typing they describe using runtime tags to implement the check of the value type when an object crosses the statically checked boundary in the program. This technique is natural as they are exploring gradual typing in the context of typical dynamically-typed systems using tagged values.

The situation for gradually typed pluggable type systems is subtly but substantially different. In a pluggable type system, the type system is static in the typed or checked portion in the same way as traditional gradual typing. However, in gradual typing, in the

unchecked portion of the program, the type system does not exist. Accordingly, not only are the types not present at compile time, they also do not exist as type tags at runtime. The errors the type system is meant to catch may manifest in different ways at runtime, but do not necessarily appear as a type error. For example, when considering nullness, the Java runtime performs a nullness check at every dereference of a Java reference. This approach could be viewed as a dynamic type checking against null values within the Nullness type system. However, for other type systems, there may not be any dynamic checking. For this reason, this thesis uses the term ”unchecked” to differentiate it from ”dynamic” as used in most gradual typing systems.

The lack of value tags or other dynamic checking complicates creating runtime tests for some type systems, and there may be type systems for which runtime tests cannot be created. The case study in this thesis uses the Nullness type system, for which a runtime test is straightforward. A value at runtime can be inspected to determine if it is a null pointer or if it is not a null pointer. Another example is the Regular Expression checker. The Regular Expression checker verifies that strings that are used as regular expressions contain a valid regular expression as well as the correct number of capture groups. Since the validity and number of capture groups of a regular expression is easy to determine at runtime simply by compiling it, it is easy to create a runtime test for this type system, however, the test is somewhat more expensive because it involves compiling a regular expression. These systems can be seen as restricting the range of values that a variable can hold.

However, other type systems may not be as easy to test. For example, in the immutability type system, the runtime value of an object does not indicate if that object is an immutable value or a mutable value. Instead the type system enforces this restriction on behaviour, making it impossible to construct a runtime test. Some additional information about the object must be stored to determine if this object is immutable. This information is not enough, however, as tests to enforce immutable behaviour must also be inserted, and there is no way to prevent the unchecked portion from modifying fields on immutable values. It is unclear if gradual features could be applied to the immutability type system.

Generally, creating a runtime test for pluggable type systems that enforce restrictions on values is more straightforward. Since creating a runtime test is necessary for gradual typing, this suggests that pluggable type systems enforcing behaviour would not be good candidates for gradual typing. However, this rule does not apply to all type systems. The JavaUI type system enforces threading behaviours [9], but a runtime test involving checking the current thread is apparent here.

Runtime Type

How to pass the type to the runtime test is an important question when looking at performance of the resulting test. Since the test is executed any time a value crosses the checked-unchecked boundary, it should be fast to minimize the overhead.

There are two directions to approach the test, depending on the complexity of the type system. One can approach the test from the perspective of the value. A test can look at a value, and classify it into categories, and verify that a given category conforms to the provided type. For example, in the nullness type system, there are only two classes of relevant values: Non-null values which conform to any type, and null values which only conform to the `@Nullable` type. A test is therefore very simple to implement: check the value for null, and if null, check that the type is `@Nullable`.

The second direction to approach the type is from a type perspective. First, convert the value into the most specific type in the type hierarchy which is being checked, and then perform a subtyping test on the type. This approach is more complex than is needed for the Nullness type system, but for more complex type systems this would provide a type test which is consistent with the type judgements made at compile time. A developer may also be able to use the same code to perform the subtype test for compile time checking as for runtime checking, simply by compiling in the relevant code into the end user program. However, this check would be more expensive than categorizing values.

One caveat with the second approach, is that a type system may have multiple type hierarchies. For example, the Fake Enumeration checker, which verifies that integer values are valid for a given Fake Enumeration (integer enumeration) value, has a type hierarchy for every Fake Enumeration. In order to know which type to convert an integer too, the test must know which enumeration hierarchy is being tested against. This information should be available in the given type.

Depending on the needs of the test, passing the compile time type to the runtime test can come in many different forms. If a type system contains a finite number of types known at compile time, an enumeration of all types would provide a fast efficient mechanism. However, not all type systems meet these requirements. The ideal option would be to pass the actual `Type` object, used to represent types in the compiler, to the runtime test so that it has all the power it has at compile time. However, these type objects are complex, containing many aggregate objects forming a tree. There is no literal format for Java objects, so either a serialization protocol for all types must be defined, or a constructor tree must be created to generate each type object. Generating a constructor tree, while providing good generality and power, would add substantial complexity to the

framework to generate this series of calls, as well as require removing dependencies from `Type` objects so they can be created at runtime. Serialization is already available on `Type`, and is simple to implement. It also provides flexibility for simple type systems, allowing a text comparison instead of parsing the `Type` object structure. Generally, the framework should provide different options for type systems so that they can use whatever format, or custom format is best suited to each type systems' particular needs. For the case study in this thesis, string serialization has been used.

In terms of performance, many tests may be simplified or even eliminated. For example, in the nullness type system, only a single bit is needed: whether the type is `@Nullable` or not. Thus a very efficient test can be constructed. Even further, a nullness check is only needed if the type is `@Nonnull`, so the framework could eliminate many tests. However, these are very type system specific optimizations. The framework should allow for them, but not require them to implement a type system. These optimizations have not been implemented in the case study.

2.3 Defining the Boundary

The gradual checking portion of the pluggable type system must identify all points at which unchecked code interacts with checked code.

In the Checker Framework, the boundary between checked and unchecked code is at a compilation unit level. A source file is either processed by the Checker Framework type system or it is not. As a result, classes are either entirely checked, or entirely unchecked. For the purposes of a generic view of the system, consider a boundary where a given class is either checked or unchecked.

This chapter discusses the boundary in order of increasing complexity of the interactions. Section 2.3.1 first discusses a scenario where entire type hierarchies (inheritance trees) are checked or unchecked. Here, the only interactions between checked and unchecked portions are between classes from different inheritance trees. Classes within an inheritance tree are entirely checked or unchecked so there is no boundary. This section discusses how to deal with checked methods receiving unchecked arguments, and unchecked methods returning unchecked return values into checked code.

Section 2.3.2 discusses the possible ways to deal with fields, both checked and unchecked fields.

Section 2.3.3 deals with the complexity added when parts of an inheritance tree can be checked or unchecked. This can happen because the boundary occurs on the line of

classes. A class may be checked, but it may inherit from an unchecked class. Virtual method dispatch adds a new level of complexity to the boundary because it is unknown at compile time if a method call will dispatch to checked or unchecked code. Section 2.3.3 first discusses checked classes inheriting from an unchecked ancestor, then how to deal with the possibility of an unchecked class inheriting from a checked class. It then notes that these scenarios may be combined in any configuration. It then goes on to discuss interfaces as a special case of inheritance.

2.3.1 Return Values and Parameters

The most obvious way in which unchecked code can interact with checked code is when a checked portion of the code uses the return value from an unchecked method. This situation is easy to identify in the abstract syntax tree, as all expressions and values in the checked portion receive a type. Since this includes method calls, the Checker Framework type checking algorithm tries to lookup the type of the method call. Since it does not find a type, it tries to use a default type. The gradual typing feature simply sets the default return value type for unchecked code to be the `@Dynamic` type. For each assignment from Dynamic to a concrete type, a runtime check can be inserted to validate that the runtime value is compatible with the compile-time static type.

This scenario is described in figures 2.3 and 2.4 and the resulting transformation needed to safely check return values is shown in figure 2.5. Figure 2.6 also describes the insertion of this check.

In order to test the value in isolation, the test occurs in the body of a method that can be called within the expression. The test method also returns the original value back to the caller. This method call can wrap the `@Dynamic` value to isolate it from the expression. More specifically, the expression which results in the `@Dynamic` typed value is used as the argument for the call to the test method, and the test method call replaces that expression within the parent expression. Since the method simply returns its parameter after testing, the behaviour of the program is preserved. Figure 2.2 shows the check method, which is a component of the gradual checking framework.

Checked and unchecked code can also interact through parameters of checked methods. In this situation, a checked method that has types assigned to parameters may be called by unchecked code and passed unchecked values for arguments. At this point, runtime values that are incompatible with the static parameter types could be passed, invalidating the type checking that validated the method. Runtime checks are required to prevent this from occurring. However, at other times, checked code may call this same checked method.

```
public class RuntimeCheckValue {
  public static Object checkValue (Object value, String type) {
    if (![runtime check value, type]) {
      [error]
    }

    return value;
  }
}
```

Figure 2.2: Runtime Check Method (part of the gradual checking framework)

```
public class A {
  static Object foo() {
    [unchecked code]
  }
}
```

Figure 2.3: Unchecked Method: A.Java

```
public class B {
  void bar() {
    @NonNull Object myObject = A.foo();
  }
}
```

Figure 2.4: Calling Unchecked Method: B.Java

```java
public class B {
  void bar() {
    @NonNull Object myObject = RuntimeCheckValue.checkValue(A.foo(),
                                                            "@NonNull Object
                                                               ");
  }
}
```

Figure 2.5: Safe Calling Unchecked: B.Java

In this case, runtime tests are unnecessary, since the type of the arguments has already been checked at compile time. In order to maximize performance of the checked portions of the code, runtime type checks must be avoided here.

To achieve this, a new method is created. The new method contains the original method body in its entirety, and is given a name by taking the original name and appending to it `_safe`, forming: `<method_name>_safe`. The original method receives a new body containing a runtime check on each of its parameters followed by a call to the safe version. Then all method invocations in the checked portion of the program, which call to checked methods, are modified to call the safe version of the method; hence checked code uses the fast path, which incurs no penalty from executing runtime checks, while unchecked code remains unmodified, and thus calls the original method, which now contains runtime checks. Therefore, runtime checks enable type safety between unchecked code and typed code, while not incurring any penalty in runtime performance of typed code. This situation is shown in figures 2.7 and 2.8 and the transformation is shown in figure 2.9.

2.3.2 Fields

Another point of interaction between checked and unchecked code is through fields. Obviously, reading from an unchecked field results in an unknown value. The same method as used for unchecked method return types applies here. The default pluggable type of any unchecked field is `@Dynamic` and runtime tests are generated to convert from `@Dynamic` to any other type.

However, this approach does not extend to unchecked references into typed fields. Unchecked code may still write a value to a field that is incompatible with the field's compile-time Checker Framework type system type. There does not seem to be an obvious

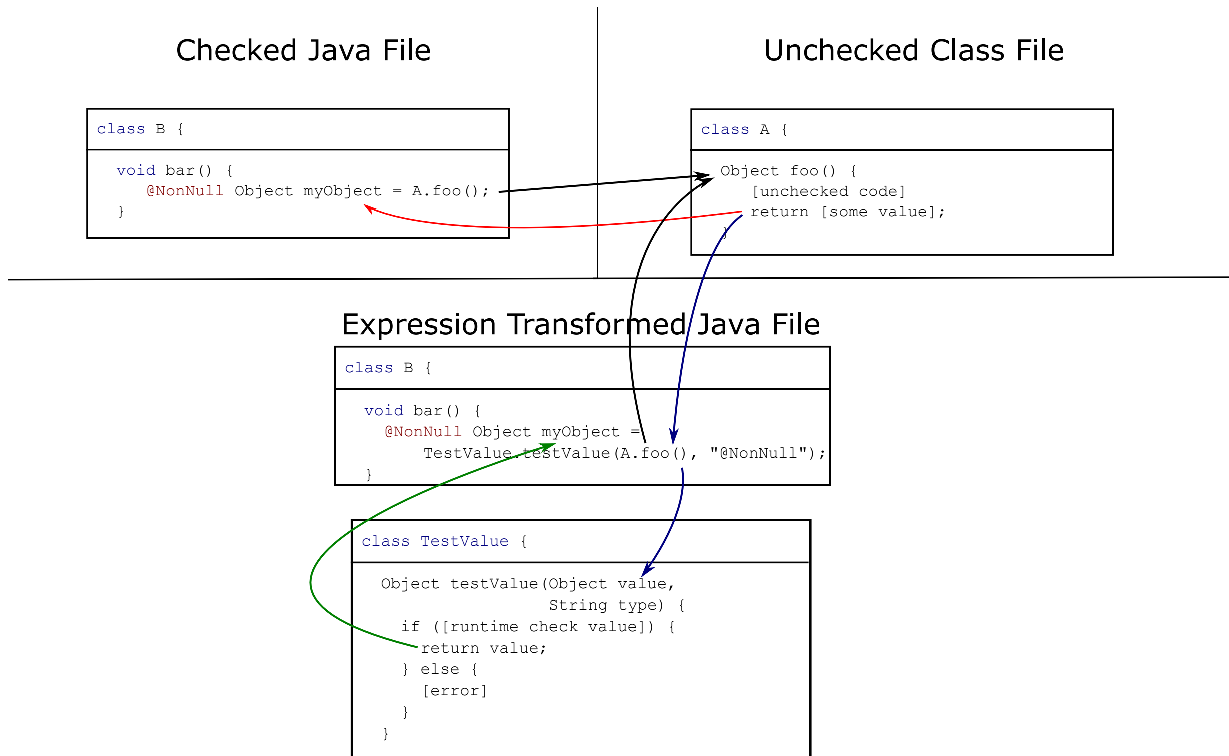

Figure 2.6: Unchecked Return Value

way of performing the same transformations as used for methods. Since fields cannot execute code upon reference, there is no way for the compiler to instrument fields to perform any type of check when they are written to. Field accesses in Java have no hooks, and do not execute checked code, thus there is no way for checked code to be informed of a write to these fields. There are two solutions, neither of which is ideal.

Treat Accessible Fields as Unchecked

The first is to treat any field whose permission and visibility modifiers allow it to be modified by unchecked code as unchecked itself. This conservatively provides type safety because a field, which can be written by unchecked code, could contain an invalid value, and there is no way for the type checker to determine if this occurs. Since there is no way to determine if accessible fields are actually written, anywhere they are used must then contain a check. Unfortunately, this causes runtime tests to be executed even if the field has never been touched by unchecked code. This is not ideal from a performance

```
class A {
  static void bar(@NonNull Object param) {
    [dereference param]
  }

  static void foo() {
    A.bar(new Object());
  }
}
```

Figure 2.7: Checked Method: A.Java

—

```
class B {
  static void foo() {
    A.bar(null);
  }
}
```

Figure 2.8: Calling Checked Method From Unchecked Code: B.Java

```java
class A {
  static void bar(Object param) {
    if ([runtime check myObj, "@NonNull Object"]) {
      A.bar_safe(myObj);
    } else {
      [error]
    }
  }

  static void bar_safe(@NonNull Object param) {
    [dereference param]
  }

  static void bar() {
    A.bar_safe(new Object());
  }
}
```

Figure 2.9: Method Transformation to Achieve Safe Calling: A.Java

perspective. However, this would only apply to fields that are modifiable from unchecked code. In Java, this means non-private, non-final fields.

Instrument Bytecode

The second solution is to attempt to instrument bytecode at load time. It should be possible to modify the bytecode of unchecked class files at loading time to modify their field references. In this solution, a setter function would be generated for each accessible and modifiable field, containing the appropriate runtime tests. Then upon loading bytecode that is not known to be checked, writes to these fields would be replaced with an invocation of the setter function. Setter functions would be added by the gradual checker at compile time. This solution should be a simple transformation to the bytecode, and as long as all write references to the field are transformed in this way at runtime, the compiler knows that fields are type safe. However, this solution deviates entirely from the compile time static checking of the existing Checker Framework, and is beyond the scope of the project. This solution would also enable many other techniques and is an interesting avenue for future work, but is not pursued in this work.

Given that Java's recommended practices involve using getters and setters to access

fields, while keeping the field itself private, in practice few fields should actually be accessible to classes in external libraries. While in the first two interaction scenarios, this would indicate this situation is unusual, the next section discusses how untyped code may have access to many more fields than expected. For the purposes of the case study, modifiable fields are treated as unchecked.

2.3.3 Inheritance

The first two scenarios in which unchecked code interacts with checked code are fairly straightforward. However, when accounting for the inheritance relationship in Java, it becomes far more complex. When considering inheritance, there are two possible interactions. A checked class can inherit from an unchecked class, and an unchecked class can inherit from a checked class. Interfaces provide a special case, which is shown in section 2.3.3. While it is similar to either of the cases with classes, there are some special considerations necessary because of the restrictions placed on interfaces.

Each type of relationship is described by a code example, and several transformations on that example to show different attempts to achieve type safety and correctness by modifying the program and inserting runtime checks.

Checked Code Inheriting from Unchecked Code

A possible configuration is to have a checked class inherit from an unchecked class. This is illustrated in Crefuntyped-A.Java,typed-B.Java.

This code example shows two classes in an inheritance relationship; the typed class `B` extends the untyped class `A`. This situation is known by the compiler at compile time since all classes in the inheritance chain must be known statically at compile time. Further, this situation occurs for all classes in the program, as all classes eventually inherit from `java.lang.Object`, which is itself unchecked.

?? ?? shows class `B` after the transformation. In both cases, `otherMethod` and `otherMethod2` calls to the method `foo` are dispatched to the correct method, which now contains runtime checks, and in the case of a checked method call, the correct method is `foo_safe`, which avoids the checks. Any call within typed code is translated to `foo_safe` and thus avoids runtime checks on the parameters when it is processed by the compiler. But all untyped code, including code in `B` dispatches to the original method, which now contains runtime checks, since it never gets processed by the gradual typing framework.

The only complexity is the super call in `B.foo_safe`. In this example, the super call is directly to `foo`. This is because the next class in the hierarchy which implements a method called `foo` is in unchecked code. If it was in checked code (meaning there was an intervening class between `A` and `B`), then the super call should instead be to `foo_safe` since in that checked class the safe, unchecked version of the method has been renamed to the new name. This situation can be detected statically at compile time, thus no runtime check is needed. Additionally, it can be known statically, if compile time checks are required for the return value.

Unchecked Inheriting from Checked

Figures 2.14 and 2.15 demonstrates an unchecked class inheriting from a checked class. This Section discusses dynamic or virtual dispatch and how to handle it. Virtual dispatch also applies to the situation where a checked class inherits from an unchecked class, and is applied in that scenario as well; it is omitted for clarity. The other complexity with this scenario is that the compiler cannot know if it will occur while compiling the base class, since any later compilation units can extend the inheritance tree.

Figures 2.14 and 2.15 show two typical classes in an inheritance relationship. Class `A` is checked by the Checker Framework, but class `B` is not.

In this situation, there are two possible calls to the method `foo`. A call can be to the super method, `super.foo`, which is not a virtual method call, and is statically bound; or it can be called on an object, `this.foo`, which is a virtual method call and not statically bound at compile time. Both of these method calls can originate either from within the checked portion of the code, or within the unchecked portion of the code.

Since there is an inheritance relationship between `A` and `B`, neither class can continue to be considered entirely checked or unchecked. This results in several issues with our original scenario, as well as some new considerations.

Figure 2.16 shows the transformation that would result from the system described in the previous section. It protects all calls to `foo` by creating a new method `foo_safe` that performs the original function body, and giving the original method a new function body, which performs a runtime check and then calls the safe version.

The function calls in the unchecked code now call the method with the runtime tests to validate the parameters, and thus are checked for safety. The super method also calls the correct method, as the overriding method is unchecked, but calls the checked portion.

The field access to `bar` is also checked since the field, while protected, is visible to any class extending `A`, and thus could have an unchecked value in it, as demonstrated by

setting a value in B.otherMethod2. If this field was instead private, or final, it would not be visible or modifiable by B and this check would not be needed.

However, when considering dynamic or virtual method dispatch, a change in behaviour has occurred. In the checked portion in function A.otherMethod, the function call to foo could dispatch to B.foo if the runtime type of the object this is actually B (this instanceof B). However, since the function is renamed to foo_safe, that virtual dispatch can no longer occur. In fact, no virtual method dispatch could occur into unchecked code from checked code since unchecked code never contains _safe methods.

To overcome this incorrect virtual method dispatch, a runtime check is required to determine if the runtime instance of the class is a class which has been checked by the Checker Framework type system. If it is, the method should dispatch to a _safe version, otherwise it should dispatch to the unmodified version. This runtime check cannot be avoided in this framework, and adds to the overhead of all method calls, except for those where the class or method is marked final within checked code, and thus cannot be overridden, or for those methods with no parameters to check.

One method to implement the runtime check is to flag each class that has been type checked with a marker, and check for that marker field on the class in the runtime check. Instead of transforming each foo call to foo_safe, a new method foo_maybe is created, which performs the runtime check and dispatches to either foo or foo_safe depending on the actual runtime class.

Figure 2.17, shows the final transformation. The foo_maybe method checks for the existence of the field checked_flag, which indicates that the actual runtime class of the object has been processed by the Checker Framework type system, and thus, it is safe to dispatch to foo_safe instead of foo. However, this is not always necessary. If B did not override foo, the runtime check would still dispatch to foo instead of foo_safe even though this would go straight to the checked code. Thus, additional runtime checks are incurred.

It is worth noting that these scenarios can all occur, possibly multiple times, within the same hierarchy chain. Figure 2.18 shows the use of the maybe method within the inheritance from unchecked code scenario. This figure clarifies exactly which calls are either unsafe, or missing in each scenario. The top two quadrants show the original code. The bottom left code block shows the transformation without the runtime maybe test as described in section 2.3.3, and the bottom right shows the same code with the maybe transformation. For each possible transformation, arrows indicate function calls which enter and leave the block. The colour of the arrows indicates if they are correct (green), incorrect (blue), or unsafe (red). In the transformation according to section 2.3.3 the

virtual method dispatch calls are incorrect. The final transform shows all outgoing and incoming function calls are correct.

Interfaces

In many ways, the situation with interfaces is similar to classes. However, interfaces have no method implementations, so only when combined with a class is there checked code. However, virtual method dispatch still causes some complexity when combined with interfaces. Figures 2.19 and 2.20 illustrate this scenario.

In this example, the interface `I` is provided, and implemented by both checked class `A` and an unchecked class `B`. In this case, the interface is provided in checked code. Both classes contain a method call to a method in the interface, on an object typed only as `I`, that at runtime could be an instance of either `A` or `B`.

In order to have the call that originates from unchecked code checked at runtime, the same transformation must be done. However, transforming the class implementing the interface (`A`), and then renaming the interface method call in `A.otherMethod` would result in a compiler error after transformation since `I` does not have a method `foo_maybe`. The compiler needs to know which class to cast the value `i` to in order to call `foo_safe` or `foo`. If both the class and the interface (`A` and `I`) are transformed, then the interface will have the method `foo_safe` and `foo_maybe`, so the renamed method call binding succeeds, but, since unchecked code does not implement these methods, the interface implementation is incomplete, and again, a compilation error results.

The solution to these problems is to use Java 8 default interface methods. This feature allows an interface to have a default implementation of a method, which provides an implementation using only the interface methods, to allow new methods to be added to an interface without requiring all implementing classes to be updated. Figure 2.21 shows this transformation.

Using the default method feature, the interface can be modified to have the two `foo_safe` and `foo_maybe` methods without breaking the implementation in `B`, and also allowing the method call in `A.otherMethod` to be transformed and still execute successfully. Note that `foo_safe` has an exception thrown as the method body in the default implementation. It should never be called, since `foo_maybe` decides between `foo` and `foo_safe` depending on the presence of the `checked_flag` which is only present if the implementing class has been type checked and thus contains an implementation of the `foo_safe` method. However, the `foo_maybe` method is implemented in the interface,

because it is possible that it may be called on an instance of `B`, and thus must work correctly in that context.

The final configuration with interfaces to consider, is an unchecked interface. Because a class is required to implement it, that class would have Checker Framework types associated with the method parameters, and the method body would use these Checker Framework types. However, since the interface itself is unchecked, it is impossible to transform it as shown previously. Performing the transformation of methods in the class results in all method calls with a receiver reference that is typed with the class type being transformed safely, but method calls with a receiver reference typed with the interface type not being transformed and the only option is to call the original method. This call incurs runtime checks, even if the class is checked. However, this behaviour is actually desirable since the method parameters are not checked statically anyway, because the interface only has default types. Thus, for interfaces, in unchecked code, calls always incur runtime checks for the parameters, even if the method is actually implemented in checked code.

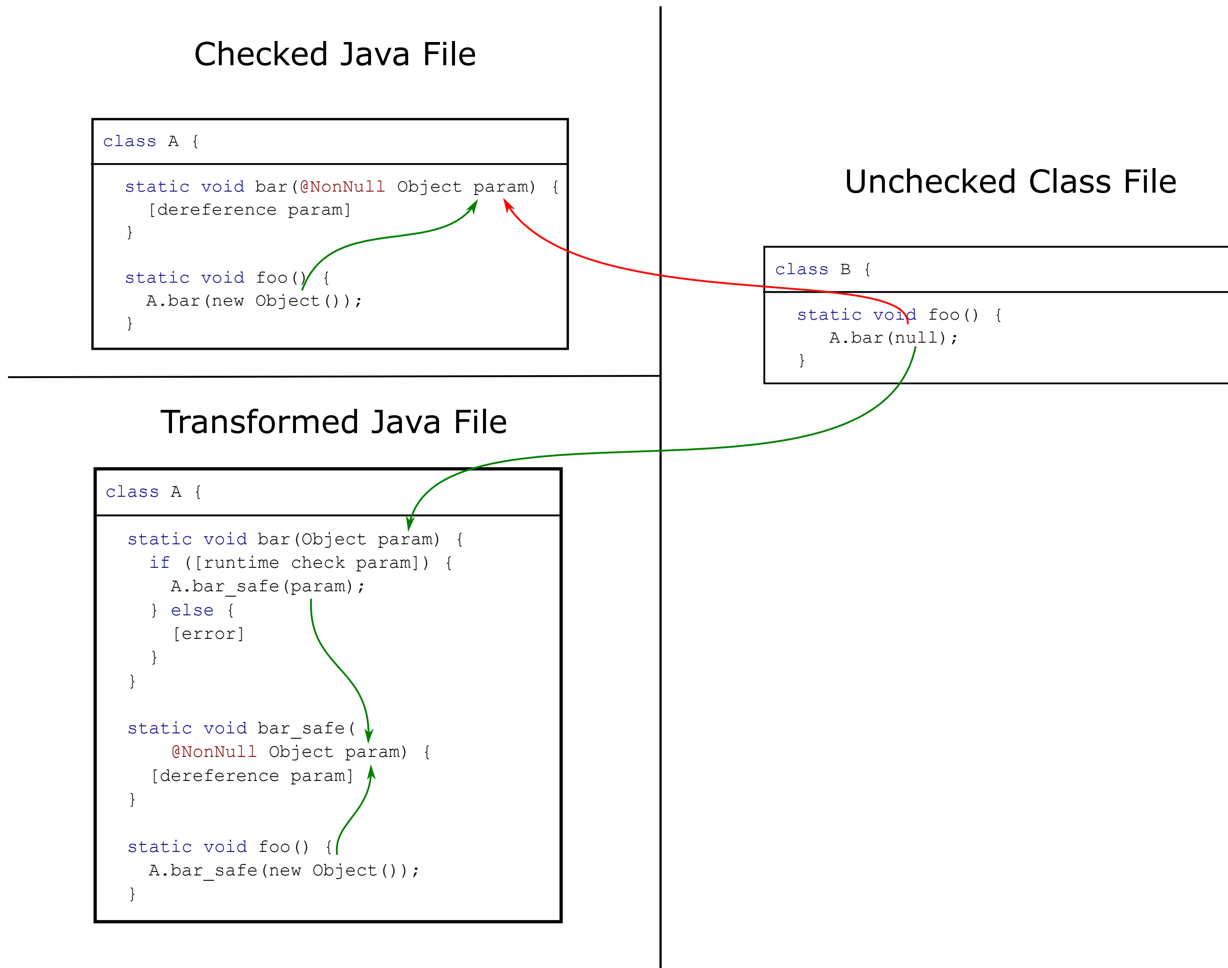

Figure 2.10: Unchecked Argument Value

```java
class A {
  Object foo(Object param) {
    ...
    return new Object();
  }

  void otherMethod() {
    this.foo(null);
  }
}
```

Figure 2.11: Unchecked Portion: A.Java

```java
class B extends A {
  @Override Object foo(@NonNull Object param) {
    ...
    return super.foo(param);
  }

  void otherMethod2() {
    this.foo(new Object());
  }
}
```

Figure 2.12: Checked Portion: B.Java

```java
class B extends A {
  @Override Object foo(Object param) {
    if ([runtime check param, "@NonNull Object"]) {
      return this.foo_safe(param);
    } else {
      [error]
    }
  }

  Object foo_safe(@NonNull Object param) {
    ...
    return super.foo(param);
  }

  void otherMethod2() {
    this.foo_safe(new Object());
  }
}
```

Figure 2.13: Checked Portion After Transformation: B.Java

```java
class A {
  Object foo(Object param) {
    ...
    return new Object();
  }

  void otherMethod() {
    this.foo(new Object());
  }
}
```

Figure 2.14: Checked Portion: A.Java

```
class B extends A {
  @Override Object foo(@NonNull Object param) {
    ...
    return super.foo(param);
  }

  void otherMethod2() {
    this.foo(null);
  }
}
```

Figure 2.15: Unchecked Portion: B.Java

```
class A {
  Object foo(Object param) {
    if ([runtime check param, "@NonNull Object"]) {
      return this.foo_safe(param);
    } else {
      [error]
    }
  }

  Object foo_safe(@NonNull Object param) {
    ...
    return new Object();
  }

  void otherMethod() {
    this.foo_safe(new Object());
  }
}
```

Figure 2.16: Checked Portion after Simple Transformation: A.Java

```java
class A {
  private Object checked_flag = null;

  Object foo(Object param) {
    if ([runtime check param, "@NonNull Object"]) {
      return this.foo_safe(param);
    } else {
      [error]
    }
  }

  Object foo_maybe(Object param) {
    if ([this.has(checked_flag)]) {
      return this.foo_safe(param);
    } else {
      return this.foo(param);
    }
  }

  Object foo_safe(@NonNull Object param) {
    ...
    return new Object();
  }

  void otherMethod() {
    this.foo_maybe(new Object());
  }
}
```

Figure 2.17: Checked Portion after Full Transformation: A.Java

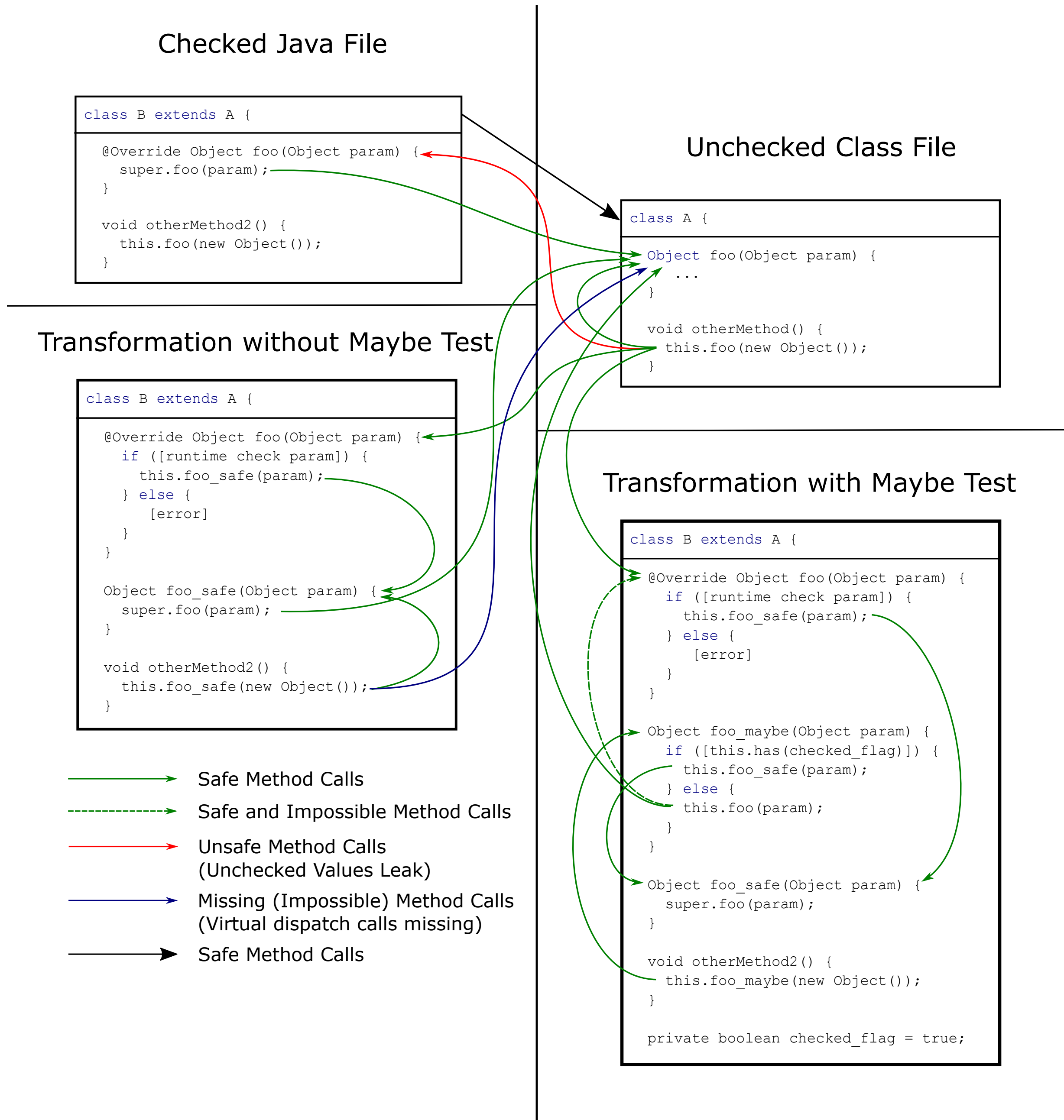

Figure 2.18: Inheriting from Unchecked

```
1  interface I {
2    Object foo(@NonNull Object param);
3  }
4
5  class A implements I {
6    Object foo(@NonNull Object param) {
7      ...
8    }
9
10   void otherMethod(I i) {
11     i.foo(new Object());
12   }
13 }
```

Figure 2.19: Checked Portion: A.Java

```
1  class B implements I {
2    Object foo(Object param) {
3      ...
4    }
5
6    void otherMethod2(I i) {
7      i.foo(mi;;);
8    }
9  }
```

Figure 2.20: Unchecked Portion: B.Java

```
interface I {
  Object foo(Object param);

  default Object foo_safe(@NonNull Object param) {
    throw RuntimeException();
  }

  default Object foo_maybe(Object param) {
    if ([this.has(checked_flag)]) {
      return this.foo_safe(param);
    } else {
      return this.foo(param);
    }
  }
}

class A implements I {
  private Object typed_flag = null;

  Object foo(Object param) {
    if ([runtime check param, "@NonNull Object"]) {
      return this.foo_safe(param);
    } else {
      [error];
    }
  }

  Object foo_safe(@NonNull Object param) {
    ...
  }

  void otherMethod(I i) {
    i.foo_maybe(new Object());
  }
}
```

Figure 2.21: Checked Portion: A.Java

Chapter 3

Implementation

This chapter discusses specific details of how the prototype framework and prototype gradual nullness type system are implemented. Section 3.1 discusses the architecture of the OpenJDK™ Java Compiler, and how these architectural choices impact the design of the Checker Framework, and the prototype described in this thesis. It also discusses the phases of compilation, and the specific points at which the Checker Framework and the prototype hook into these phases. Section 3.2 describes the Checker Framework, and how it integrates with the OpenJDK™ Java Compiler. Section 3.3 explains how the runtime transformations described in Chapter 2 are implemented, and how the runtime checks are implemented. Section 3.4 describes how to overcome limitations on constructors in Java in order to achieve the required transformations for constructors. Specifically it discusses the use of a dummy parameter to mark checked methods, as well as the method of checking parameters to overcome the Java requirement that the first statement of a constructor be a `super` or `this` constructor call. Finally, Section 3.5 describes the implementation of the runtime, which checks if the actual runtime type has been checked by the Checker Framework.

3.1 OpenJDK™ Java Compiler

The OpenJDK™Java compiler is a free and open source Java compiler that transforms Java source files into Java class files containing Java bytecode. Java bytecode is the hardware independent instruction set that is executed by the Java virtual machine (JVM).

In Java, code (bytecode or source code) is organized into classes, representing templates for objects. Class files contain the bytecode for exactly one class, plus a number of constants

used to reference other classes. Java source files can represent only one publicly visible class, but may contain multiple classes with different visibility modifiers. As a result one source file may generate multiple class files.

Java also allows circular references in classes. That is to say class `A` can reference class `B` even though class `B` also references class `A`. This provides flexibility to Java programmers but somewhat complicates the compilation process.

Java annotations are a Java language feature providing developers with a syntax to tag declarations and types with an annotation. The annotations can be viewed in source code, used by development tools, processed during compilation, or even accessed at run-time. These annotations can be used for any purpose; the Java standard provides several by default including `@Deprecated()`, which allows developers to mark a function as deprecated to generate a warning, before actually removing it.

The Java annotation processing framework is the Java application programming interface (API) used to process these annotations during compilation. The OpenJDK™Java compiler provides an implementation of this API to allow developers to write plugins that process annotations in a source program during compilation. The user specifies which annotation processors to use on the command line when invoking the compiler. During compilation, these annotation processors are invoked to process the annotations present in the Java source. Figure 3.1 shows the phases of compilation a compilation unit will pass through, and illustrates where the annotation processing phase occurs.

Compilers are conceptually seen to execute as a series of phases: scanning (or lexing), parsing, various language specific analysis phases including type checking, and code generation.

In the Java language, and specifically in the OpenJDK™Java compiler, the language specific phases include in order, annotation processing, type checking, de-sugaring (or simplifying language constructs to base constructs), and optimizations. Code generation results in Java bytecode.

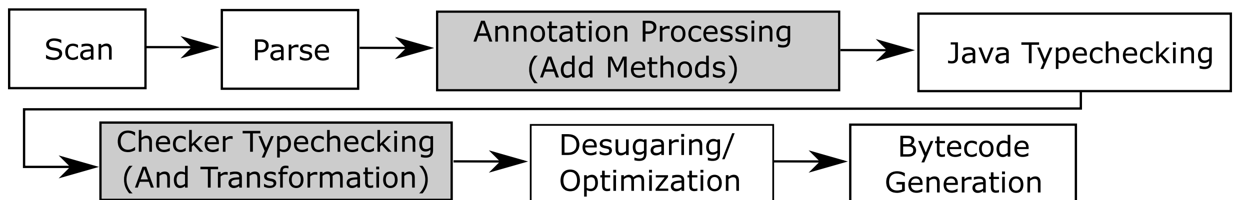

Figure 3.1: Phases of the OpenJDK™Java Compiler. Grey boxes indicate compilation phases during which the gradual checking prototype modifies the Java AST.

This idealized view of compilation phases is not entirely realistic. A compiler is often invoked with many source files so it may be compiling more than one file at a time and not all files are necessarily at the same phase at the same time.

In languages like C, which require all names or symbols to be forward declared, a dependency tree of files can be built, indicating which files must be processed first, in order to generate the symbols needed for depending files. Since Java allows circular references without forward declarations, the file dependency graph might not be a tree, and there may not be an ordering which ensures all symbols a class uses are compiled before that class.

As a result, the Java compiler may begin compilation on one file, but once it reaches the type checking phase, it may encounter a symbol defined in another file. It may then begin compiling that file, before the first file is completed. In the event of circular dependencies, the Java compiler can create a 'thunk' to stand in for one symbol in the circular chain, and compile all files in that chain. It compiles the file defining the symbol the thunk is standing in for, and reconcile the uses of the thunk with the actual symbol.

Annotation processing is done in a series of rounds. During each round, a set of `TypeElement` objects is passed to the annotation processor. A `TypeElement` represents a Java class type that should be processed by the annotation processor. Since circular dependencies mean it is impossible to ensure that all dependent symbols that a `TypeElement` references are processed first, during the annotation processing round, it is possible that some symbols may not have a Java type fully instantiated for them.

The Java compiler does not provide an interface for modifying the Java abstract syntax tree during annotation processing. The Java standard indicates that annotation processing should not modify the resulting bytecode, and so no interface to do so is provided. However, since the compiler is open source, it is possible to cast objects from the public interface type into the actual implementation class type to access this functionality.

3.2 Checker Framework

The Checker Framework [5] is a pluggable type system framework for Java. It is built on, and depends directly on the implementation of the Java compiler provided by OpenJDK™. It provides a framework where type system designers can easily develop a type system that runs on top of the Java type system.

The framework provides a set of meta annotations (annotations that apply to annotations) allowing type system designers to declaratively define their type hierarchy and

defaults in the form of a set of annotations. End users of the type system can apply these type annotations to their programs to annotate the Java types with additional type information for the new type system.

The core of the framework provides an annotation processor class, a default type hierarchy, a type defaulting mechanism, and a default type-checking algorithm, all of which a type system designer can extend or override to customize the processing of their type system.

The Checker Framework is invoked during annotation processing, however, as discussed in section 3.1, not all the Java types of a `TypeElement` are fully instantiated at this time. This poses a challenge for the type checking portion of the Checker Framework. It would be unable to differentiate between a Java type-error that is found later in the Java compiler's type checking phase, and a Checker Framework type-error. As a result, the Checker Framework performs no type checking at this point; instead it installs a listener to listen for the type checking completed event emitted by the Java compiler.

When the Checker Framework receives the type-checking completed event, it now knows that all the Java types in all the `TypeElement` objects in the compiler have been instantiated and checking the new type system can now proceed. Creffig:Java-compiler-phases shows the phases of the Java compiler and which phases the Checker Framework hooks.

3.3 Runtime Transformations

The Java compiler annotation processing application programming interface (API) does not provide any mechanism for modifying the actual Java abstract syntax tree (AST). In order to make the earlier described transformations, the AST must be modified by the Checker Framework. Investigations into the Java compiler, and an earlier experimentation by Sampson et al. on EnerJ [17] showed that it is possible to use internal APIs to modify the AST tree during annotation processing, and after Java type checking has completed.

There are several stages of compilation at which the tree may be modified. The gradual transformations discussed require hooking into two different phases. The first is during the standard annotation processing. At this point, the AST has not been fully type checked, and not all types are available. However, it is during this phase that new methods must be inserted. This, requirement is to overcome Java's circular dependency features. Since Java type checking is not complete, the gradual type system can insert new methods, and new method calls, even though the methods they reference have not yet been inserted into the classes they are being invoked on. Since the Checker Framework knows it will eventually

process those classes and add the required methods to those classes, type checking will eventually succeed.

The second phase the gradual typing system must hook is after all type checking has been completed. The Checker Framework listens for the compiler's type checking completed events. This phase is useful to the Checker Framework as all Java types for all expressions are fully realized and complete, and any Java type errors in the program have already been caught. At this point, the Checker Framework runs its type checking algorithm and instantiates all Checker Framework type system types. Once all types have been realized, the second phase of transformations can be completed by the gradual type system.

The first transformation is to insert runtime checks at every point where a `@Dynamic` value is used. The exact nature of these runtime checks are described in section 3.3.1.

The second transformation is to fill in type information for tests on method parameters that were constructed in the first phase. During the first phase, new methods are added to classes that contain runtime tests. However, since neither Java type checking, nor Checker Framework type checking is complete at the time, the gradual type system does not know what type to test against. Instead, a dummy value is used as the type for test. Later, during the second phase, the gradual type system looks for and replaces these dummy values with actual Checker Framework types.

3.3.1 Runtime Checks

As described in section 2.2.1 the type system and what possible runtime values occur in each concrete type influence exactly what form a test takes. However, the type system designer does not need to develop their runtime test in the form of an AST. The framework should provide an interface to allow them to build tests in a standard format, which can be invoked by the transformed AST as needed.

The framework should support an efficient implementation of the test. To this end, the gradual type system framework provides for the language designer to implement the test as a static method. The designer then provides the class and the method, using Java reflection, to the framework, which then calls it from the runtime test site in the AST tree. This avoids any dynamic or virtual dispatch overhead that would result from providing an interface or base class for the language designer to override. This flexibility allows the type system designer to implement as efficient a test as is possible. The interface for these methods is shown in figure 3.2, however, since the methods are provided to the Framework using Java reflection, only the parameter list types are important. The actual naming of the method or parameters is not relevant to the interface, and is given here for illustration.

```java
class TypeSystemRuntimeTest {
  static boolean testValue(Object value, String type) {
    ...
  }

  static void typeError(Object value, String type) {
    ...
  }
}
```

Figure 3.2: The Interface For Runtime Test and Error Functions

The framework should, but does not currently, provide the ability for the type system designer to decide how to pass the compile-time type information. Currently, the type is serialized into a Java string, and the designer's static method must accept a string. The framework could easily provide the ability for the type system designer to choose what data type should be passed to their runtime test method, and then provide a method to convert compile time types to this data type. However, because Java lacks a general literal format for objects, this would have to be a type that can be specified by a literal in Java code. If the type system designer wanted to pass a custom type to their runtime test function, they would need to provide code to build an AST tree, which would result in that data type's construction. This requirement would be a very high bar for a type system designer. This is one reason to prefer supporting String serialization.

A final optimization that the prototype framework does not support, but which would improve performance, would be supporting different methods for Java types other than reference types. The framework currently boxes all types and passes them to the type system designer's runtime test method as an Object. Boxing and unboxing of primitive Java types is unnecessary overhead. Additionally, the type system designer may be able to optimize their test if they know the runtime value is a primitive type.

In the event of a runtime test failure, the type system designer provides a second method that can be called to report an error, or simply log the error.

```java
class A extends C {
  A(Object param) {
    super(param);
    ...
  }

  void otherMethod() {
    A foo = new A(new Object());
    ...
  }
}
```

Figure 3.3: Checked Portion: A.Java

```java
class B {
  void otherMethod2() {
    A foo = new A(null);
    ...
  }
}
```

Figure 3.4: Unchecked Portion: B.Java

3.4 Constructors

Another implementation difficulty is constructors. Because Java constructors have several restrictions on them, the transformations described in section 2.3 do not map directly onto them. First, constructors must, as their first statement, call the super constructor. Second, constructors are not named and thus cannot have names mangled to `_safe` versions. Figures 3.3 and 3.4 illustrate this situation.

In this example, class `A` has a one parameter constructor, which is invoked both from checked and unchecked code. This example is similar to a method call from checked and unchecked code. Using the method translation, the class would gain a `_safe` version of the constructor, and then checked code would call that version. The unchecked version would continue to invoke the original constructor, which would now have runtime tests for all the parameters. This solution presents two problems. First, Java does not allow flexibility

for naming constructors, so it is impossible to rename one with `_safe`. Second, Java requires that the first statement in the constructor body be a call to a super constructor, or another constructor for this class (either a `this()` call or a `super()` call). However, in this example, the runtime tests are the first statement in the constructor.

To solve these problems, there must be a way to differentiate between safe constructors without runtime checks and unsafe constructors with runtime checks. Java classes can have multiple constructors, but they are not disambiguated by name, only by parameter types. Thus, to create a safe version of the constructor, a transformation would need to create a constructor with a different parameter list. This approach is unsafe, however, as it would be possible for an end user to create a constructor with a conflicting parameter list. The transformation, ideally, would choose a parameter type that is impossible for the end user to use in their program. For the purposes of this system, it sufficed to choose a type which the user is unlikely to choose. A dummy class can be used for this purpose. While an end user could use this class to create a conflicting parameter list, it would result in a compile error, for two identical constructors. This approach adds a small amount of overhead to calling a constructor as an additional parameter must be passed. The value can simply be a null.

To overcome the requirement to have the super (or this) constructor call be the first statement in the constructor, a static method call can be used. A method call is an expression, thus it can be used as an argument to calling the safe constructor. Each parameter is passed to the static function, which performs the runtime check, verifies the result, and either returns the same value again, or throws an error. Figure 3.5 illustrates the new transformation.

Figure 3.5 shows the new class added to the framework providing the dummy type. It shows the new method for testing the parameter value. It is important to note that although the parameter test method, `ConstructorParamTest.testConstructorParam`, returns the same value it is passed in its parameter, the actual return type is `Object`. In this case, `Object` is the type of the parameter the compiler expects, however, this is not always the case. Thus, the return value of the parameter test method must be cast to the correct type when calling the constructor. This ensures that the correct constructor is matched by the types, in case there is a more general constructor overload available.

Figure 3.6 shows the original constructor parameter list, which now simply calls the test function on every parameter and passes the returned values (with appropriate casts) as arguments to the new constructor along with a null value cast to the marker type (`SafeConstructorMarkerDummy`). Additionally, the new constructor is shown, which has the additional marker parameter.

```java
public class SafeConstructorMarkerDummy {}

public class ConstructorParamTest {
  public static Object testConstructorParam(Object param, String type) {
    if ([runtime check param with type]) {
      return param;
    } else {
      [error]
    }
  }
}
```

Figure 3.5: Code Added to the Gradual Checking Framework

```java
class A {
  A(Object param) {
    A((SafeConstructorMarkerDummy) null,
      (Object) ConstructorParamTest.testConstructorParam(param, <type>));

  A(SafeConstructorMarkerDummy dummyParam, Object param) {
    super((SafeConstructorMarkerDummy) null, param);
    ...
  }

  void otherMethod() {
    A foo = new A((SafeConstructorMarkerDummy) null,
                  new Object());
    ...
  }
}
```

Figure 3.6: Checked Portion A.Java

Finally, the call to the constructor in `A.otherMethod` has been modified to pass an extra null value, cast to the marker type, in order to match to the safe version of the constructor. In the unchecked class `B`, the original code remains, and thus, the constructor with the runtime checks is called instead. No unnecessary runtime checks are performed, but all unchecked parameters are checked.

This method also works with inheritance as described in section 2.3.3. Checked classes that extend this class have the super call modified to pass the extra parameter, and bypass runtime checks, but unchecked classes do not, and thus the super call parameters are checked appropriately. This transformation is shown in figure 3.6; the super call in the constructor has been modified to route to the safe version. However, in the event this class extended an unchecked class, the super call would not be modified. Since constructors do not use virtual dispatch, this can be determined statically at compile time.

This scenario for constructors is simpler than for general methods in Java since constructors do not have virtual dispatch. Any methods which do not use virtual dispatch in Java can be simplified over the general case scenario. These methods include private methods, static methods, constructors, and super calls.

3.4.1 Checking Parameters

The same method that eases constructing tests for constructors applies to any place where method parameters need to be checked. A helper function simply encapsulates running a test on a value, while still providing that same value. As figure 3.7 shows, this follows the same basic design of the earlier test, but in a generic form that provides the same value it tests. Thus all arguments to a method, which is being tested, can be tests in line with the method call.

Figure 3.8 demonstrates how the previously described helper method is used to check the arguments to each method. This construct is generated as the body for every "non-safe" version of a method. As described in section 2.3.1, non-private methods with parameters are transformed into safe and non-safe versions, where the safe version retains the original method body, with a modified name, and is called by checked code. The unsafe method receives a method body which performs checks on the parameters, and calls the safe version if they pass. Figure 3.8 demonstrates these checks.

Providing a mechanism to check all the parameters in a single method call provides a clean transformation for method calls. However, it is also required for constructors. Since the Java Language Specification (JLS) requires that the first statement in a constructor be a call to a super constructor, or a call to another constructor in the same class, it

```
static Object testRuntimeArgument(Object runtimeValue, String staticType) {
  if (RuntimeTests.testRuntimeValue(runtimeValue, staticType)) {
    return runtimeValue;
  } else {
    RuntimeTest.runtimeTestFailure(runtimeValue, staticType);
    return runtimeValue;
  }
}
```

Figure 3.7: Helper Method for Checking Arguments To a Method Call

```
obj.method(
  (ArgumentType) RuntimeTest.testRuntimeArgument(argument,
                                                  "@NonNull ArgumentType"));
```

Figure 3.8: Using the Method Call Helper to Runtime Check a Method Call

would be impossible to place runtime tests above this call. This mechanism allows the first call to continue to be a call to another constructor, but still provide runtime tests of the arguments.

Since the return type of `RuntimeTest.testRuntimeArgument(...)` is `Object`, a cast back to the original argument type is necessary to ensure that the same method overload is chosen in the event that there is a more general method overload available. However, since this is generated code, these casts are guaranteed to be correct. They are only necessary to ensure the correct method overload is selected. Remove Java's runtime check of these casts would further reduce the overhead of this method.

3.5 Virtual Dispatch Runtime Check

The last component of the transformations required to insert the runtime checks into code is a check to determine if the runtime class of an object has been type checked by the Checker Framework or not. As described in section 2.3.3 in order to support virtual method dispatch with inheritance, where some classes in the inheritance chain could be untyped, it is necessary to know at runtime if a given object instance is an instance of a

```java
class A {
  private boolean checked_flag = true;

  public void method_maybe(Object param) {
    try {
      this.getClass().getDeclaredField("checked_flag");
      this.method_safe(param);
    } catch(NoSuchFieldException ex) {
      this.method(param);
    }
  }
}
```

Figure 3.9: Virtual method Dispatch Runtime Check

checked class or an unchecked class.

Since this runtime check is executed for every virtual method dispatch, it must be as fast as possible. The check is simple: a single bit is sufficient. This check is implemented for the purposes of the gradual type system in the Checker Framework by inserting a field into every class that is checked, and using reflection to look for the presence of that field in the test.

Figure 3.9 demonstrates the form of the runtime test. The method `getClass()` returns the actual runtime class of the object. The method `getDeclaredField()` is used to get the `checked_flag` field if it exists within the class. The `getDeclaredField()` method throws a `NoSuchFieldException` if the field is not present within the class. This method checks specifically for fields that are declared in the actual class, as opposed to inherited.

Chapter 4

Evaluation

The evaluation of the prototype considers several research questions. First, Section 4.1 evaluates the correctness of the prototype through a combination of unit tests and simulated errors in existing programs. Next, Section 4.2 evaluates performance by compiling components of several libraries with and without the gradual typing feature enabled, in order to simulate a situation where part of a program is checked, and part of a program is not checked. Performance of these configurations is measured. Finally, Section 4.3 evaluates the usability of the system by looking at the alternative options for improving type safety.

4.1 Correctness

First, this evaluation aims to demonstrate the prototype is correct. That is, a program with no type errors, should not result in any of the inserted runtime checks failing, when executed. And a program with runtime value type-errors inserted that cross the boundary between checked and unchecked portions should result in a runtime error from the inserted checks.

Specifically, since the prototype implements gradual checking within the nullness type system, the evaluation should verify that if any boundary receives a null value for a type that cannot accept a null value, an error should be generated. This boundary is defined in chapter 2.

4.1.1 Setup

Evaluations of the gradual nullness type system make use of an existing project called Daikon[1]. The Daikon project is a dynamic analysis tool to detect invariants in programs. It aims to help developers understand and reason about their code. It has previously been annotated with the Nullness type system and is type checked by the Checker Framework.

Daikon makes extensive use of an open source library called Plume-lib[2], which provides a number of useful abstractions for Java developers. Plume-lib is also annotated for the Nullness type system and checked by the Checker Framework. Plume-lib also has a number of internal utility classes including `ArraysMDE`, which is used for array operations.

Both Plume-lib and Daikon have existing comprehensive unit test suites, which can be used to exercise the libraries' code paths.

Daikon and Plume-lib can be compiled separately, and the utility classes within Plume-lib can be compiled separately from the rest of Plume-lib. Therefore, it is possible to create several artificial checked-unchecked boundaries within the project by compiling some parts with the standard Java compiler without the Checker Framework, and other parts with the gradual nullness checker enabled.

Daikon consists of 169,902 lines of Java code, and Plume-lib consists of 13,797 lines of Java code, as measured by cloc v. 1.6[3].

4.1.2 Results

This thesis includes two different assessments of correctness of the gradual nullness type system.

First, a set of twenty synthetic unit tests were developed during the development process. They validate the behaviour of the gradual nullness type system and the behaviour of the inserted runtime tests. The tests create, in isolation, a checked-unchecked boundary for each scenario described in Chapter 2, and validate that the correct runtime checks are inserted, and that the checks result in the correct runtime behaviour when the resulting program is executed.

To validate a real software scenario, experiments using Daikon are performed. As described in Section 4.1.1 Plume-lib can be compiled with gradual nullness type checking

[1] http://plse.cs.washington.edu/daikon/
[2] http://mernst.github.io/plume-lib/
[3] http://cloc.sourceforge.net/

enabled, and Daikon compiled without. The result simulates the motivating scenario where the typing is only applied to a single component or library of a larger program.

Plume-lib can also be compiled with gradual nullness enabled, but, first, one of the internal utility classes, `ArraysMDE`, can be compiled without gradual nullness. This simulates another motivating scenario, where type annotations are only gradually added to a program, leaving some classes untyped, and others typed.

Since both Daikon and Plume-lib are fully typed with the Nullness type system, they are known to be free of NPEs. Therefore, they can be used to validate that a correct program, one that would be typable if fully annotated by the Nullness type system, does not generate any type errors. The unit tests suites for the respective programs were used to exercise the code paths in each program when compiled as described above. No runtime type errors were observed during execution.

Next, the code was inspected and errors were manually inserted into the programs. The unit test suites for the respective programs were again used to exercise the code paths, where the errors were inserted, and the resulting behaviour was observed. In each case, a runtime type error was detected at the boundary, before the null value could cause a NPE within the main program.

Together, these experiments show that the gradual nullness type system performs correctly under the scenarios tested, both, in highly artificial but comprehensive tests, and in real world examples.

4.2 Performance

The evaluation aims to characterize the compile time and runtime performance overhead of the gradual checking type system. For the compiler, there is overhead of generating and type checking the new ASTs inserted into the existing program. Runtime overhead involves actually executing the runtime checks, as well as executing checks that disambiguate whether runtime checks are required when methods are dispatched virtually. These checks are described in section 2.3.3.

The aim is only to characterize the overhead. The specific performance overhead numbers will depend on the exact nature and size of the boundary, which depends on the surface of code between the checked and unchecked portions. This surface is related to how tightly coupled the two portions of the program are. Therefore, the evaluation looks at a typical example of interaction, and characterize the performance overhead.

	Plume-lib		Daikon	
	Mean	STDEV	Mean	STDEV
Gradual Nullness	386.2	6.5	703.2	29.8
Nullness	278.3	4.8	508.8	6.8
Standard Java	9.2	1.6	66.8	5.3

Table 4.1: Compilation Times (in seconds)

4.2.1 Setup

Performance is evaluated using the same two compilation configurations as used to test correctness. The performance evaluation is completed using the correct versions of Plume-lib and Daikon without errors inserted.

First, the time required for compilation with the Checker Framework gradual checking type system enabled is compared with the time required to compile with only the Checker Framework Nullness type system enabled, and with the compilation time required for the standard Java compiler with no additional checking. This experiment shows the performance overhead incurred when inserting and type checking the actual runtime check code.

Second, the execution time for each program's unit test suite is timed with and without the gradual nullness runtime checks added. No runtime errors are inserted in this experiment, and all runtime type tests have logging and diagnostic data removed from them. This characterizes the runtime overhead incurred by adding the runtime type tests.

The test machine is an Ubuntu 14.04.1 LTS virtual machine hosted on VMWare Workstation 10. The host machine has a six core AMD FX 6100 processor with 8 GB of RAM.

All timed executions were run twenty times and the results averaged, and the standard deviation computed. The timed component consists of invoking the make command required to either compile the program, or run the unit test suite. Before each run, the make clean command is run to remove all temporary and generated files. Before each set, the timed execution command is run without timing data to ensure consistency between disk cache for all timed runs.

4.2.2 Results

Table 4.1 shows the compilation times for each scenario when compiled with the gradual

	Plume-lib		Daikon	
	Mean	STDEV	Mean	STDEV
Gradual Checks	5.4	0.4	7.7	0.5
No Checks	5.0	1.5	5.7	0.7

Table 4.2: Runtime (in seconds)

nullness type system as compared with both the Nullness type system, as well as the standard Java compiler. Inserting the gradual checks adds an almost 40% performance penalty to the compiler over the Nullness type system. However, both Checker Framework type systems are many times slower than the standard Java compiler. The gradual nullness type system adds some overhead, but clearly the majority of the overhead is incurred just by using the Checker Framework. Any effort spent improving the performance of the Checker Framework would benefit the gradual nullness type system as well.

Table 4.2 gives the running times for each scenario's associated unit test suite when compiled with the gradual nullness type system as compared with no runtime checks. Since the Nullness type system has no effect on the compiled output, this time is the same as the standard Java compiler. As described earlier, the size of the runtime performance overhead is highly dependent on the size of the interface between the checked and unchecked portions of code. This difference is visible in the results, as the scenario involving a checked library (Plume-lib) with an unchecked main program (Daikon) shows a far higher performance overhead (35%) as compared with the scenario involving only a small unchecked portion, which results in a far smaller overhead (8%). With an overhead of up to 35%, the performance penalty for using a gradual type system such as gradual nullness checking is not inconsequential. However, depending on the problem domain it may well be worth it in return for the additionally type safety provided. Additionally, further research into optimizing the runtime checks, as well as using the added type information for additional optimizations may well reduce this overhead.

4.3 Usability

Usability of the system is important to justify the performance overhead incurred by the runtime checks required for gradual checking. To evaluate usability, this thesis compares gradual nullness with conservative defaults as an option for increasing type safety when annotating only parts of a program with Checker Framework type annotations.

Adopting conservative defaults for the Checker Framework means that at the boundary to unchecked code, any method that is only available in bytecode, uses a conservative defaulting policy instead of the usual policy for the type system. In the context of the Nullness type system, this means instead of the source-code default policy (NonNull except locals, NNEL), for a bytecode only method, the top type `@Nullable` is used for the return value, and the bottom type `@NonNull` is used for its parameters.

By choosing these defaults, the type system guarantees that method calls to bytecode methods are safe. Since the top type is used for the return value, any value returned is compatible with this value. And since the bottom type is used for parameters, only values compatible with all types can be used as arguments.

However, it cannot provide any type safety when a method call to checked code originates in unchecked code. When unchecked code calls checked code, there is no guarantee that the value passed is compatible with the types on the method parameters. This guarantee is provided by the runtime checks in the gradual nullness system. A small demonstration program was built to demonstrate this weakness in practise.

Additionally, since the standard defaulting policy was developed to reduce annotation effort for end developers, the conservative defaulting method may require a much higher annotation effort. To evaluate this difficulty, the Checker Framework flag to turn on conservative defaults was added to the compile target of the Plume-lib and Daikon makefiles and then the projects were built. This resulted in 1581 and 11,316 type errors identified by the Checker Framework respectively. This is an indication of the number of additional annotations or other modifications that would be required to enable conservative defaults in addition to the existing Nullness type system with NNEL defaulting policy. Conversely, the gradual nullness type system requires no additional type annotations, because the default policy uses the `@Dynamic` type, which is consistent with other types, and only signifies the need for a runtime test.

Chapter 5

Related Work

There is substantial research on both pluggable typing and gradual typing disciplines. This chapter first discusses formative work into pluggable typing in Section 5.1 along with other pluggable typing systems. Section 5.1.1 discusses work describing the features and development of the Checker Framework and evaluation of it and its original type systems. Section 5.1.2 discusses work on a number of type systems that used the Checker Framework. Next, section 5.2 discusses formative work into gradual typing, work exploring the formal underpinnings, as well as expansion of that formalization to other type systems. Finally, in section 5.3, it explores other examples of applying gradual typing techniques to other type systems.

5.1 Pluggable Typing

Bracha [4] introduces pluggable typing as an extension of previous work into optional typing. Optional typing has been used to describe different type system semantics in the past including systems where the type annotations are not checked for correctness, but have a direct effect on the runtime semantics. However, Bracha defines optional typing as requiring two features: type annotations must not be mandatory in the syntax, and types that are specified must have no effect on the runtime semantics of the language.

Once this definition is adopted as an optional typing paradigm in the language, Bracha argues this leads naturally to the idea that a type system should function as a plug-in to the language. This idea would allow any number of type systems to be implemented and applied to any given program. Bracha explains this would allow traditionally exotic and

difficult to use research projects to be integrated cleanly into a language allowing wider use. He also highlights that type annotations would function as metadata annotation, and appear on each AST node.

JavaCOP is a framework developed by Andreae et al. [1] to provide a declarative rule based language to define pluggable type systems in Java. They argue their rule based system is easier for type system designers, and more closely matches the traditional syntax directed type rules. A rule compiler is then used to compile rules into Java code that implements those rules in their type checker.

JavaCOP is based upon a custom compiler built on the existing Oracle Java Compiler. The authors modified the source code to add pluggable type system checking as an additional compiler pass. As a result, their program analysis makes use of the same modularity visible in the Checker Framework: the type checking algorithm may only see the interface to a module specified in the bytecode, without seeing the source code, and must trust the type annotations present within the interface. JavaCOP is also limited by the use of Java 1.5 annotations, which were only permitted on declarations, not all type uses.

5.1.1 Checker Framework

Papi et al. introduced the Checker Framework that implements a pluggable type system as a true plug-in to the Java compiler [15]. Built using the Java Annotation Processing API and the Java Tree API, it can plug into Oracle's OpenJDK™ Java compiler, and is invoked by any end user with a simple command line parameter. It makes use of the enhanced Java annotations defined in JSR 308 and included in Java 8 and above to annotate any type expression in a Java program.

Papi et al. [16] explored the Checker Framework in the context of the role of type system designers versus the role of end user developers. The authors' design goal for the Checker Framework was to ensure it is easy for type system designers to build simple type systems, but still possible to build powerful type systems. They describe the architecture of the Framework, how qualifiers, the type hierarchy, and type introduction rules work, and how the framework interfaces with the compiler. The authors also provide a comparison between the Checker Framework and other pluggable type systems including JavaCOP [1] and JQual [10].

Dietl et al. [5] looked into issues preventing uptake of pluggable type systems in practice. The authors believed that there were four common arguments against using additional type systems: difficulty in building a new type system, difficulty in learning how to use the new type system, effort required to annotate code, and that extra type systems provided little

benefit over testing. The authors explored these criticisms through several case studies and found difficulties in all cases to be manageable and that new bugs were in fact located.

Regarding annotation effort, the authors discovered that adding annotations to code that is well understood is simple. However, annotating foreign code that is not well understood by the developer is more difficult. As a result, developers still fear the effort required to add annotations to an existing code base.

5.1.2 Checker Framework Type Systems

Gordon et al. [9] demonstrated building a complex type and effect system using the Checker Framework to enforce thread restrictions on Graphical User Interface (GUI) method calls. The authors noted that these types of bugs are difficult to fix because they result in non-local errors and in some cases undefined behaviour. The authors also noted difficulty in annotating some foreign GUI code for which the design was not well understood.

Ernst et al. used the Checker Framework to build an information flow type system as part of a proposed high security Android app verification system [7]. The app developer would provide an information flow policy, as well as source code annotated with the information flow type system. The app store vendor could then verify that the program obeys the specified policy, and then manually evaluate if the policy is appropriate. They found that their type system required approximately 6 annotations per 100 LOC. The authors note that they treat un-verified annotations on libraries as a trusted component.

Barros et al. [3] used the Checker Framework to build type systems to assist in the analysis of implicit control flow in Android Java apps. The authors focused on two indirect control flow mechanisms: Android intents and Java reflection, and built a type system for each. Both of their type systems used annotations to apply specific types to general reflection or intent code. They used inference and data flow analysis to lower the annotation overhead for end users. Their static analysis leveraged the types to provide a more precise analysis, but maintained safety through conservative defaults.

Dietl et al. built an ownership type system modeled on Generic Universe Types [6]. The authors used type inference to lower the annotation overhead, which they saw as critical to allowing developers to adopt a new type system. However, since inference does not provide unique typing for ownership type systems, they allowed end users to tune the system using annotations, in order to highlight desirable ownership structures in the code.

Vakilian et al. [25] similarly identify annotation overhead as an obstacle for end users to adopt additional type systems within their projects. To overcome this, the authors propose

Cascade, a speculative, annotation inference, and refactoring tool for developers. They propose adding annotations as a refactoring task, and argue that too much automation, as may be present in automatic inference tools, or defaulting options, make the task more difficult for end users. The proposed tool, Cascade, helps developers identify refactorings to fix annotation type errors, the results of those refactorings, and apply those refactorings to their code.

Sampson et al. [17] used the Checker Framework to build a type system to safely utilize low power hardware that can produce lower resolution results either in computation or in storage retrieval. The authors used typing to enforce the restriction that portions of the problem that tolerate these lower resolution results (e.g., image processing) are executed on low power hardware, and the data is stored in low power memory, and portions of the program requiring precise computations (e.g., program flow) are executed in precise hardware. The evaluation involved using the Checker Framework to modify the AST tree to simulate each type of hardware. This AST modification technique is leveraged in the gradual typing prototype.

5.2 Formative Work in Gradual Typing

Siek and Taha introduced their version of gradual typing as a trade off between fully dynamic and fully static languages [20]. They motivated their work with the idea that dynamic typing is well suited to starting a project, but static typing becomes more valuable for maintaining and improving performance of a program. Thus a method of gradually migrating between the two is desirable. The authors present a formal type system that supports gradual typing, and a calculus that they call $\lambda^{?}_{\to}$ that supports functional and imperative programming paradigms.

The authors define run-time semantics for $\lambda^{?}_{\to}$ via conversion to the simply-typed lambda calculus extended with explicit runtime casts: $\lambda^{\langle\tau\rangle}_{\to}$. They prove several properties of their calculus and type system. Specifically, the authors prove that the translation does not insert casts in a fully typed program, and thus a fully typed program is equivalent to the simply-typed lambda calculus. They also prove type safety, that after translation and evaluation, the result is either a value or a cast error but never a type error.

The authors provide proofs of their theorems using an automated proof assistant. They also explain and justify the consistency relation for dynamic types. They compare this with previous work on gradual typing that proposes a subtype relation. They show their consistency relation allows safe gradual typing of structural function types.

Siek and Taha further expand their gradual typing system to support objects [19]. They introduced a new calculus $\mathbf{Ob}^{<:}$ that supports objects and proved the same properties for it, also using a proof assistant. Finally, they extended their type consistency relation to support objects and object structural subtyping, which they described as orthogonal to consistency.

Siek et al. further expand the formal basis for gradual typing to support mutable objects. This work was motivated by the need to support dynamic languages that allow members of objects to be added, removed, or their types changed at runtime. The authors also wished to address deficiencies in their earlier works which restricted reference types to be non-polymorphic. That is, a reference could be consistent with a dynamic type, but two references were not consistent with each other unless the inner types were the same type. The authors developed two different techniques to deal with mutable objects.

In the first method, the authors propose wrapping objects in a guarded reference, which is a type of proxy object. When an object member is accessed through this guarded reference, in static code, the guard creates a cast to the static type. This closes the type safety hole opened by allowing mutable objects, but at the cost of requiring checks at every read and write of object members, even within checked code. The authors note that guarded objects are not novel [22], and are analogous to how Java maintains type-safety with covariant array subtyping.

The second method introduced by the authors is through monotonic objects. Instead of guarding an object through a proxy at runtime, the type system is made more restrictive. An object with a statically typed reference, has parts of its type fixed, or made immutable. An object can be no more dynamic than the least dynamic reference to it. In this way, members can be added, but any members which are referenced by a statically typed reference cannot be removed. And any members, which have a static type assigned, cannot have their type changed at runtime. In this way, object casts cause an object to become less dynamic. Therefore, this type system is less expressive than guarded references, but avoids the overhead involved with guarding every reference, while still providing type safety.

The authors provide a formal model for both of these approaches and prove them sound.

Siek and Wadler provide a method of optimizing sequences of casts of higher order types [21]. They demonstrate that any sequence of casts, which they represent as a sequence of twosomes, source type and target type of the cast, can be coalesced into a threesome, a cast where the initial source type is cast to the greatest lower bound, that is to say, the most specific type, and then to the target type. Vitousek extends this work to optimize mutable object casts, by coalescing sequences of object guard proxies [26].

5.3 Gradual Type Systems

Gronski et al. propose a language and type system called SAGE [11]. They propose that interface specifications should evolve as a project matures, naturally from simpler specifications in an immature project to precise specifications in a mature project where more guarantees are required. They propose a language that combines the Dynamic type, with refinement types to provide this flexibility. To overcome the undecidability of refinement types, they eschew traditional compile time guarantees, and insert run time casts. They guarantee that a program only fails at these casts, and not with a type error. On a failed cast, the runtime system feeds this failure back into the compiler as it can then in future compilations make a subtype judgement in this case.

Lehtosalo and Greaves explored gradual typing of Python by considering what language features make typing difficult [14]. As a result, they aimed to create a pluggable gradual typing system in a toy language called Alore, which is similar to Java. By restricting some features of Python and using nominal typing they simplified the task of building a gradual typing system. Their approach uses guards in a similar manner as checks are used in this work. Their work presents a prototype and a formalization of their type system in $\mathbf{FJ}^{?}$, an extension of featherweight Java.

The authors also felt that allowing a gradual type system to result in a type error changed the semantics of a language. In order to achieve a type system that does not affect the semantics of the language, they sought only to report the error, and allow the program to continue. The Checker Frameworks gradual typing extensions presented here allow this as an option for type system designers.

Ina and Igarashi explored gradual typing of featherweight Java (FJ) [13]. The authors called their new formalism $\mathbf{FJ}^{?}$ and later expanded it to support generics. They enhance **FJ** with a ? type and describe a translation from $\mathbf{FJ}^{?}$ to $\mathbf{FJ}_{refl}$ a version of **FJ** that uses reflection to look up members of dynamic types. This approach allows execution to proceed without binding method names until runtime.

The authors also discuss expanding their work to support dynamic types in type parameters, and provide a formal treatment of their expanded language. In the gradual nullness system provided here, the nature of the static dynamic boundary eliminates this question. Type bounds are erased at compile time, so checking them does not make sense. And when a type parameter is used as the type for a value that crosses the checked-unchecked boundary it is checked at that point, as any other value would.

Sergey and Clarke developed a gradual ownership type system for a Java-like language [18]. They wanted to reduce the annotation burden for developers wishing to use ownership

types. The authors acknowledge that annotation burden impedes the use of ownership types, and that reducing the annotation burden is complicated by the fact that trivial ownership typing is present for all programs. They discuss the consistency relation; the nature of their runtime checks and their expense; and the nature of the static-dynamic boundary in their system. The authors note that their system does not account for code which is unchecked. They provide a formalism and proofs of the properties of their system, as well as a prototype. They evaluated their prototype using components from the Java Collection Framework.

Thiemann and Fennell develop a formal gradual typing system for annotated type systems [23]. While focusing on functional languages, they propose an approach to add gradual typing to existing generic annotated type systems. They define the types of properties for which their system applies. They also provide motivating examples, including an interesting suggestion that gradual types can be used to model problems that are traditionally the domain of dependent types. The authors develop a formal system and prove several properties about it.

Schwerter et al. develop a gradual-effect system based on a generic-effect framework that models effects with a set of privileges, which represent the allowable effects during evaluation of an expression and predicates which check that an expression has the correct permissions [2]. The authors note difficulty in implementing practical fully-static effect-systems, citing JavaUI's inability to deal with dynamic behaviour as an example. The authors argue for a gradual effect system to alleviate this issue. They introduce the notion of an unknown permission, and adapt Siek and Taha's type-consistency relation to the concept of sets of permissions. They develop a formal system to show the correctness of their framework, based upon the simply typed lambda calculus.

Herman and Flanagan describe work by the Ecma TC39-TG1 working group on the next generation ECMAScript, more commonly known as Javascript, specification [12]. The authors describe the features of the new language, including the use of gradual typing to provide a bridge between newer code for which typing is a desired feature, and existing legacy dynamically-typed code.

Tobin-Hochstadt and Felleisen provide a design for a typed version of Scheme [24]. Their design incorporates typing features that they felt were necessary for idiomatic Scheme, based on the types of contracts that Scheme programmers already write. They discuss the difficulties that Scheme induce in adding typing. These difficulties include how to support typing the powerful macro expansion facilities present in Scheme, which allows end developers to define new syntactic forms. They provide a formal lambda calculus for their type system. They also discuss the motivation for this work, in the context of the original

purpose for dynamically-typed languages, and the apparent limitations on maintenance when developing larger, longer term projects in these languages. This realization leads them to the necessity to support interoperating with untyped code. The authors adopt a module-based typed-untyped boundary similar to the one used in this work, and discuss how runtime checks can be used to ensure safety across this boundary.

Fennell and Thiemann propose a gradually typed security type system [8]. They provide an ML core language with references with a gradually annotated security type system. The authors prove the system meets their information-flow security requirements both statically and at runtime. They are motivated by the observation that often security is introduced as a project requirement later, when it is more difficult to redesign a project to use a static security type system. Additionally, they note that security requirements or policies may change over time.

Chapter 6

Conclusion

This thesis aims to provide an alternate method for ensuring type safety of checked code in the presence of unchecked code within a pluggable type system using gradual checking techniques. Specifically, this problem is motivated by envisioned use cases for Checker Framework type systems in which a type system is gradually deployed to an existing project, and where a main project or a library has a type system in use, but the library or main project being linked too does not.

This thesis contributes a prototype for a gradual pluggable typing framework, in addition to a demonstration type system, the gradual nullness type system. Background material on the gradual typing, and on pluggable typing including the design of the Checker Framework is presented first. Limitations of the Checker Framework and motivating examples provide a direction to the work. The specific theoretical challenges posed by implementing gradual pluggable typing using the Checker Framework in an object oriented, inheritance based language like Java are discussed, and techniques for overcoming them presented. The implementation details of building the previously discussed techniques to implement gradual nullness typing using the Checker Framework and the OpenJDK™ implementation of the Java language compiler are presented. An evaluation of the implementation considers the correctness, performance overhead, and applicability to the motivating limitations of the Checker Framework. Finally, related works in the fields of pluggable typing and gradual typing are discussed, as they pertain to this work, specifically contrasting their results with this work, and supporting the motivating problems.

The evaluation demonstrated the correctness of the prototype using both synthetic tests, and errors inserted into real world software. It demonstrated the effectiveness of gradual typing for increasing type safety in partially annotated software, even when com-

pared with conservative defaulting. It also demonstrated lowering the annotation overhead compared with other options for partially typing programs. This demonstrated that the motivating examples, gradually adding annotations to a software package, and having only some components of a program (for example a library) annotated are satisfied by gradual pluggable typing. Finally the evaluation looked at the performance overhead in real world programs, again modeled on the motivating examples, and showed that both runtime overhead as well as compile time overhead may be acceptable in return for the benefits demonstrated.

6.1 Future Work

As future work, there may be value in more formally proving some type safety properties of the type system. While several papers have proved formal properties of similar or related type systems, proving this one could provide the theoretical basis needed to make optimizations safe. As one of the goals of gradual typing is to enable the static type information to improve performance, the framework could be extended to allow type system designers to affect code generation so as to allow them to use the static type information to optimize the resulting code. In addition, improving the performance of the runtime checks is worth additional work. Several options described in Section 2.2.1 should be evaluated to determine how much if any performance benefits they provide.

The framework developed here should be tested with additional type systems, to validate its usefulness for developing general gradual pluggable type systems. This validation should include investigation into what kinds of effect systems may be modeled, and how to construct runtime tests for these type systems. Finally, while compile time overhead of the gradual checking framework is not substantial, the Checker Framework incurs a substantial performance overhead. Improving compile time performance of the Checker Framework would benefit all type systems built on it, including gradual type systems.

Some future work on the prototype itself was outlined in this thesis and included more flexible options for implementing runtime test. Another major feature which is not explored here is handling multiple type systems. The most straight forward method would to enhance the name mangling of methods and the checked field to include the type system in question. This would allow multiple type systems to be applied to code, but would result in a combinatorial explosion in the number of methods. A second option would be to enhance the runtime test to allow multiple type systems to execute tests on a value within each test, this would eliminate the problem of growth in the number of methods, but would

require more careful optimization and design. And more exploration of the formal type safety implications of allowing multiple type systems at once is also warranted.

References

[1] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *ACM SIGPLAN Notices*, volume 41, pages 57–74. ACM, 2006.

[2] Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 283–295. ACM, 2014.

[3] Paulo Barros, Rene Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo dAmorim, and Michael D. Ernst. Static analysis of implicit control flow: Resolving Java reflection and Android intents. In *Automated Software Engineering (ASE)*, pages 669–679, Nov 2015.

[4] Gilad Bracha. Pluggable type systems. In *OOPSLA workshop on revival of dynamic languages*, volume 1, 2004.

[5] 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*, pages 681–690. ACM, 2011.

[6] Werner Dietl, Michael D. Ernst, and Peter Müller. Tunable static inference for generic universe types. In *ECOOP 2011–Object-Oriented Programming*, pages 333–357. Springer, 2011.

[7] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1092–1104, New York, NY, USA, 2014. ACM.

[8] Luminous Fennell and Peter Thiemann. Gradual security typing with references. In *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*, pages 224–239. IEEE, 2013.

[9] Colin S Gordon, Werner Dietl, Michael D Ernst, and Dan Grossman. Java UI: effects for controlling UI object access. In *ECOOP 2013–Object-Oriented Programming*, pages 179–204. Springer, 2013.

[10] David Greenfieldboyce and Jeffrey S Foster. Type qualifier inference for java. In *ACM SIGPLAN Notices*, volume 42, pages 321–336. ACM, 2007.

[11] Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

[12] David Herman and Cormac Flanagan. Status report: specifying JavaScript with ML. In *Proceedings of the 2007 workshop on Workshop on ML*, pages 47–52. ACM, 2007.

[13] Lintaro Ina and Atsushi Igarashi. Towards gradual typing for generics. In *Proceedings for the 1st workshop on Script to Program Evolution*, pages 17–29. ACM, 2009.

[14] Jukka Lehtosalo and David J Greaves. Language with a pluggable type system and optional runtime monitoring of type errors. In *Proceedings of International Workshop on Scripts to Programs (STOP)*, 2011.

[15] Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Pluggable type-checking for custom type qualifiers in java. 2007.

[16] 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*, pages 201–212. ACM, 2008.

[17] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011.

[18] Ilya Sergey and Dave Clarke. Gradual ownership types. In *Programming Languages and Systems*, pages 579–599. Springer, 2012.

[19] Jeremy Siek and Walid Taha. Gradual typing for objects. In *ECOOP 2007–Object-Oriented Programming*, pages 2–27. Springer, 2007.

[20] Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

[21] Jeremy G Siek and Philip Wadler. Threesomes, with and without blame. In *ACM Sigplan Notices*, volume 45, pages 365–376. ACM, 2010.

[22] T Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: run-time support for reasonable interposition. In *ACM SIGPLAN Notices*, volume 47, pages 943–962. ACM, 2012.

[23] Peter Thiemann and Luminous Fennell. Gradual typing for annotated type systems. In *Programming Languages and Systems*, pages 47–66. Springer, 2014.

[24] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. *ACM SIGPLAN Notices*, 43(1):395–406, 2008.

[25] Mohsen Vakilian, Amarin Phaosawasdi, Michael D Ernst, and Ralph E Johnson. Cascade: A universal type qualifier inference tool. 2014.

[26] Michael M Vitousek. Gradual typing with efficient object casts. 2012.