NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

MSc THESIS

A Study of Typing-Related Bugs in JVM compilers

Stefanos A. Chaliasos

Supervisors: **Alex Delis,** Professor NKUA
Dimitris Mitropoulos, Assistant Professor NKUA

ATHENS

JULY 2021

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Μελέτη Σφαλμάτων Σχετικά με Τύπους στους Μεταγλωττιστές των JVM Γλωσσών Προγραμματισμού

Στέφανος Α. Χαλιάσος

Επιβλέποντες: **Αλέξανδρος Δελής,** Καθηγητής ΕΚΠΑ
Δημήτρης Μητρόπουλος, Αναπληρωτής Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

ΙΟΥΛΙΟΣ 2021

MSc THESIS

A Study of Typing-Related Bugs in JVM compilers

Stefanos A. Chaliasos
S.N.: CS3190004

SUPERVISORS: **Alex Delis,** Professor NKUA
Dimitris Mitropoulos, Assistant Professor NKUA

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Μελέτη Σφαλμάτων Σχετικά με Τύπους στους Μεταγλωττιστές των JVM Γλωσσών Προγραμματισμού

Στέφανος Α. Χαλιάσος

Α.Μ.: CS3190004

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Αλέξανδρος Δελής,** Καθηγητής ΕΚΠΑ
Δημήτρης Μητρόπουλος, Αναπληρωτής Καθηγητής ΕΚΠΑ

ABSTRACT

Compiler testing is a prevalent research topic that has gained much attention in the past decade. Researchers have mainly focused on detecting compiler crashes and miscompilations caused by bugs in the implementation of compiler optimizations. Surprisingly, this growing body of work neglects other compiler components, most notably the front-end. In statically-typed programming languages with rich and expressive type systems and modern features, such as type inference or a mix of object-oriented with functional programming features, the process of static typing in compiler front-ends is complicated by a high-density of bugs. Such bugs can lead to the acceptance of incorrect programs, the rejection of correct programs, and the reporting of misleading errors and warnings.

In this thesis, we undertake the first ever effort to the best of our knowledge to empirically investigate and characterize typing-related compiler bugs. To do so, we manually study 320 typing-related bugs (along with their fixes and test cases) that are randomly sampled from four mainstream JVM languages, namely Java, Scala, Kotlin, and Groovy. We evaluate each bug in terms of several aspects, including their symptom, root cause, bug fix's size, and the characteristics of the bug-revealing test cases.

Finally, we implement a tool for finding front-end compiler bugs in Groovy and Kotlin compilers by exploiting the findings of our thesis.

SUBJECT AREA: Programming Languages/Software Testing

KEYWORDS: Compiler bugs, Compiler testing, Static typing, Java, Scala, Kotlin, Groovy

ΠΕΡΙΛΗΨΗ

Ο έλεγχος των μεταγλωττιστών είναι ένα ερευνητικό πεδίο το οποίο έχει τραβήξει το ενδιαφέρον των ερευνητών την τελευταία δεκαετία. Οι ερευνητές έχουν κυρίως επικεντρωθεί στο να βρουν σφάλματα λογισμικού που τερματίζουν τους μεταγλωττιστές, και εσφαλμένες μεταγλωττίσεις προγραμμάτων οι οποίες οφείλονται σε σφάλματα κατά της φάσης των βελτιστοποιήσεων. Παραδόξως, αυτό το αυξανόμενο σώμα εργασίας παραμελεί άλλες φάσεις του μεταγλωττιστή, με την πιο σημαντική να είναι η μπροστινή πλευρά των μεταγλωττιστών. Σε γλώσσες προγραμματισμού με στατικό σύστημα τύπων που προσφέρουν πλούσιο και εκφραστικό σύστημα τύπων και μοντέρνα χαρακτηριστικά, όπως αυτοματοποιημένα συμπεράσματα τύπων, ή ένα μείγμα από αντικειμενοστραφείς και συναρτησιακά χαρακτηριστικά, ο έλεγχος σχετικά με τους τύπους στο μπροστινό μέρος των μεταγλωττιστών είναι περίπλοκο και περιέχει αρκετά σφάλματα. Τέτοια σφάλματα μπορεί να οδηγήσουν στην αποδοχή εσφαλμένων προγραμμάτων, στην απόρριψη σωστών προγραμμάτων, και στην αναφορά παραπλανητικών σφαλμάτων και προειδοποιήσεων.

Πραγματοποιούμε την πρώτη εμπειρική ανάλυση για την κατανόηση και την κατηγοριοποίηση σφαλμάτων σχετικά με τους τύπους στους μεταγλωττιστές. Για να το κάνουμε αυτό, μελετήσαμε 320 σφάλματα που σχετίζονται με την διαχείριση τύπων (μαζί με τις διορθώσεις και τους ελέγχους τους), τα οποία τα συλλέξαμε με τυχαία δειγματοληψία από τέσσερις δημοφιλής JVM γλώσσες προγραμματισμού, την Java, την Scala, την Kotlin, και την Groovy. Αξιολογήσαμε κάθε σφάλμα με βάση διάφορες πτυχές του, συμπεριλαμβανομένου του συμπτώματος του, της αιτίας που το προκάλεσε, της λύσης του, και των χαρακτηριστικών του προγράμματος που το αποκάλυψε.

Τέλος υλοποιήσαμε ένα εργαλείο το οποίο χρησιμοποιεί τα ευρήματα μας ώστε να βρει με αυτοματοποιημένο τρόπο σφάλματα στο μπροστινό μέρος των μεταγλωττιστών της Kotlin και της Groovy.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Γλώσσες Προγραμματισμού/Επαλήθευση Λογισμικού

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Σφάλματα Μεταγλωττιστών, Επαλήθευση Μεταγλωττιστών, Στατικό σύστημα τύπων, Java, Scala, Kotlin, Groovy

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dimitris Mitropoulos, for his support and guidance throughout the years. Dimitris is the one who motivated me to focus on research and the one who encouraged me during all failures.

It was a pleasant experience to work closely with Thodoris Sotiropoulos. Thodoris brought me into the world of software testing. I am grateful for his expertise, guidance, and the personal friendship over the years.

CONTENTS

1. INTRODUCTION 15

1.1 Research Questions . . . 15

1.2 Languages selection . . . 16

1.3 Contributions . . . 16

1.4 Summary of findings . . . 17

1.5 Proof-of-concept program generator . . . 17

2. BACKGROUND AND RELATED WORK 18

2.1 Software testing . . . 18

2.2 Compilers . . . 18

2.3 Compilers Testing . . . 19

2.4 Examined languages . . . 20
2.4.1 Java . . . 21
2.4.2 Scala . . . 21
2.4.3 Kotlin . . . 22
2.4.4 Groovy . . . 22

2.5 Non-JVM Compiler Bugs and Prior Studies . . . 23
2.5.1 Understanding compiler bugs . . . 23
2.5.2 Other bug studies . . . 23

3. METHODOLOGY 25

3.1 Collecting bugs and fixes . . . 25
3.1.1 Collecting Java Bugs . . . 26
3.1.2 Collecting Scala Bugs . . . 26
3.1.3 Collecting Kotlin Bugs . . . 27
3.1.4 Collecting Groovy Bugs . . . 27

3.2 Analyzing bugs . . . 27

3.3 Threats to validity . . . 28

4. BUG STUDY 29

4.1 RQ1: Symptoms . . . 29
4.1.1 Unexpected Compile-Time Error . . . 29
4.1.2 Internal Compiler Error . . . 30
4.1.3 Unexpected Runtime Behavior . . . 31

4.1.4 Misleading Report 32
4.1.5 Compilation Performance Issue 33
4.1.6 Comparative Analysis 34

4.2 RQ2: Bug Causes 34
4.2.1 Type-related Bugs 35
4.2.2 Semantic Analysis Bugs 36
4.2.3 Resolution Bugs 37
4.2.4 Bugs Related to Error Handling and Reporting 38
4.2.5 AST Transformation Bugs 39
4.2.6 Comparative Analysis 40

4.3 RQ3: Bug Fixes 40
4.3.1 How Bugs are Introduced? 40
4.3.2 Size of Bug Fixes 41
4.3.3 Duration of Bugs 41
4.3.4 Comparative Analysis 43

4.4 RQ4: Test Case Characteristics 43
4.4.1 General Statistics 43
4.4.2 Language Features 44
4.4.3 Comparative Analysis 46

5. IMPLICATIONS AND DISCUSSION 47

6. A PROOF-OF-CONCEPT PROGRAM GENERATOR 49

7. CONCLUSIONS AND FUTURE WORK 51

ABBREVIATIONS - ACRONYMS 52

REFERENCES 56

LIST OF FIGURES

4.1 The distribution of symptoms. 29
4.2 The distribution of bug causes. 34
4.3 Size of bug fixes. 42
4.4 Cumulative distribution of bugs through time. 43
4.5 The classification of the language features that appear in test cases, along with their frequency. For each category, we show the four most frequent features. 44

LIST OF TABLES

3.1 Statistics on bug collection. 26

4.1 General statistics on test case characteristics. 44
4.2 The five most frequent and the five least frequent features supported by all studied languages. 45
4.3 The five most bug-triggering features per language. 46

6.1 Summary of the bugs found by our proof-of-concept tool. In total, we have found 28 bugs in `kotlinc` and `groovyc`, of which, 16 have been fixed by developers. 50

LISTINGS

2.1 A Scala program with a higher-kinded type. 21
2.2 A Kotlin program with a nullable type. 22
4.1 KT-10711: A program that triggers a `kotlinc` bug 29
4.2 GROOVY-7618: A program that triggers a `groovyc` bug with an *internal compiler error*. 30
4.3 JDK-7041019: A program that triggers a `javac` bug with an *unexpected runtime behavior*. 32
4.4 KT-5511: A program that triggers a `kotlinc` bug with a *misleading report*. . 32
4.5 Dotty-10217: A program that triggers a Dotty bug with an *compilation performance issue*. 33
4.6 KT-9630: A Kotlin program that triggers a bug related to an *incorrect type transformation*. 35
4.7 JDK-8039214: A Java program that triggers a bug related to *incorrect type comparisons*. 36
4.8 Scala2-5878: A Scala program that triggers a bug related to *missing validation checks*. 37
4.9 JDK-7042566: A Java program that triggers a *resolution* bug. 38
4.10 Scala2-6714: A Scala program that triggers an *AST transformation bug*. . . 39

PREFACE

Part of the work presented in this thesis has been done in collaboration with Thodoris Sotiropoulos, Charalambos Mitropoulos, George Drosos, Diomidis Spinellis, and Dimitris Mitropoulos.

1. INTRODUCTION

Over the past decade, we have witnessed tremendous advances in compiler reliability improvement techniques. Dozens of techniques and methods have emerged to validate compilers' correctness or facilitate compiler testing and debugging: from program generators [49, 36, 37, 28] and transformation-based techniques [24, 50, 25, 42], to test case reduction [40] and test case prioritization approaches [7, 8]. Although the initial focus was on C/C++ compilers, researchers have also invested much effort on testing other compilers [27, 13, 15], runtime systems [11, 10], or even dynamic programming languages [45, 20, 39]. This exciting research work has led to the discovery and fixing of thousands of bugs in industrial-strength compilers, and has assisted compiler developers in preventing crashes and miscompilations (i.e., generation of incorrect machine instructions) from happening.

Most of the proposed techniques though, focus on finding bugs in optimizing compilers. For example, Nagal et al. [36, 37] craft C programs that exercise optimizations on arithmetic expressions. Another example is the most recent program generator for C/C++ programs [28], which adopts a set of program generation policies that are tailored to triggering specific buggy optimizations.

We find it surprising that this growing body of work currently neglects other compiler components, most notably the front-end. The compiler front-end is responsible for performing 1) the source code's lexical analysis and parsing, and 2) a set of semantic analyses that verifies whether the input code is error-free and respects the semantics of the language. In statically-typed languages with 1) rich and expressive type systems that rely on complex type theories (e.g., *higher-kinded types* [35], parametric polymorphism, or path-dependent types [3]), and 2) modern features (e.g., type inference, mix of object-oriented with functional programming), the implementation of front-ends (and especially the task of typing programs) has become particularly complex exhibiting a high density of bugs. For example, at the time of writing, the type checker of the Scala 2 compiler (*typer*) is the component that suffers from the most bugs (see scala/bug). Bugs in the implementation of front-end's semantic analyses and typing algorithms can potentially affect the ability of the compiler to effectively deal with certain programs leading to type-safety breaches, and allowing the compilation of non-portable code, or propagating themselves to other compiler phases.

1.1 Research Questions

In this work, we conduct the first quantitative and qualitative study of the characteristics of typing-related compiler bugs. Specifically, we aim to understand their manifestations, their nature, and obtain insights into how these bugs are introduced, triggered, and fixed. Specifically, our study seeks answers to the following research questions.

1. **(Symptoms) What are the main symptoms of typing-related compiler bugs?** What is the frequency of these symptoms? (Section 4.1)
2. **(Bug Causes) What are the categories into which we can group typing-related bugs based on their root cause?** What is the frequency of these categories? (Section 4.2)

3. **(Bug Fixes) How are typing-related compiler bugs introduced?** What is the size of their fixes? How long time does it take to fix these bugs? (Section 4.3)
4. **(Test Case Characteristics) What are the main characteristics of the bug-revealing test cases?** What language features are prevalent in these test cases? (Section 4.4)

1.2 Languages selection

To answer the aforementioned research questions, we examine bugs from the compilers of four mainstream JVM programming languages, namely Java, Scala, Kotlin, and Groovy. All these languages are statically-typed object-oriented languages, feature a nominal type system, and support parametric polymorphism by using the Java generics framework [5]. Beyond that, they support (some to a lesser or greater extent) functional programming features, while they also adopt some sort of type inference. Java is in the list of the most widely-used and popular programming languages [18, 44]. The Scala programming language [38] is a research product that unifies the object-oriented and functional paradigms. One of the strengths of Scala is its type system, which offers higher-kinded types, implicits, and structural types. Regarding Kotlin: although it is quite a new language (it first appeared in 2011), it has gained much popularity recently. It is now Google's preferred programming language for building Android applications [32]. Finally, Groovy is a popular programming language [44] that supports both dynamic and static typing, and also provides flow-sensitive typing.

Using carefully-crafted search criteria and some heuristics, we obtain 4,101 previously reported typing-related bugs taken from the issue trackers of the studied languages. We apply our method on a random sample of 320 bugs. We study each bug report of this sample, along with the accompanying developers' discussion, bug fix and test case, and we finally evaluate every bug in terms of several aspects including, its symptom, its root cause, and its test case's characteristics.

1.3 Contributions

Our work makes the following contributions:

- We present a method for collecting and assessing typing-related compiler bugs, and provide a corresponding reference dataset consisting of bugs taken from popular JVM compilers (Section 3).
- By examining 320 typing-related bugs, we provide an in-depth analysis on diverse aspects, including bug symptoms, root causes, bug fixes, and test case characteristics (Section 4).
- We enumerate the implications of our findings, and discuss potential futures directions on compiler testing (Section 5).
- We demonstrate the leverage obtained from our work's findings through the design and implementation of a proof-of-concept Kotlin and Groovy test-program generator (Section 6.

1.4 Summary of findings

Some of our representative findings are: 1) most of typing-related bugs (50.94%) manifest as unexpected compile-time errors: the buggy compiler mistakenly rejects correct programs, 2) the majority of typing-related bugs (40.31%) lie in the implementations of the underlying type systems and in other core components related to operations on types (e.g., type inference, subtyping rules), 3) although typing-related bugs are typically fixed without requiring extensive modifications in compilers' code base, developers take a few months to resolve a bug, 4) parametric polymorphism is the most pervasive feature in the bug-revealing test cases: 57.19% of the bug-revealing test cases involve parametric polymorphism-related features, e.g., declaration of a parameterized function / class or use of a parameterized type.

1.5 Proof-of-concept program generator

To demonstrate the practicality of our study, we leverage some of our observations to design and implement a proof-of-concept program generator for testing the Koltin and Groovy compilers' front-end. Our program generator was able to find 28 previously unknown bugs within two months of testing. More than a half (16 / 28) of the reported bugs have already been fixed. We do believe that our study can help researchers to build appropriate testing techniques or adapt the existing ones for a more holistic testing of compilers.

2. BACKGROUND AND RELATED WORK

In this chapter, we start with discussing the general practice of software testing and then extend to compilers' testing. Next, we present the essential elements of the programming languages examined. The final section covers the related work.

2.1 Software testing

Software testing is an essential part of the software development process. It aims to detect deficits of software under test. Software testing evaluates the software against various properties. If testing does not find any issues, the software under test is considered high quality. In practice, software testing is conducted through bug finding. In general, test oracles other than requirements/specifications can be used to detect software bugs.

Software testing is as old as software development. Through the years, many methodologies have emerged around it. A few more well-known software testing practices are [48]:

- Functional testing: Functional testing tries to verify a specification or functionalities of the code.
- White-box testing vs Black-box testing vs Gray-box testing: Black-box testing verifies and validates the software without any knowledge of the internal implementation. White-box testing verifies and validates the software with knowledge of the internals of an application. Gray-box testing applies black-box level tests with knowledge of the internal implementation.
- Unit testing vs Integration testing: Unit testing verifies the functionality of an encapsulated component in the software under testing. Integration testing is a follow-up to component testing. It tries to prove the interfaces between components, making sure the integrated product works according to a specification.
- Regression testing: Regression testing focuses on finding software regressions, where previously working software functionality stops working due to recent changes.
- Security testing: Security testing focuses on finding security issues that enable system intrusion by hackers.

2.2 Compilers

A compiler is a program that transforms source code written in high-level programming languages into low-level representations [1]. The low-level representations can be in the form of assembly language, machine code, or byte-code (e.g., JVM bytecode). The primary functionality of a compiler is to translate programs written in more human-friendly languages into programs that are ready for execution.

The functionality of a compiler includes (but is not limited to):

- Validating the syntactical correctness of the programs, and optionally providing feedbacks to users.

- Generating correct and efficient low-level code through translations and optimizations.
- Producing the low-level code according to specifications of assemblers, linkers, or virtual machines.

A compiler typically consists of three main components: the front-end, the middle-end, and the back-end. A compilation undergoes the aforementioned three phases. First, the front-end checks whether the language syntax of the program is correct. Semantic errors, such as divide-b-yzero , could also be checked if possible. Third, type checking is performed statically. Second, the front-end generates an *intermediate representation* (IR) of the source code for processing by the middle-end. For optimizing compilers, the middle-end is where most of the code optimization takes place. Finally, the back-end is responsible for translating the IR produced by the middle-end into low-level code. The output from the back-end is typically a program in an assembly language which is then translated into machine code by an assembler.

2.3 Compilers Testing

Testing compilers is a topic that has attracted many researchers and practitioners over the past decade. During this time period, plenty of testing techniques have emerged to stress-test compilers by either detecting compiler crashes, or miscompilations caused by bugs in the implementation of compiler optimizations or code generation. Although compilers are considered very reliable, the proposed techniques and tools have managed to detect thousands of bugs in various, industrial-length compilers. Below we discuss the main dimensions of compiler testing. more details about compiler testing can be found in the survey of Chen et al. [9].

Finding compiler crashes. A compiler crash (or internal compiler error) is an unexpected error (exception) that happens during compilation, which prevents the compiler from generating the target program or producing an informative error message (in case the input program is grammatically or semantically incorrect). These crashes are typically caused by bugs that occur in various compiler phases: from lexical analysis to code generation.

Detecting compiler crashes is done through a straightforward manner. There is typically a program generator (or a mutation-based fuzzer [45, 39]) producing arbitrary programs (written in the corresponding source language), which are then given to the compiler under test. Compiler correctness is trivially checked by examining the compiler output for crashes. Note that these program generators are not necessarily interested in producing semantically correct programs, as a crash is a clear indication of a bug and should not happen regardless of whether the input program is legal or not. For example, Superion [45] and Fuzzball [2] generate programs that are far from being correct.

Finding miscompilations. Another kind of compiler bugs is miscompilations. A miscompilation happens, when the compiler emits incorrect target code which does not respect the semantics of the original source program. Miscompilations are usually caused by bugs in compilation optimizations. Compiler optimizations is a series of semantics-preserving transformations that are applied to an intermediate language in order to boost the performance of final code written in target language. Bugs in optimizations often break the equivalence between the initial and transformed program, and lead to miscompilations.

Unlike crashes, miscompilations are subtle and difficult to detect, because the compiler does not raise any compile-time error or any other indication of erroneous behavior during compilation. Worse, due to the test oracle problem [47], we cannot determine whether the code generated by the compiler is the expected one, or is equivalent with the original source program. To address the oracle problem and find miscompilations, there are two popular testing techniques, which have been extensively used: *differential testing* and *equivalence modulo inputs (EMI)*.

Differential testing [33] is a generally-applicable technique, which is suitable for testing equivalent compiler implementations that follow the same language specification. In this context, a semantically valid program P is fed to two equivalent compilers, which finally produce two executables T_1 and T_2. After running T_1 and T_2 against a specific input, we compare the results of executables. A mismatch in the results of T_1 and T_2 indicates that there is at least one compiler that generate an executable that does not follow the semantics of P.

EMI [24] addresses cases where there are no equivalent compiler implementations. EMI works in the following steps. A semantically correct program P is given to a compiler, which in turn generates the corresponding executable T. EMI then executes T against a specific input I and (1) collects coverage information stemming from the execution T, and (2) tracks the result of T, namely O. As a next step, EMI mutates the initial program P and generates a new program P' by deleting those statements of P that are dead based on input I. The compiler under test then takes the mutated program P' and produces T', which is ultimately run against the same input I. Since P' and P contain the same live statements (with respect to input I), their corresponding executables T and T' should produce the same result O in response to the same input I. When this is not a case, EMI reports a miscompilation.

Differential testing and EMI are used in conjunction with program generators, or they are applied to existing programs taken from compilers' test suites [49, 11, 10, 24, 27]. However, as differential testing and EMI test the validity of a compiler by running the resulting executables, the program generators should produce compilable programs.

Finding front-end-related bugs. There is currently a research gap in automated testing of compiler front-ends (e.g., implementation of type systems, semantic analyses, etc.), and there are very few testing methods focusing on this particular component of compilers. [13, 41]. The aim of the remainder paper is to study compiler front-ends and see what are the main types of bugs that mainstream compilers suffer from, how these bugs manifest themselves, how they are introduced, and what are the characteristics of the input programs that trigger these bugs. Building upon the findings of our study, future researchers can design appropriate testing techniques.

2.4 Examined languages

In this study, we examine the compilers of four mainstream JVM programming languages, namely Java, Scala, Kotlin, and Groovy. All these languages are statically-typed object-oriented languages, their type system is nominal, while they all support parametric polymorphism by using the Java generics framework [5]. They also support (some to a less extend, but others to a greater extend) functional programming features, including higher-order functions, function closures, or function composition. Finally, all the examined languages adopt some sort of type inference, e.g., they allow variable declarations with omit-

Listing 2.1: A Scala program with a higher-kinded type.

```
class A[X[_], T] {
  def m(x: X[T]): T = ???
}
...
val obj = new A[List, String]()
val x: String = obj.m(List("1", "2", "3"))
```

ted types, instantiations of type constructors with omitted type arguments, or function declarations with omitted return types.

2.4.1 Java

Java is in the list of the most widely-used and popular programming languages [18, 44]. Due to high competition in the JVM ecosystem, Java is constantly evolving. Starting from Java 8 where functional programming features (e.g., lambdas, higher-order functions) were added to the language, the Java team is announcing more and more releases (in short time period) that make a lot of improvements to the language. For example, Java 9 added support for modules, Java 10 introduced the `var` keyword that allows developers to omit the type of a local variable, Java 14 enhanced the `instanceof` keyword with pattern matching support [17], and more.

The Java team is also working on experimental projects in the Java Development Kit (JDK) that investigate the integration of future Java features. For example, as an addition to primitive and reference types, the project Valhalla [6] is working on supporting value types (e.g., objects represented as values). This feature will come with many updates in both runtime and type system.

2.4.2 Scala

The Scala programming language [38] is a research product that first appeared in 2004. Scala unifies the object-oriented paradigm with functional programming: beyond standard object-oriented features, other features, which are typically seen in functional languages (e.g., algebraic data types, pattern matching), are supported inherently. One of the strengths of the language is its type system that offers some sophisticated features, such as higher-kinded types [35], path-dependent types [3], implicits, or even structural types.

As an example, consider higher-kinded types. Higher-kinded types offer an extra layer of abstraction over type constructors. Listing 2.1 shows a small Scala program that defines a higher-kinded type, i.e., a type constructor that receives another type constructor as type parameter. On lines 1–3, the code defines a type constructor named `A` that takes two type parameters. The first type parameter is a type constructor (i.e., denoted as `X[_]`), while the second one is a simple type variable named `T`. The given type constructor `X` can be ultimately used to instantiate other types. For example in line 2, the method `m` receives a variable of type `X[T]` that comes from the application of type constructor `X` to the type argument `T`.

At the time of writing, the stable version of Scala is 2.13.5. However, Scala 3 along with

Listing 2.2: A Kotlin program with a nullable type.

```
val y: String? = null
// compile-time error, y is nullable
y.length
y?.length // OK, returns null
```

its corresponding compiler (i.e., Dotty), which initially begun as a experimental project, are approaching. Scala 3 involves major enhancements and additions compared to Scala 2, including contextual abstractions, union and intersection types, metaprogramming features, and other.

2.4.3 Kotlin

The Kotlin programming language is developed by the JetBrains team. Although it is a quite new language (first appeared in 2011), it has gained much popularity recently. It is now the Google's preferred programming language for building Android applications [32].

Kotlin provides some distinct features. For example, its type system guarantees null safety: Kotlin types do not hold `null` values by default, and there is a special kind of types called nullable types for storing `null` values. Nullable types also come with dedicated programming constructs for performing safe property accesses on variables that may point to `null`.

Consider Listing 2.2 where we define a nullable variable whose type is `String?`. This type indicates that the variable can take any string value, or `null`. Attribute accesses of the form `x.f` are not allowed by the type system, when the type of the receiver is nullable (lines 2, 3). Instead, attribute accesses on nullable variables are done through the `?.` operator (line 4), which returns `null` if the receiver is `null`, otherwise it returns the value of the attribute being accessed (i.e., `length`).

Other important Kotlin's features are (1) delegation, where the implementation of an interface is delegated to a specific object, (2) extensions where developers can extend the functionality of a certain class without modifying its declaration. (3) operator overloading, (4) data classes. Finally, similarly to Scala, it supports both declaration-site and use-site type variance. Note that non-nullable types, and extension methods are also part of the new Scala compiler, Dotty.

2.4.4 Groovy

The Groovy programming language supports both dynamic and static typing. A Groovy program can be run as a script, but there is also the Groovy compiler (i.e., `groovyc`) for compiling the code into Java bytecode. The syntax of Groovy is compatible with Java: every program written in Java should be accepted by `groovyc`. Groovy's type system is very similar to that of Java. However, an important difference is that Groovy provides flow-sensitive typing. In this context, the inferred type of a local variable changes depending on its location in the flow of the program.

2.5 Non-JVM Compiler Bugs and Prior Studies

In the related work, we first survey studies on compilers bugs. Then, we continue with other bug studies.

2.5.1 Understanding compiler bugs

The closest bug study to our work is that conducted by Sun et al. [43]. The authors collected and automatically analyzed 52,732 bugs and 31,399 revisions from the GCC and LLVM compilers. Their study focused on the following aspects: (1) location of bugs, (2) size of test cases and bug fixes, (3) lifetime of bugs, and (4) priorities of bugs. Some of their key findings are: 1) C++ is the most buggy component of the examined compilers, 2) GCC and LLVM bugs are typically triggered by small test cases 3) most of the bug fixes are local and 4) developers need a couple of months to resolve the reported bugs. Zhou et al. [51] recently repeated the empirical study initially conducted by Sun et al. [43], but this time, the researchers gave emphasis to optimization bugs in GCC and LLVM. Some of their results (e.g., size of test cases, duration of bugs, locality of bugs) are consistent with the findings of Sun et al. [43]. In a different spirit, Marcozzi et al. [29] tried to measure the effect of compiler bugs found by fuzzing tools on real-world application code. According to their results, most of the fuzzer-found bugs indeed affect the final executables produced by compilers, but they semantically change only a small portion of the code (typically involving a small number of functions). Our work is complementary to these previous studies; as it provides the first insights into understanding the nature of typing-related bugs, a category of bugs that is currently overlooked. The findings of our work can be combined with the findings of the other studies in order to perform a more rigorous testing of compilers.

2.5.2 Other bug studies

An often-cited examination of program faults is the seminal paper of Knuth [22]. A survey [30] lists 18 studies of software system faults and summarizes their results. Here we briefly present recent studies that are closely related to our work. Trying to investigate the characteristics of distributed concurrency bugs, Leesatapornwongsa et al. [26] manually analyzed 104 non-deterministic concurrency bugs from four distributed systems used in a production environment. Their analysis consisted of several aspects, including bug symptoms and fixing. Their findings contribute to the better understanding of distributed bugs and facilitate the design of future verification and testing tools. Other studies on concurrency bugs in server-side JavaScript Wang et al. [46]; Davis et al. [12] showed that such bugs are mainly caused by atomicity and ordering violations, and beyond shared memory, a significant number of bugs is triggered by races in external resources, such as files or databases. Bagherzadeh et al. [4] focused on actor-based concurrency bugs. They constructed a dataset consisting of 186 concurrency bugs found in Akka coming from Stack Overflow questions, and GitHub projects. For each bug in the dataset, they identified its symptom, root cause, and the Akka APIs that the buggy program uses. Their results showed that crashes is the most prevalent category of symptoms, while Akka concurrency bugs are mainly caused by logic faults.

Numerical bugs is another category of bugs that has been examined by previous empirical studies. Di Franco et al. [14] selected 269 numerical bugs from five popular numerical libraries (i.e., NumPy, SciPy, LAPACK, and classified them into four categories based

on their patterns and root causes. One of this study's takeaways is that some of the numerical bugs can be detected and fixed by adopting rule-based approaches, as many of them follow specific patterns, e.g., some accuracy bugs can be fixed by simply re-ordering arithmetic expressions. In a subsequent study, Dutta et al. [16] characterized inference-related bugs by manually analyzing 118 commits from three probabilistic programming systems. Their categorization involves accuracy bugs, bugs associated with the handling of special numerical values (e.g., `NaN`), and other correctness issues. Based on their findings, they also proposed a differential testing approach for finding such bugs.

Jin et al. [21] performed one of the first bug studies for performance bugs. They selected 109 real-word performance bugs from well-established systems (e.g., GCC, MySQL), and showed how these bugs are introduced and fixed. They designed a bug-finding tool that was able to detect 332 performance issues in MySQL, Apache and Mozilla.

3. METHODOLOGY

First, we create a corpus of typing-related bugs taken from the issue trackers of four JVM languages (Section 3.1). Then, we explain how we study and analyze the collected bugs (Section 3.2), and finally, we discuss the limitations and threats to validity of our method (Section 3.3).

3.1 Collecting bugs and fixes

Our bug collection approach consists of two stages, namely, *bug collection* and *post-filtering*. In the former phase, we search the issue trackers of the studied languages to gather *fixed* bugs related to the typing algorithms of the corresponding compilers. Our study excludes bugs related to the implementation of lexers and parsers. Similarly, bugs in the implementation of compiler optimizations or code generation are beyond the scope of this paper. The output of the first phase includes four sets containing the URLs of the retrieved bug reports. Each set $\mathcal{B}_l$ contains bugs related to a language l.

We further filter the collected bugs by performing the *post-filtering* step. This step aims to exclude bugs without any explicit fix, and bugs whose fix or report is not accompanied by a test case that triggers the bug. To do so, we follow three steps. First, for each language l, we get the ID of each previously collected bug $b \in \mathcal{B}_l$. Second, we search the repository of the corresponding compiler to find all the commits that refer to the given bug identifier. Third, for completeness, we use the GitHub API to retrieve pull requests that have references to the given bug. Note that it is a standard practice for the developers of the studied compilers to include the bug's numeric identifier in the description of their bug fixes.

In the *post-filtering* step, having kept the bugs for which we are able to find corresponding commits, we further examine their revisions to check whether they contain a test case. The development team of each compiler places test cases in dedicated directories, e.g., `tests/`. For instance, Scala developers put their test cases into the `tests/` directory of their repository. Therefore, to decide whether the associated commits contain a test case, we look for file updates in the aforementioned directories. When a commit does not contain a test case, we look into the corresponding bug report to discover and retrieve any linked test cases. At the end of the *post-filtering* phase, we obtain four sets of bugs, where each set $\mathcal{B}'_l$ is a subset of the corresponding set $\mathcal{B}_l$ produced by the first step of our collection approach.

While applying our bug collection method, we had to tackle the following challenge: the development teams of the languages under examination use diverse issue trackers, and adopt various categorization strategies for the reported bugs. Therefore, before collecting bugs, we carefully examined the corresponding issue trackers to identify all the relevant categories and filtering criteria that can be used to obtain fixed typing-related bugs.

Table 3.1 shows descriptive statistics of our bug collection effort. After applying the *bug collection*, and *post-filtering* step to each language, we got our final dataset, which consists of 4,153 bugs in total, of which 873 bugs are in Java compiler (`javac`), 1,433 bugs are in Scala compilers (either `scalac` or Dotty), 1,601 bugs are in Koltin compiler (`kotlinc`), and 246 bugs are in Groovy compiler (`groovyc`). In the following, we discuss some language-specific details.

Table 3.1: Statistics on bug collection. Each table entry shows per language statistics about 1) the total number of the reported issues (Total issues), 2) the number of the selected bugs after running the first phase of our approach (Phase 1), and 3) the number of the remaining bugs after running the second phase (Phase 2).

Language	Issue Tracker	REST Endpoint	Total Issues	Phase 1	Phase 2
Java	Jira	`https://bugs.openjdk.java.net/rest/api/latest/search`	10,872	1,252	873
Scala 2	GitHub	`https://api.github.com/repos/scala/bug`	12,315	1,180	1,067
Scala 3	GitHub	`https://api.github.com/repos/lampepfl/dotty`	4,286	429	366
Kotlin	YouTrack	`https://youtrack.jetbrains.com/api/issues`	40,998	2,189	1,601
Groovy	Jira	`https://issues.apache.org/jira/rest/api/2/search`	9,710	300	246

3.1.1 Collecting Java Bugs

Focusing on `javac` typing-related bugs, we inspected bugs reported in the OpenJDK project, which is the open-source implementation of the Java SE platform. The OpenJDK project employs the Jira issue tracker, which, at the time of writing, hosts 10,872 issues associated with a large number of JDK components, such as the JVM runtime, the Just in Time (JIT) compiler, Java's standard library, or other external JDK tools like the bytecode disassembler.

We used the Jira REST API to find JDK issues that meet the following selection criteria: 1) the type of the issue is "bug", 2) its status is either "resolved" or "closed", 3) its "resolution" field is set to "fixed", and 4) the issue is related to the Java compiler (i.e., the bug is assigned to the `javac` sub-component of JDK). Due to the large volume of JDK issues (> 200k), we applied two more filters. First, we selected bugs that affect JDK 7 and onwards. We excluded bugs that affect early versions of JDK where crucial features of Java (e.g., generics) are not present. Second, we filtered JDK bugs based on their priority. Specifically, we selected bugs that are considered important, and their priority is "P1", "P2", or "P3". Running the *bug collection* and *post-filtering* steps yielded the final set of `javac` bugs, namely $\mathcal{B}_j'$, containing 873 JDK issues.

Remark. The JDK developers do not classify `javac` bugs further. This means that we were unable to distinguish between parser bugs or type checker bugs in `javac` through automated means. We excluded any `javac` bug not related to our study during our manual analysis of bugs (Section 3.2).

3.1.2 Collecting Scala Bugs

We collected Scala bugs from two sources. The first source contains bugs reported for the Scala 2 compiler (`scalac`), while the second one includes bugs related to Dotty, the Scala 3 compiler. Both sources are using the issue tracking system of GitHub. For every bug report related to `scalac`, there is a dedicated repository, namely `scala/bug`. Dotty-related bugs on the other hand, are hosted in the official Github repository of the compiler. At the time of writing, 12,315 and 4,386 issues have been reported, in total, for `scalac`, and Dotty respectively.

The developers of these two compilers perform the classification of the reported issues by assigning different labels to each issue. We constructed two queries for fetching bugs that contain labels associated with Scala's type system and typing procedures. Specifically, for `scalac` bugs, we looked for *closed* GitHub issues to which at least one of the following labels is assigned: "typer", "infer", "should compile", "should not compile", "patmat", "overloading", "dependent types", "structural types", "existential", "gadt", "valueclass", "type-level", "compiler crash", "implicit classes", and "implicit". For Dotty, we were interested

in *closed* GitHub issues that combine the “itype:bug”, “itype:crash” or “itype:performance” label with at least one of the following labels: “area:typer”, “area:overloading”, “area:gadt”, “area:implicts”, “area:f-bounds”, “area:pattern-matching”, “area:erasure”, “area:match-types”. We also checked that the issues do not have the labels “won’t fix” and “stat:wontfix”. We used the Github REST API and fetched 1,180 bugs for `scalac`, and 429 bugs for Dotty. After excluding the bugs without an explicit fix or a test case, we were left with 1,067 and 366 bugs for `scalac`, and Dotty respectively. The final set of bugs $\mathcal{B}'_s$ includes 1,433 bugs coming from both Scala compilers.

3.1.3 Collecting Kotlin Bugs

Kotlin developers use the YouTrack issue tracker. Currently, it hosts 40,998 issues and bugs associated with different aspects of the Kotlin compiler, including type inference, code generation, IDE support and Android support.

We examined the tracker to identify issues with type: “bug” or “performance problem”, and status: “fixed”. Kotlin developers characterize issues that cause performance degradation in compilation or runtime as “Performance problems”. We further search for such issues. Also, Kotlin developers follow a fine-grained categorization for determining the components affected by the issue. This made it easy for us to identify bugs that occur in the implementations of the semantic analyses and type checker. Specifically, all typing-related compiler issues are assigned to categories prefixed by the term “Frontend”. Thus, we searched for Kotlin issues that belong to such categories. Bugs in the lexer and the parser are placed in a dedicated category named “Frontend. Lexer & Parser”, so it was easy for us to exclude them. Our search returned 2,189 Kotlin bugs. After running the *post-filtering* phase, we ended up with the final set of Kotlin bugs $\mathcal{B}'_k$. This set contains 1,601 elements.

3.1.4 Collecting Groovy Bugs

Groovy issues are hosted on a Jira instance that currently contains 9,710 cases. We were interested in Groovy issues that have the “bug” label, are either “closed” or “resolved”, and their resolution status is “fixed”. To identify typing-related bugs, we searched for issues assigned to a category named “Static Type Checker”. Our Jira query fetched 300 Groovy bugs. The *post-filtering* step produced the $\mathcal{B}'_g$ set consisting of 246 Groovy bugs.

3.2 Analyzing bugs

The total bug population contains 4,153 bugs. Since the manual analysis of each bug requires a certain amount of time to understand the root cause and the nature of the bug, it was not feasible for us to study every bug in the population. Therefore, we randomly sampled 80 bugs from the bug set of each language, leaving us with 320 bugs for manual analysis in total. Note that analyzing 320 bugs is consistent with the literature of bug studies. Specifically, Jin et al. [21] have manually analyzed 110 real-world performance bugs, Di Franco et al. [14] analyzed 269 bugs in numerical libraries, Leesatapornwongsa et al. [26], and Bagherzadeh et al. [4] analyzed 104 and 186 concurrency bugs in distributed and actor-based systems respectively.

To better understand the nature of the examined bugs, cover a wide range of scenarios on how these bugs are triggered, and reduce the possibility of getting biased, we chose to uniformly study bugs in the selected compilers rather than primarily focusing on a single one. To this end, our manual analysis was done in an iterative manner. Specifically, in every iteration, we randomly picked 20 bugs from each language set, and the first two authors made a first pass over the selected 80 bugs and excluded bugs that are outside the scope of this study (e.g., a parser bug in a compiler that was mistakenly selected during bug collection). After agreement, we randomly chose additional bugs, until we had 20 analyzable bugs for each language. Then, the actual analysis of these bugs occured. We studied each of the selected 80 bugs, (along with their fixes, and the discussion made by the development teams), and tried to assign every bug to categories based on 1) its symptom (RQ1), 2) its root cause (RQ2), and 3) the characteristics of the bug-revealing test cases (RQ4). The procedure described above was repeated four times, i.e., until studying 320 bugs in total. During these four iterations, we revisited, adapted (i.e., split, merged or renamed) the proposed categories, and, if it was necessary, re-assigned each aspect of bugs to other categories.

3.3 Threats to validity

One potential threat to internal validity is associated with the selection criteria and representativeness of the examined bugs. We were interested in *fixed* bugs accompanied with a fix and a test case. Such fixed bugs 1) are real bugs, 2) are important for the developers (since they are fixed), and 3) have enough information (i.e., a fix and a test case) to understand and characterize them. This is in line with prior work [43, 21, 14], where fixed bugs were also studied. For selecting typing-related compiler bugs, we first manually examined the categorization adopted by each development team, and applied (if possible) the necessary filters for fetching such bugs (e.g., getting all bugs prefixed with the “Frontend” term in case of Kotlin). When we did not have such information (e.g., in the case of `javac`), we did not apply additional filters. We avoided using keywords during search to reduce the chance of missing relevant bugs. In all cases, during our bug analysis, the selected bugs were manually examined, and excluded irrelevant ones.

A threat to external validity is the representativeness of the chosen languages and their compilers. We selected these programming languages, as they hold an important stake in the JVM technology. The vast majority of applications run on JVM are written in these languages. According to the annual statistics of Github [18], Java is among the top 3 most popular programming languages for the last five years, while Scala, beyond industry, it is also widely used in academia. Kotlin is gaining more popularity over the years, and now it has become the Google’s primary language for Android development [32]. Finally, Groovy is on the list of the top 20 most popular languages based on the TIOBE programming community index [44]. All these languages are object-oriented, while involving complicated features including, type inference, functional programming features, generics, overloading. We argue that the chosen languages can represent, to some degree, other statically-typed, object-oriented programming languages (e.g., C++, C#, TypeScript). However, some of the findings of our work may not be generalized to languages, such as Haskell, OCaml, or Go.

4. BUG STUDY

We present the main findings of our work providing answers to each of our research questions. All references to specific bugs provided as examples, are hyperlinked to the corresponding entry in the compiler project's issue tracking system.

4.1 RQ1: Symptoms

Every bug report of our dataset consists of a short description that contains information about how the bug is triggered along with the compiler's expected and actual behavior. To identify symptom categories, we answered the following question: *What did it make the bug reporter believe that something goes wrong in the compiler?* We carefully examined the differences between the compiler's expected and actual behavior, and grouped these differences into categories. We ultimatly identified 5 categories of symptoms, namely, *Unexpected Compile-Time Error*, *Internal Compiler Error*, *Unexpected Runtime Behavior*, *Misleading Report*, and *Compilation Performance Issue*. Figure 4.1 shows the distribution of the symptom categories. In the following, we discuss each symptom category in detail. Further, we elaborate on its frequency and impact, and present a concrete example of a compiler bug associated with the symptom.

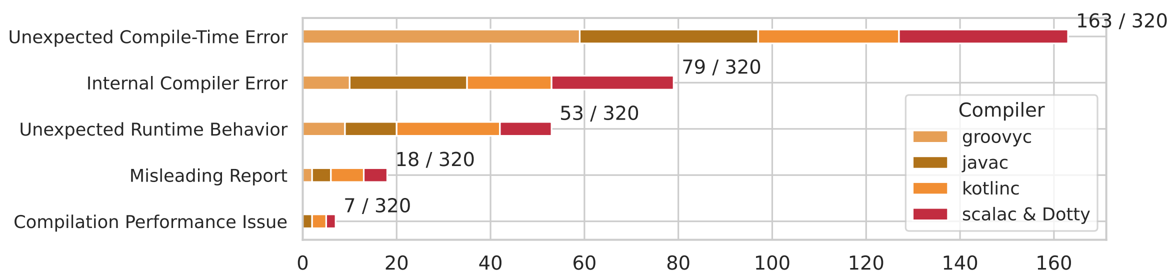


Figure 4.1: The distribution of symptoms.

4.1.1 Unexpected Compile-Time Error

A bug involving this symptom manifests itself when the compiler rejects a *well-typed* program, producing an informative error message to the developer. Such errors may frustrate developers, leaving them with the impression that their programs are indeed incorrect. *Unexpected compile-time error* is by far the most common symptom, accounting for 50.94% of the examined bugs.

., Listing 4.1 shows an instance of this symptom (related to `kotlinc` — see KT-10711). The program includes a parameterized class named `A` that takes one type parameter `T`, and defines a property named `f` whose type is given by the type parameter (line 1). Later,

Listing 4.1: KT-10711: A program that triggers a `kotlinc` bug

```
class A<T>(val f: T)
fun test() {
  listOf<String>().map(::A)
}
```

Listing 4.2: GROOVY-7618: A program that triggers a `groovyc` bug with an *internal compiler error*.

```
interface I {
  int m()
}
int m2(I x) {
  x.m()
}
void test() {
  m2 { -> 1  }
}
```

the program creates a list of strings (line 3). To convert every element of this list into an object of class `A`, the code applies the function `map` by passing a reference to the constructor of class `A`. `map` is a parameterized method in class `List` whose signature is `<I, O> map(fun: I => O): O`. Specifically, `map` 1) is instantiated with two type parameters, `I` and `O`, 2) expects a function with type `I => O` as input, and 3) returns a value of type `O`.

The Kotlin compiler rejects the above program providing the following error message: *"error: not enough information to infer type variable T"*. Specifically, `map` is applied to a list of strings. Thus, the type variable `I` of function `map` is instantiated with a `String` type, making `map` expect a function of type `String => O` as input. However, there is a bug in the inference engine of `kotlinc`, which prevents the compiler from instantiating the type variable `T` defined in class `A` (and as a result, the corresponding type of function reference `::A`) based on the expected function type `String => O`.

In the above example, the compiler considers the program as invalid and produces a corresponding diagnostic message (i.e., inference is not feasible). Other similar types of wrong error messages involve type mismatches (e.g., inferred type is `X`, but Y was expected), unresolved references (e.g., cannot find method `m`), and accessibility issues (e.g., private variable cannot be accessed in this context).

4.1.2 Internal Compiler Error

Internal Compiler Error (or crash) is the second most common symptom in our dataset (24.69%). Such errors manifest themselves when the compiler terminates its execution abnormally. This symptom differs from *unexpected compile-time error*, because the compiler is unable to yield a normal diagnostic message, or even generate target code. Internal compiler errors are clear indications that something is not working well in the compiler.

Listing 4.2 presents a Groovy program that triggers a bug (see GROOVY-7618) leading to an *internal compiler error*. The program defines a *single abstract method (SAM)* interface named `I` containing an abstract method `m` that takes no parameters and returns an integer (lines 1–3). Note that every SAM interface is also a functional one, meaning that instead of concrete classes, such SAM interfaces can also be implemented by lambda expressions and functions. The program later defines a function called `m2` that expects an instance of `I`, and returns a value of `int` by calling the method `m` of the given instance. Finally, the program calls `m2` by passing a lambda as an argument (line 8). The type of lambda is `() => int`.

When `groovyc` tries to coerce the type of lambda to a SAM type, it computes the arity of

lambda by accessing the property `params.length`. Note that `groovyc` internally represents a lambda expression with an object, which among other things, contains a field named `params` that stands for the parameter list of lambda. Nevertheless, the given lambda is parameterless (line 8), and therefore the value of `params` is `null`. This in turn, leads to a `NullPointerException`, because the `params.length` access is not guarded by a null-check of the receiver (`params`).

Internal compiler errors are often caused by runtime exceptions (e.g., `AOBE`). Also, they are triggered by failures of assertions included in the compiler's source code. Such failures happen, when the compiler is in an illegal state. For example, Dotty performs a post-condition check after type erasure to ensure that all types of the program tree are erased and are consistent with the type of system of JVM. This is not the case in Dotty-7041, where after type erasure the program tree contains an illegal type leading to an `AssertionError`.

4.1.3 Unexpected Runtime Behavior

The 16.56% of our bugs come with an *unexpected runtime behavior* symptom. Unlike previous symptoms, a bug related to an *unexpected runtime behavior* manifests itself when running the executable generated by the compiler. This involves the successful compilation of a given source program and the generation of a faulty executable that in turn, may lead to errors and wrong outcomes.

There are two reasons why a compiler may generate incorrect executables. First, a compiler bug can break the soundness of the type system. Hence, the compiler accepts an *invalid* program which it should have rejected. Such bugs are important, because they defeat the safety offered by type systems in statically-typed languages [34]. Second, the compiler may perform wrong static linking between methods and objects (e.g., it chooses the wrong overloaded method to call). Like miscompilations caused by optimization bugs, typing-related bugs with *unexpected runtime behavior* are very confusing for developers, and worse, they may be released unnoticed, as many of these unexpected runtime behaviors are triggered by specific application inputs.

Consider the Java program of Listing 4.3, which causes a known `javac` bug (see JDK-7041019) associated with an *unexpected runtime behavior* symptom. First, the code defines a parameterized interface `A` which is instantiated with a type parameter `E`. This interface contains an abstract method `m` expecting a value of `E` (lines 1–3). Another parameterized interface called `B` has one type parameter (`Y`), and extends the interface `A` instantiated as `A<Y[]>` (line 4). Later, a class called `C` implements `B<Integer>` by overriding the abstract method `m` (line 7). Furthermore, on lines 8–11, class `C` defines a static parameterized method, `m2`. This method defines a type variable `T` with upper bound `B<?>`. Also, `m2` receives a parameter `x` whose type is `T`, and returns nothing. The body of this method calls `x.m()` by passing an array of strings as input (line 10). Finally, the code defines `main`, which invokes `m2` using an instance of `C` as a call argument (12–14).

`javac` compiles this program successfully, and produces the corresponding bytecode. Unfortunately, JVM throws a `ClassCastException` when running the method call on line 10. This is because

JVM tries to pass an array of strings in a method expecting an array of integers (notice that the receiver of the callee method `m` is an object of class `C` at runtime, see lines 7, 10, 13)! This soundness issue is caused by a bug in `javac`. Specifically, when typing the

Listing 4.3: JDK-7041019: A program that triggers a `javac` bug with an *unexpected runtime behavior*.

```
interface A<E> {
  void m(E x);
}
interface B<Y> extends A<Y[]> { }
class C implements B<Integer> {
  @Override
  void m(Integer[] x) { }
  static <T extends B<?>> void m2(T x) {
    //Boom! ClassCastException at runtime.
    x.m(new String[]{"s"});
  }
  static void main(String[] args) {
    m2(new C());
  }
}
```

Listing 4.4: KT-5511: A program that triggers a `kotlinc` bug with a *misleading report*.

```
interface  X<T> {
    inner enum class C : X<T>
}
```

method call at line 10, `javac` instantiates the expected type of `m` (which at that time is `X[]`, where `X` stands for a fresh type variable) based on the upper bound of type parameter `T` (i.e., `B<?>`) of method `m2`. `javac` substitutes the type variable `X` with a capture type represented as `CAP#1`, but instead of creating an array type holding elements of type `CAP#1`, it mistakenly creates an array type that stores elements of type `Object`. After this incorrect type substitution, `m` now expects something of type `Object[]`. In Java though, arrays are covariant, thus, `javac` treats the argument type `String[]` as a subtype of the expected type `Object[]`, and mistakenly allows the call at line 10.

Some common runtime behaviors caused by typing-related bugs with an *unexpected runtime behavior* include bytecode verification failures (`VerifyError`), dynamic linking and resolution failures (`AbstractMethodError`, `IllegalAccessError`), execution failures (`NullPointerException`, `ClassCastException`), or wrong execution results.

4.1.4 Misleading Report

The fourth most common symptom is *misleading report* (5.62%). Such symptoms appear when for a given program, the compiler emits a false warning or a false error

False warnings and error messages may be misleading because they suggest ineffective fixes (e.g., warning about an unsafe cast, but the cast is actually safe). Furthermore, spurious messages can hide other program errors (e.g., the compiler reports a type mismatch error instead of an uninitialized variable error). Unlike *unexpected compile-time error*, in case of *misleading report*, the compiler correctly accepts (or rejects), a valid (or invalid) program. However, it does so by producing wrong diagnostic messages.

Listing 4.4 shows a Kotlin program triggering a bug (see KT-5511) with a *misleading report*

Listing 4.5: Dotty-10217: A program that triggers a Dotty bug with an *compilation performance issue*.

```
trait A
trait B
trait C
...
trait W
trait Foo[T]
val f: Foo[A | B | C | ... | W] = ???
```

symptom. The code defines a parameterized interface named `X` instantiated by one type parameter `T` (line 1). Inside the body of `X`, the code declares an inner enum named `C` that implements `X<T>`.

For this program, `kotlinc` generates two compile-time error messages: 1) *"error (2, 3): Modifier 'inner' is not applicable to enum class"*, and 2) *"error (2, 26): Expression is inaccessible from a nested class 'C', use 'inner' keyword to make the class inner"*. These two error messages are *contradictory*: the first message says that enum class cannot be inner, while the second one suggests developer make the enum class inner. This example program is indeed invalid, and the first error message is correctly reported by the compiler. Specifically, a Kotlin developer says: *"Enums should have a finite set of values, enum entries. Each enum entry is a result of statically invoking the constructor of the enum class. Inner class constructors have an additional non-static parameter which is an instance of the outer class. Thus they can't be invoked statically and therefore enums can't be inner."*. However, the second error of the compiler is spurious, and it is caused by a bug in the reporting mechanism of `kotlinc`.

4.1.5 Compilation Performance Issue

The least common symptom is *compilation performance issue* (2.19%). Bugs related to this symptom cause noticeable degradations in compilation performance. The impact of such bugs is the waste of developers' time and resources, because the compiler requires much time or memory to compile even the simplest fragment of code, and in many cases, compilation never terminates.

Consider the Scala program of Listing 4.5, which triggers a bug in Dotty (see Dotty-10217).

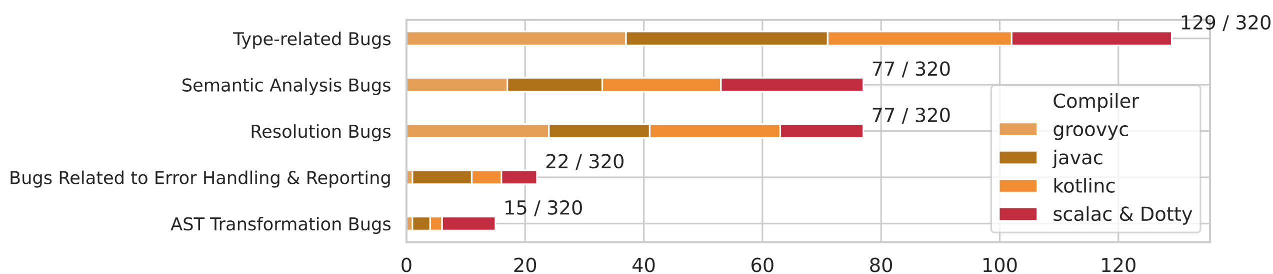
The program defines 24 types (most of them are omitted for brevity): from type `A` to type `W`. Then, the program defines a type constructor `Foo` which is instantiated with one type variable `T`. Finally, the code declares one variable named `f` with a type that comes from the application of type constructor with a union type consisting of types `A` to `W`.

Dotty spends roughly five minutes to compile this program. Specifically, Dotty performs a type optimization on union types: a union type of the form `T | Null` or `Null | T` becomes a regular type `T`. In this context, Dotty examines the union type passed as a type argument of type constructor `Foo` (line 7) to see whether this optimization is applicable to this union type. To do so, Dotty recursively checks if the union type consists of a bottom type (i.e., `Null` or `Nothing`) by using an internal function named `derivesFrom`, which returns true if a given type is an instance of a given class (e.g., `NothingClass`). The complexity of `derivesFrom` is exponential, which means that for a union type containing 24 terms, Dotty performs 2^{24} calls to `derivesFrom`!

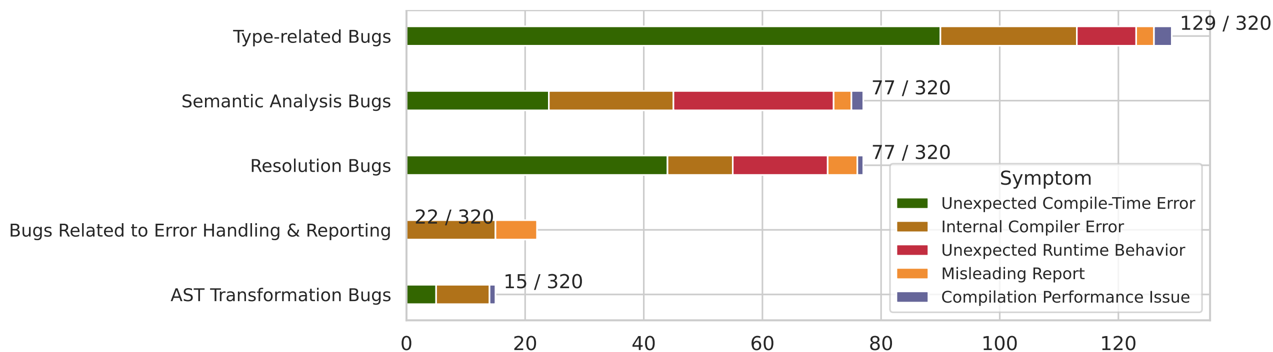
4.1.6 Comparative Analysis

From Figure 4.1, we observe similar trends among studied compilers. The *unexpected compile-time error* symptom is the most common symptom for all compilers followed by *internal compiler error*, and *unexpected runtime behavior*. The only exception is `kotlinc`, where *unexpected runtime behavior* is the second most common symptom category. Specifically, 22 `kotlinc` bugs were marked with an *unexpected runtime behavior* symptom, while 18 bugs were crashes. The high number of `kotlinc` bugs with an *unexpected runtime behavior* symptom is explained by missing well-formed checks on declarations in the compiler's implementation. An example of such a missing check is that a Kotlin class must not implement two interfaces containing members with conflicting signatures. Around three quarters of `groovyc` bugs (59 out of 80) make the compiler reject valid code, while we found only ten `groovyc` crashes compared to 18, 25, and 26 crashes found in the Kotlin, Java, and Scala compilers. Finally, we did not observe any `groovyc` bug causing any compilation performance issue.

4.2 RQ2: Bug Causes



(a) The distribution of bug causes per compiler



(b) The distribution of bug causes per symptom.

Figure 4.2: The distribution of bug causes.

We classified the examined bugs into categories based on their root cause. To do so, we studied the fix of each bug and identified which specific compiler's procedure was buggy. From our manual inspection, we derived five categories that include bugs sharing common root causes: *Type-related Bugs*, *Semantic Analysis Bugs*, *Resolution Bugs*, *AST Transformation Bugs*, and *Bugs Related to Error Handling & Reporting*. Figure 4.2 illustrates the distribution of our bug causes. In the following, we provide descriptions and examples for every category.

Listing 4.6: KT-9630: A Kotlin program that triggers a bug related to an *incorrect type transformation*.

```
interface A
interface B
class C : A, B
fun <T> T.m(): Unit where T : A, T : B { }
fun main() {
  C().foo()
}
```

4.2.1 Type-related Bugs

All types appearing in a program are checked against the set of allowed typing rules as incorporated in the compiler. A type system designates the main types allowed by a compiler, valid operations on these types, how types relate to each other and finally, permissible ways for the types to be combined. In this context, a compiler internally represents all types and properties of the underlying type system using specialzed data structures. Further, when typing an input program, it applies a broad spectrum of operations to these data structures based on the rules and design of the type system. Corresponding examples include, type variable substitutions, type constructor applications, subtyping checks, type normalizations, and more.

We define a *type-related bug* when one of these type operations is not implemented correctly. Since types and their operations are at the heart of a compiler, such correctness issues have a great impact on the ability of the compiler to accept the given code. Therefore, type-related bugs are mainly responsible for unexpected compile-time errors (see Figure 4.2b). We classified 129 out of 320 (40.31%) bugs as *type-related*, which makes this bug cause the most common one. Type-related bugs belong to one of the following scenarios: 1) *incorrect type inference & type variable substitution*, 2) *incorrect type transformation / coercion*, or 3) *incorrect type comparisons & bound computations*.

Incorrect Type Inference & Type Variable Substitution. In languages supporting type inference, explicit types may be omitted in a program. The compiler represents these omitted types with type variables, which in turn, are replaced with concrete types at compile-time, typically by solving a type constraint problem. Many type-related bugs are caused by building a wrong constraint problem (e.g., the constraint system contains excessive, missing, or contradictory constraints), or instantiating a type variable in a wrong way. As a result, for a certain type variable, the compiler infers a wrong type, or in many cases, it is unable to infer a type at all.

Listing 4.1 gives an example of such a bug. Due to an incorrect handling of function references, `kotlinc` constructs a constraint problem with incomplete constraints. This makes it impossible for the compiler to solve the system and find an optimal solution, leading to an *unexpected compile-time error*. Another example is shown in Listing 4.3. When dealing with an array type containing a type variable, `javac` performs a wrong type variable substitution, which causes a soundness bug.

Incorrect Type Transformation / Coercion. Guided by certain rules, a compiler may transform a certain type into another type for numerous reasons, e.g., type normalization, type erasure. For example, as shown in Listing 4.5, Dotty normalizes a union type of the form `T | Null` to `T`. Another example involves type erasure where all studied compilers erase type information from parameterized types.

Listing 4.7: JDK-8039214: A Java program that triggers a bug related to ***incorrect type comparisons*****.**

```
interface I<X1,X2> {}
class C<T> implements I<T,T> {}

public class Test {
  <X> void m(I<? extends X, X> arg) {}
  void test(C<?> arg) {
    m(arg);
  }
}
```

Similarly, we have the boxing and unboxing processes where a value type becomes a reference type, and vice versa. Diverse bugs in the implementation of these type transformations cause many problems.

As an example, consider Listing 4.6, where a Kotlin program triggers KT-9630. Specifically, this program defines a parameterized extension function named `m` instantiated by one type variable `T` that has two upper bounds: `A` and `B` (line 4). The code later calls this function using a receiver of type `C` (line 6). When typing this program, `kotlinc` instantiates type variable `T` with the intersection type `A & B`. Since in Kotlin, intersection types are only used internally for type inference purposes, `kotlinc` needs to convert the intersection type `A & B` into a type that is representable in a program. The problem in this example is that `kotlinc` fails to convert type `A & B` to type `C`. Consequently, `kotlinc` rejects the given code, because it is unable to find the method `m` in a receiver of type `C`, even though this type has been extended with method `m`.

Incorrect Type Comparison & Bound Computation. Another instance of type-related bugs are incorrect type comparisons and bound computations. A compiler applies different kinds of comparisons between types, which are underpinned by formal rules and relations of the type system. For example, a compiler consults the subtyping rules of the type system to check whether a value of type T_1 is assignable to a variable of type T_2. Beyond that, a compiler implements a number of algorithms dealing with type bounds, such as computation of lowest upper bound and greatest lower bound. We have identified many type-related bugs caused by type comparisons and bound computations that do not obey the rules of the type system.

Listing 4.7 demonstrates a `javac` bug (see JDK-8039214) caused by an incorrect type comparison. While type checking the call on line 7, `javac` checks whether the argument type `C<?>` is subtype of the expected type `I<? extends X, X>`. As part of this subtyping check, `javac` tests if the type argument `?` of type constructor `C` is contained in type argument `? extends X` of type constructor `I`. This type argument comparison is guided by the containment relation defined in the Java Language Specification (JLS) [19, §4.1.5]. Unfortunately, the implementation of `javac` does not follow this containment relation to the letter. Hence, it considers that `C<?>` is not subtype of `I<? extends X, X>`. This makes `javac` reject this well-formed program.

4.2.2 Semantic Analysis Bugs

Semantic analysis occupies an important space in the design and implementation of compiler front-ends. A compiler traverses the whole program and analyzes each program node

Listing 4.8: Scala2-5878: A Scala program that triggers a bug related to *missing validation checks*.

```
case class A(x: B) extends AnyVal
case class B(x: A) extends AnyVal
```

individually (i.e., declaration, statement, and expression) to type it and verify whether it is well-formed based on the corresponding semantics. A *semantic analysis bug* is a bug where the compiler yields wrong analysis results for a certain program node. The 24.06% of the inspected bugs are classified as semantic analysis bugs. A semantic analysis bug occurs due to one of the following reasons: 1) *missing validation checks*, and 2) *incorrect analysis mechanics*.

Missing Validation Checks. This sub-category of bugs include cases where the compiler fails to perform a validation check while analyzing a particular node. This mainly leads to unexpected runtime behaviors because the compiler accepts a semantically invalid program because of the missing check. In addition to these false negatives, later compiler phases may be impacted by these missing checks. For example, assertion failures can arise, when subsequent phases (e.g., back-end) make assumptions about program properties, which have been supposedly validated by previous stages. Some indicative examples of validation checks include: validating that a class does not inherit two methods with the same signature, a non-abstract class does not contain abstract members, a pattern match is exhaustive, a variable is initialized before use.

Consider the Scala program of Listing 4.8, which demonstrates a semantic analysis bug related to a missing validation check (see Scala2-5878). The program defines two value classes `A` and `B` with a circular dependency issue, as the parameter of `A` refers to `B`, and the parameter of `B` refers to `A`.

This dependency problem, though, is not detected by `scalac`, when checking the validity of these declarations. As a result, `scalac` crashes at a later stage, when it tries to unbox these value classes based on the type of their parameter. The developers of `scalac` fixed this bug using an additional rule for detecting circular problems in value classes.

Incorrect Analysis Mechanics. Another common issue related to semantic analysis bugs is *incorrect analysis mechanics*. This sub-category contains bugs with root causes that lie in the analysis mechanics and design rather the implementation of type-related operations, i.e., these bugs are specific to the compiler steps used for analyzing and typing certain language constructs. Incorrect analysis mechanics mostly causes compiler crashes and unexpected compile-time errors.

For example, in Dotty-4487, the compiler crashes, when it types `class A extends (Int => 1)`, because Dotty incorrectly treats `Int => 1` as a term (i.e., function expression) instead of a type (i.e., function type). Specifically, Dotty invokes the corresponding method for typing `Int => 1` as a function expression. However, this method crashes because the given node does not have the expected format. Dotty developers fixed this bug by typing `Int => 1` as a type.

4.2.3 Resolution Bugs

One of a compiler's core data structures is that representing scope. Scope is mainly used for associating identifier names with their definitions. When a compiler encounters an identifier, it examines the current scope and applies a set of rules to determine which

Listing 4.9: JDK-7042566: A Java program that triggers a *resolution* bug.

```
class Test {
  void test() {
    Exception ex = null;
    error("error", ex);
  }
  void error(Object o, Object... p) { }
  void error(Object o, Throwable t,
             Object... p) { }
}
```

definition corresponds to the given name.

In languages like those examined in our study where features, such as nested scopes, overloading, or access modifiers, are prevalent, name resolution is a complex and error-prone task. A *resolution bug* is a bug where the compiler is either unable to resolve an identifier name, or the retrieved definition is not the right one. We found that the 24.06% of our bugs lie in this pattern. These bugs are caused by one of the following scenarios: 1) there are correctness issues in the implementation of resolution algorithms, 2) the compiler performs a wrong query, or 3) the scope is an incorrect state (e.g., there are missing entries). The symptoms of resolution bugs are mainly unexpected compiler-time errors (when the compiler cannot resolve a given name or considers it as ambiguous) or unexpected runtime behaviors (when resolution yields wrong definitions) — see Figure 4.2b.

Listing 4.9 presents a test case that triggers the `javac` bug JDK-7042566. For the method call at line 4, `javac` finds out that there two applicable methods (see lines 6, 7). In cases where for a given call, there are more than one applicable methods, `javac` chooses the most specific one according to the rules of JLS [19, §15.12.2.2 and §15.12.2.3]. For our example, the method error defined at line 7 is the most specific one, as its signature is less generic than the signature of `error` defined at line 6. This is because the second argument of `error` at line 7 (`Throwable`) is more specific than the second argument of `error` (`Object`) at line 6. However, a bug in the way `javac` applies this applicability check to methods containing a variable number of arguments (e.g., `Object...`) makes the compiler treat these methods as ambiguous, and finally reject the code.

4.2.4 Bugs Related to Error Handling and Reporting

When an error is found in a given source program, modern compilers do not abort compilation. Instead, they continue their operation to find more errors and report them back to the developers. In the context of type checking this is typically done by assigning a special type (e.g., the top type) to erroneous expressions. Compilers also strive to provide informative and useful diagnostic messages so that developers can easily locate and fix the errors of their programs. A *bug related to error handling & reporting* is a bug where the compiler correctly identifies a program error, but the implementation of the procedures for handling and reporting this error does not produce the expected results. We found that the 6.88% of our bugs are associated with error handling and reporting. All bugs of this category are related to crashes and wrong diagnostic messages (i.e., misleading reports).

For example, the Kotlin program of Listing 4.4 triggers a bug related to error handling and reporting. As already discussed, in this program, `kotlinc` produces an excessive

Listing 4.10: Scala2-6714: A Scala program that triggers an *AST transformation bug*.

```
class A
class B {
  def apply(x: Int)(implicit a: A) = 1
  def update(x: Int, y: Int) { }
}
object Test {
  implicit val a = new A()
  val b = new B()
  b(3) += 4  // compile-time error here
}
```

diagnostic message. This message suggests developers to take actions that contradict with previously reported messages.

4.2.5 AST Transformation Bugs

The semantic analyses of a compiler works on a program's abstract syntax tree (AST). Before or after typing, a compiler applies diverse transformations and simplifications to the AST so that the given program is expressed in terms of simpler constructs.

For example, `javac` applies a transformation that converts a foreach loop over a list of integers `for (Integer x: list)` into a loop of the form `for (Iterator<Integer> x = list.iterator(); x.hasNext();)` An *AST transformation bug* is a bug where the compiler generates a transformed program that is not equivalent with the original one, something that invalidates subsequent analyses. We found that the 4.69% of our bugs are AST transformation bugs, which cause many unexpected compile-time and internal compiler errors.

Listing 4.10 demonstrates an instance of this bug category (see Scala2-6714). This Scala 2 program defines a class `B` overriding two special methods named `apply`, and `update` (lines 2–5). The function `apply` allows developers to treat an object as a function. For example, a variable `x` pointing to an object of class `B` can be used like `x(10)`. This is equivalent to `x.apply(10)`. Furthermore, the `update` method is used for updating the contents of an object. For example, a variable `x` of type `B` can be used in map-like assignment expressions of the form `x(10) = 5`. This is equivalent to calling `x.update(10, 5)`. Notice that in our example, the `apply` method takes an implicit parameter of type `A`. This means that when calling this function, this parameter may be omitted, letting the compiler pass this argument automatically by looking into the current scope for implicit definitions of type `A`.

Before `scalac` types the expression on line 9, it "desugars" this assignment, and expresses it in terms of method calls. For example, `b(3) += 4` becomes `b.update(3, b.apply(3)(a) + 4)`. Note that the final argument `a` passed in `apply` call corresponds to the implicit definition of line 9. However, due to a bug, `scalac` ignores the implicit parameter list of `apply`, and therefore, it expands the assignment of line 9 as `b.update(3, b.apply(3) + 4)`. Consequently, the expanded method call does not type check, and `scalac` rejects the program.

4.2.6 Comparative Analysis

According to Figure 4.2a, type-related bugs form by far the most common bug cause for all studied compilers. This suggests that reasoning about types is a complex and challenging task for compilers, and that the corresponding type system implementations are susceptible to errors. Type-related bugs, resolution bugs, and semantic analysis bugs are almost uniformly distributed across studied compilers. The Scala compilers and `javac` are the outliers though. Specifically, we classify more `javac` bugs as *bugs related to error handling & reporting* compared to the remaining compilers. Furthermore, notice that AST transformation bugs are particularly common in Scala compilers. We attribute this to the fact that Scala is a very powerful language, meaning that individual features are often combined together to establish new features and use cases. `scalac` and Dotty apply a large number of transformations (e.g., Dotty implements more than 50 passes until it performs type erasure) that simplify program tree so that complex features are expressed through simpler primitives. AST transformation bugs of Scala are associated with eta expansion, inlining, and desugaring of various language constructs.

4.3 RQ3: Bug Fixes

To get an insight into the complexity of typing-related compiler bugs, we studied how these errors are introduced, and what are the properties of their fixes. We examined the revisions of each bug fix to measure its size and how many components are affected by the fix. Finally, we computed and examined the time compiler developers need to resolve a bug.

4.3.1 How Bugs are Introduced?

To understand how bugs are introduced, we manually studied every bug fix and the discussion among developers in the corresponding bug reports or commit messages. We found that the bugs of our dataset are mainly introduced by logic errors, algorithmic errors, design errors, or other programming errors.

Logic errors, which stand for defects in logic, sequencing, or branching of a procedure [52], are the dominant source of bugs in our dataset. Most of these logic errors are missing cases or steps in the implementation of a routine, or incorrect conditions of an `if` statement. Other instances of logic errors are extraneous computations, incorrect sequence of operations, or wrong / insufficient parameters passed to a function. We observed that the fixes of logic errors usually include changes to a single method or file and consist of few lines of code. For example, many logic errors are fixed by adding a missing `if else` case in the body of a buggy method.

Algorithmic errors are related to errors in the structure and implementation of various algorithms employed by compilers (e.g., inference of a type variable, resolution of a method). Algorithmic errors arise either because the implementation of an algorithm is wrong or because a wrong algorithm has been used. Unlike logic errors, fixes of algorithmic errors usually involve changes in a few dozen lines of code. A characteristic example of this category is Dotty-10217 ListingFigure 4.5), in which the implementation of the underlying algorithm has exponential complexity.

In contrast to logic and algorithmic errors that describe defects in compilers' implementations, *language design errors* express issues at a higher level. They describe the cases

where although the compiler has the intended behavior and is not buggy, a program reveals that this behavior can lead to undesired results. As a result, a re-design is essential for both the language and the compiler. Fixes of design errors include changes from a few code lines to significant refactorings in a compiler's code base. For example, KT-11280 demonstrates a bug that stems from a design issue in the language. When encountering a condition of the form `if (x == A()) b else c`, `kotlinc` implicitly coerces the type of `x` to type `A` inside the true branch of the `if` statement. However, KT-11280 demonstrates that this behavior is a source of unsoundness. A developer is free to override the method `equals` (this is the method invoked when performing ==), meaning that `x` is not guaranteed to have a type of `A` whenever the check `x == A()` returns true. Kotlin designers and developers fixed this by forbidding these implicit coercions whenever `equals` is overridden.

Other programming errors we observed include declarations of a variable with an incorrect data type, out-of-bounds array accesses, accesses to null references, and unchecked exceptions. For example, the `groovyc` bug of Listing 4.2 is introduced by a missing null check causing a `NullPointerException`. The fixes of such faults are usually trivial and involve a single change in one line of code.

4.3.2 Size of Bug Fixes

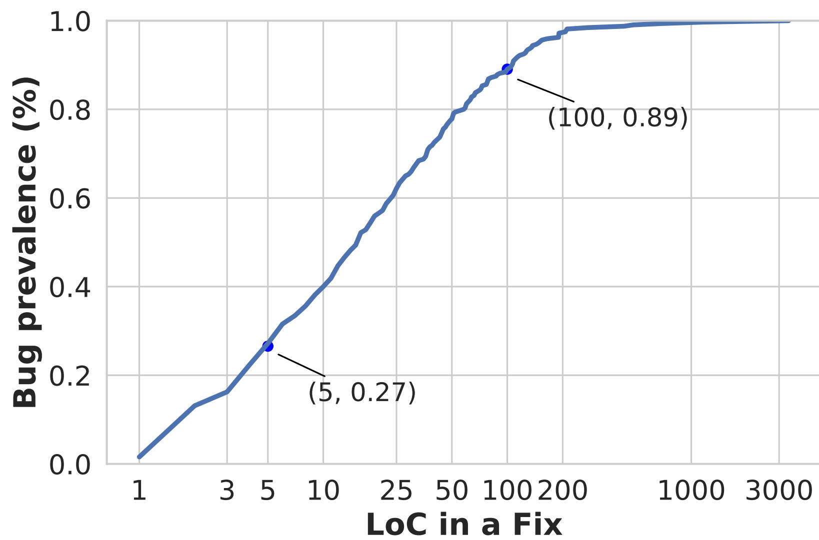
We considered the revisions of every bug fix of our dataset, and we excluded file modifications and creations related to test files (e.g., test cases) plus all non-source files (e.g., updates in docs). Using automated means, we counted the lines of code modifications made to source files, and we computed how many source files are updated in each fix.

As Figure 4.3a shows, 89% of the bug fixes contain fewer than 100 lines of code, and 40% of the bug fixes are less than 10 lines. These results are consistent with the study of Sun et al. [43]. Specifically, the study indicates that 92% and 50% of the GCC and LLVM bug fixes include less than 100 and 10 lines of code respectively. On average, the number of lines of code modified in a bug fix is 52, and the median is 16. For completeness, Figure 4.3b shows how many files are modified in a bug fix. The majority of fixes change only few files: 60% of the patches update a single file, and only 4% of the fixes change more than 5 files. One exception to this pattern is Scala-2742, where the corresponding fix requires updates in the Scala specification, which result in scattered updates across multiple compiler components. In summary, this fix consists of more than 3,000 lines of code and modifies more than 40 source files.

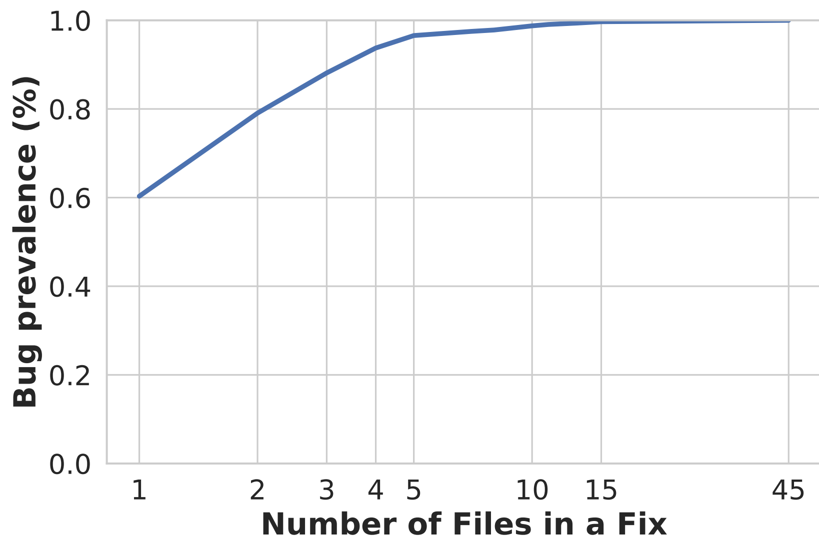
4.3.3 Duration of Bugs

Considering the plots in Figure 4.3, a reader may conclude that most of the bugs are simple and easy to fix, because they affect only a small part of the compiler. Despite the small size of fixes, during our manual inspection, we observed that many bugs are challenging to solve. The developers have long-lasting conversations about potential solutions and their implications. Hence, we decided to investigate the bugs' lifetime to better understand the complexity of bug fixes. To do so, we conducted a quantitative analysis of the time that elapsed in order to fix them. All bug tracking systems of our studied compilers provide details about the creation date and resolution date of each bug report. We defined the duration of a bug as the time interval between its creation and resolution date.

Figure 4.4 shows the bugs' cumulative distribution function over time. The blue plot indic-



(a) Cumulative distribution of lines of code in a fix.



(b) Cumulative distribution of files in a fix.

Figure 4.3: Size of bug fixes.

ates that over a half of the investigated bugs were fixed in one month, and 15% of the bugs took more than a year to be fixed. In terms of days to fix, the median is 24 days and the mean is 186 days. This suggests that many typing-related bugs are not fixed immediately after a bug report is opened. Indeed, we encountered many cases where the corresponding bugs undergo careful examination and risk evaluation by developers and the language committee. This is because fixes of typing-related bugs can potentially break backward compatibility — a fixed compiler may not be able to compile existing programs that rely on the old compiler's behavior. Therefore, to prevent regression bugs, developers carefully estimate the impact of each suggested fix. For example, after one year of discussions, the Java team decided to address JDK-8075793 so that existing applications written in Java 7 do not break under the new versions of `javac`. Beyond that, many typing-related bugs are closely related to the language specification and design (e.g., Scala-2742, and KT-22517), and they require fixes and enhancements in both the implementation of the compiler and the design of the language.

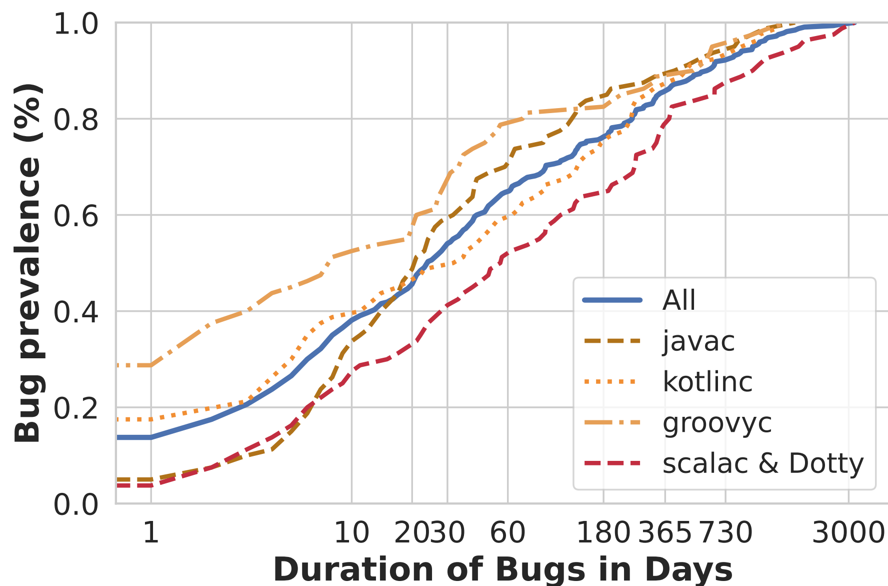


Figure 4.4: Cumulative distribution of bugs through time.

4.3.4 Comparative Analysis

Consider again Figure 4.4. `groovyc` bugs (see yellow line) need considerably less time to be fixed than the bugs of the other compilers. Specifically, the median duration of `groovyc` bugs is only 8 days, while the median duration is 21, 34 and 55 days for `javac`, `kotlinc`, and Scala bugs respectively. One explanation to this deviation could be that `groovyc` is not as mature as the other compilers, and many `groovyc` bugs are simple programming errors (e.g., GROOVY-7618, Listing 4.2), which can be fixed easily, rather than defects that require much domain expertise and knowledge. Another explanation may lie in the motivation and resources associated with the project's development team.

We also performed the Mann-Whitney U test to examine whether the distributions of bugs' duration are the same for the studied compilers. We found that the duration of Groovy and Scala bugs is statistically different than that of Kotlin and Java bugs, while the durations of Kotlin and Java bugs are not.

4.4 RQ4: Test Case Characteristics

We now present a discussion on the characteristics of the bug-revealing test cases. Studying the characteristics of test cases gives us an intuition regarding what language features are promising for uncovering typing-related bugs.

4.4.1 General Statistics

Table 4.1 presents some general statistics on test cases. Roughly 67.5% of the inspected bugs are triggered by compilable test cases. However, around one third (32.5%) of typing-related bugs occurs when compiling invalid code, i.e., the corresponding test case contains e.g., type mismatches, ill-formed declarations. This is an important observation, because in addition to using valid test cases (as prior work did for detecting optimization bugs [24, 49, 28]), identifying typing-related bugs requires passing *non-compilable*

Table 4.1: General statistics on test case characteristics.

Compilable test cases	216 / 320 (67.5%)
Non-compilable test cases	104 / 320 (32.5%)
LoC (mean)	10.2
LoC (median)	8
Number of class decl. (mean)	2
Number of class decl. (median)	2
Number of method decl. (mean)	2.9
Number of method decl. (median)	2
Number of method call (mean)	2.5
Number of method call (median)	1

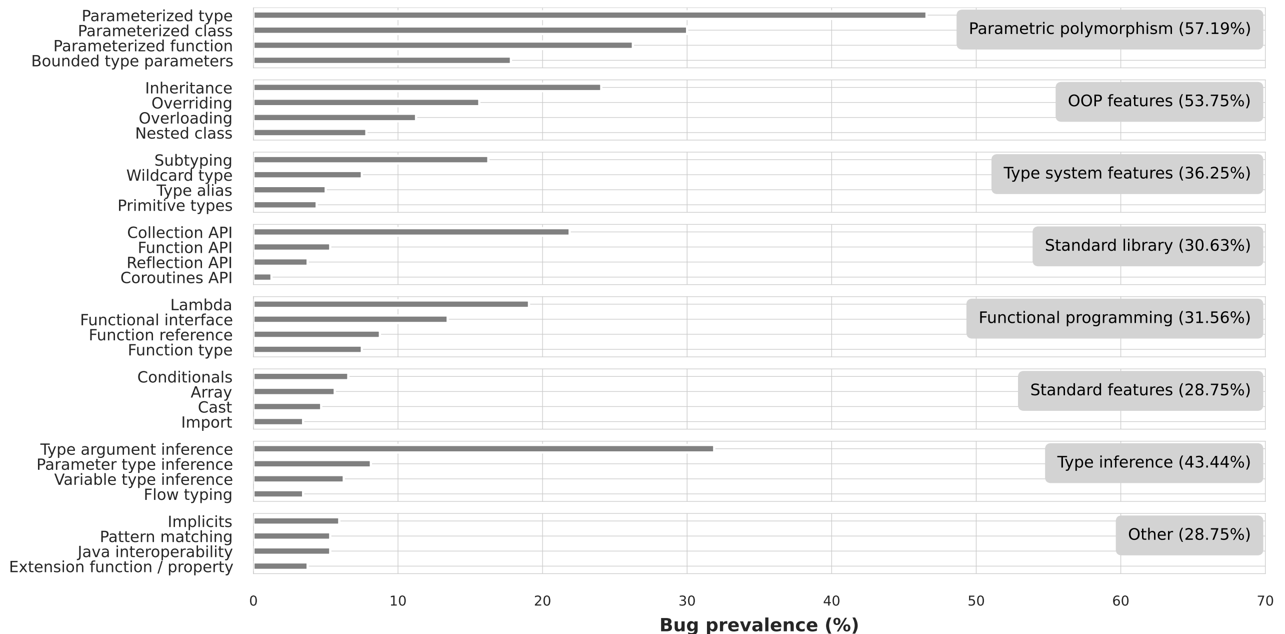


Figure 4.5: The classification of the language features that appear in test cases, along with their frequency. For each category, we show the four most frequent features.

programs as input to the compiler under test. These incorrect programs mainly trigger bugs that cause internal compiler errors, unexpected runtime behaviors, and misleading reports. The average size of test cases is 10.2 lines of code (LoC), while the median is 8 LoC. This suggests that typing-related bugs are mainly triggered by small fragments of code. Note that Sun et al. [43] made similar observations for GCC and LLVM bugs.

4.4.2 Language Features

We also identified what specific language features are involved in each test case. Since the studied languages are primarily object-oriented, we excluded prevalent object-oriented features that we encountered in almost every test case (e.g., class declaration, object initialization). Then, we grouped the features exercised in every test case into eight categories: 1) *standard language features* containing features seen in almost every language (e.g., exceptions, casts, loops), 2) *object-oriented programming (OOP) features* (e.g., overriding, inheritance), 3) functional programming features (e.g., higher-order functions), 4) *parametric polymorphism* (e.g., parameterized functions), 5) *type inference features* (e.g., type argument inference) 6) *type system-related features* (e.g., subtyping), 7) *standard library* (e.g., use of collection API), and 8) *Other* including features not belonging to any of the previous categories (e.g., named arguments).

Table 4.2: The five most frequent and the five least frequent features supported by all studied languages.

Most frequent features		Least frequent features	
Feature	**Occ (%)**	**Feature**	**Occ (%)**
Parameterized type	46.56%	Multiple ‘implements’	2.19%
Type argument inference	31.87%	‘this’ expression	2.19%
Parameterized class	30%	Arithmetic expression	1.88%
Parameterized function	26.25%	Loops	1.25%
Inheritance	24.06%	Sealed class	0.94%

For every category of features, Figure 4.5 presents its frequency along with its four most frequent features. Parametric polymorphism is pervasive in the corresponding bug-revealing programs: more than a half of the examined bugs are caused by test cases (57.19%) containing features, such as declaration and use of parameterized functions, use-site variance, and bounded type parameters. Another interesting observation is that around one third of test cases employ the standard library, and especially the collections API, which includes functions and classes for creating and manipulating data structures (e.g., lists). An example of such a test case is the program of Listing 4.1. Interestingly, APIs such as, the collections API or the reflection APIs, heavily rely on parametric polymorphism. Therefore, for stress-testing compilers, a future program generator could generate expressions that use these APIs in a complex manner without requiring to generate the corresponding parameterized definitions. Other frequent features are: inheritance for OOP features, subtyping (e.g., `A x = new B()`) for type system-related features, lambda expressions for functional programming features, conditionals for standard features, type argument inference (e.g., `X<String> x = new X<>()`) for type inference features, and Scala implicits for *Other*.

Table 4.2 shows which features are the most frequent and which features are the least frequent in the examined test cases. This table presents features that are supported by all studied languages. Features associated with parametric polymorphism (i.e., usage of parameterized types, functions and classes) and their combinations with type argument inference are highly common in the bug-revealing test cases. However, features like arithmetic expressions and loops have a small bug-triggering capability, as they appear only in 1–2% of the bug-revealing programs. This finding contradicts prior testing efforts [24, 28] for optimization bugs, which rely on programs with complex arithmetic expressions, control- and data-flow (e.g., nested loops).

To find out whether there are any interesting features’ combinations that are more likely to trigger bugs, we also computed the lift score, which has been also used in previous bug studies [21], For two features *A* and *B*, the lift score is given by: *lift*$(A, B) = \frac{P(A \cap B)}{P(A)P(B)}$, where $P(A \cap B)$ is the probability of a test case containing both features *A* and *B*. The lift score gives an estimation of how strongly two features are correlated. A lift score greater than 1 means that the features are positively correlated: when a test case contains feature *A*, it is also likely to contain feature *B*. A lift score close to 1 indicates no correlation, while a lift score smaller than 1 denotes that the features are negatively correlated.

The most positively correlated categories are *standard library* with *functional programming features*, and *standard library* with *type inference features* with a lift score of 5. Indeed, many bug-revealing test cases invoke higher-order methods coming from the standard library, such as the function `map` in the program of Figure 4.1.1. Moreover, such test cases often let the compiler infer some type information, e.g., signature of lambda expres-

Table 4.3: The five most bug-triggering features per language.

Java		Scala		Kotlin		Groovy	
Feature	**Occ (%)**	**Feature**	**Occ (%)**	**Feature**	**Occ (%)**	**Feature**	**Occ (%)**
Parameterized type	51.25	Parameterized type	57.5	Parameterized type	36.25	Parameterized type	41.25
Type argument inference	42.5	Parameterized class	42.5	Parameterized class	33.75	Collection API	35
Functional interface	37.5	Inheritance	32.5	Type argument inference	32.5	Type argument inference	35
Parameterized function	35	Implicits	23.75	Parameterized function	26.25	Lambda	25
Parameterized class	30	Parameterized function	22.5	Inheritance	25	Parameterized function	21.25

sions, or type argument of a callee parameterized function. Regarding individual features, some interesting combinations are: 1) variable arguments with overloaded methods (e.g., Listing 4.9) with a lift score of 24, 2) use-site variance with parameterized function (e.g., Listing 4.3) with a lift score of 17.1, 3) type argument inference with parameterized function (e.g., Listing 4.3, 4.6) with a lift score of 12.7, 4) Scala implicits with parameterized class with a lift score of 10.9, 5) type argument inference with collection API (e.g., Listing 4.1) with a lift score of 8.6, and 6) type argument inference with parameterized types with a lift score of of 7.

Remark. Our analysis on test case characteristics does not provide information about the behavioral characteristics of each test case. For example, our analysis gives the frequency of casts, but it does not offer details about how and where a cast is used [31]. However, we argue that future testing techniques can still take advantage of our findings to produce interesting programs by considering features that are more likely to trigger bugs, and combining these features in divergent ways (see Section 6).

4.4.3 Comparative Analysis

Table 4.3 shows the five most bug-triggering features per language. Again, parametric polymorphism-related features are in the top of every language under study. Based on our observation, such techniques could be applicable to more than one compiler. For example, future testing methods could be effective for testing both `kotlinc` and `javac`, as both compilers suffer from many bugs caused by test cases that use e.g., parameterized functions. Another interesting finding is that implicits, a powerful and popular Scala-only feature [23], appears in 23.75% of the examined `scalac` and Dotty bugs. Therefore, in addition to parametric polymorphism, Scala implicits is a feature which researchers and Scala developers could profitably invest time to deeply test. Beyond implicits, other language-specific features that are common are: pattern matching (21.25%), higher-kinded types (13.75%), and algebraic data types (13.75%) for Scala, as well as nullable types (16.25%), and extensions (15%) for Kotlin.

5. IMPLICATIONS AND DISCUSSION

We now discuss several implications of our work, and how our findings can serve as a basis for future research endeavors in compiler testing.

Typing-related bugs have diverse manifestations. Contrary to optimization bugs, which mainly manifest as at runtime as errors [24, 49], typing-related bugs can potentially affect both compilation and runtime (Section 4.1). Researchers should develop appropriate test oracles that can catch typing-related bugs with a plethora of manifestations. For example, for finding bugs that manifest as unexpected compile-time errors, a fuzzer should generate programs that are valid by construction so that rejection of these programs indicates a potential bug. Similarly, for detecting bugs with a *misleading report* symptom, a fuzzer should generate or use programs with known compile-time errors or warnings, and compare these expectations with the actual ones using a form of pattern matching (e.g., via regular expressions).

Typing-related bugs are located in few specific compiler components. According to Figures 4.3a and 4.3b (which show that bug fixes are mostly local), typing-related bugs are caused by incorrect implementations of some few and specific compiler tasks and routines. In Section 4.2, we showed that these buggy tasks and routines are typically associated with operations on types (e.g., type inference), name resolution, semantic analysis of declarations, desugaring, or error handling & reporting. A possible direction for researchers is to introduce targeted methods for identifying bugs in these components. For example, for finding bugs related to type inference, a mutation strategy could gradually remove type information from a program, e.g., from variable declarations, or type constructor applications. For triggering bugs in resolution algorithms, a promising approach could be the creation of programs that contain and use many overloaded methods or nested declarations. Similarly, for detecting missing validation checks, a potential mutator could inject faults in the program's declarations, e.g., it could inject a circular dependency as in the program of Listing 4.8.

A large number of typing-related bugs is triggered by non-compilable programs. Almost one third of the studied bugs is triggered by invalid code (Table 4.1). This observation comes in contrast to existing compiler testing techniques, which feed compilers with compilable programs [49, 24, 28]. Generating incorrect programs is a challenging task, as the generated programs must be *subtly* incorrect, meaning that the programs should be syntactically correct and contain at most one semantic error. A future research direction could be the proposal of new program generators and mutators that provide such invalid test cases. However, since the search space of invalid programs is enormous, a challenge related to this is to determine the program point to inject the fault, and the nature of the injected fault (e.g., type mismatch error or non-static method in a static context call error). To address this, a technique similar to *skeletal program enumeration (SPE)* [50] could be used to enumerate all subtly invalid programs based on a given program structure.

Parametric polymorphism is the feature with the most bug-triggering capability. Test cases that make an extensive use of parametric polymorphism-related features are responsible for more than a half of the examined bugs (Tables 4.2, 4.3). Therefore, parametric polymorphism is a promising feature that future program generators should consider for uncovering typing-related bugs. Parametric polymorphism is supported by all the studied languages. Conseququently, parametric polymorphism-oriented testing techniques (e.g., a mutator that converts a given class / function into a parameterized one) could be invaluable for testing multiple compiler implementations. For example, such a

mutator could be applied to testing both `javac` and `kotlinc`. Finally, our findings suggest that parametric polymorphism works well with type argument inference (Section 4.4.2). Therefore, generating programs involving parameterized types and functions that omitted type arguments is another interesting research direction.

Use of the standard library is pervasive in test cases. Based on our observation that around one third of our test cases use the standard library and particularly the collection API, an interesting direction for stress-testing compilers could be the generation of small expressions that use these APIs in a complex manner, without requiring the generation of the corresponding definitions. (e.g., see Listing 4.1, line 4). Interestingly, such APIs heavily rely on parametric polymorphism.

Control-flow constructs and arithmetic expressions are not common in bug-revealing test cases. Table 4.3 shows that control-flow constructs (e.g., loops) and arithmetic expressions barely trigger typing-related compiler bugs. This conflicts with the design and motivation of prior approaches [49, 28, 37]. For example, as an effort to uncover optimization bugs, the recent work of Livinskii et al. [28] adopts a generation policy that creates complex arithmetic expressions and bitwise operations. Our findings suggest that the existing techniques should be adapted so that they also consider features that are more likely to cause typing-related bugs. This would lead to a more holistic testing of compilers.

Implicits and pattern matching are two promising features for testing Scala. Implicits and pattern matching appear in 23.75% and 21.25% of the examined `scalac` and Dotty bugs. Hence, in combination with parametric polymorphism, it is worth proposing methods that are specifically targeting these Scala features. For example, future testing methods can be inspired by the work of Křikava et al. [23], which describes how implicits are used in practice, to produce programs that, in turn, exercise different implicits' idioms and patterns in a complex manner. Similarly, for validating exhaustiveness checks of pattern matching expressions, a possible testing solution could be the generation of random algebraic data types, along with the enumeration of their corresponding match patterns. Such a technique could be also applicable to languages such as Haskell or OCaml.

6. A PROOF-OF-CONCEPT PROGRAM GENERATOR

We demonstrate the leverage obtained from our work's findings through the design and implementation of a proof-of-concept Kotlin and Groovy test-program generator. Specifically, our prototype relies on the following observations: 1) parametric polymorphism is a crucial feature for uncovering typing-related bugs, 2) parametric polymorphism is supported by both Groovy and Kotlin, and 3) parameterized types are often combined with type argument inference (Section 4.4.2).

Our program generator produces programs written in an intermediate language (IR) representing a simple object-oriented language that has a limited support on parametric polymorphism and type inference. Specifically, our language supports class declarations, method declarations, local variable declarations, inheritance, subtyping, object initializations, assignments, method calls, property accesses, constant expressions, (e.g., integers), conditionals, logical operators (e.g., `&&`), comparison operators (e.g., $<=$), parameterized classes, bounded type parameters, declaration-site variance, type argument inference, and local variable type inference. Regarding the type system, our IR contains three different kinds of types: (1) abstract types for representing type variables and type constructors, (2) parameterized types for representing types that come from type constructor applications, (3) regular types for representing any type that is neither abstract nor parameterized. Note that our IR does not support other special types, such as primitive types or nullable types.

Our algorithm for generating programs written in the IR is simple and straightforward. Every program is *well-formed* by construction and consists of a number of randomly generated declarations (i.e., classes or methods). Each class declaration includes a random number of methods and fields, while each method declaration contains one or more expressions. During generation, we randomly assign types to the signature of methods (e.g., return type, formal parameter type), or the signature of class fields. These types are chosen randomly from our *type pool*, a data structure that contains types that have already defined in the program. For example, whenever we generate a class declaration, we add the corresponding type to the pool. For generating expressions, we randomly pick a type t from the pool and then, we randomly generate a random expression whose type is a subtype of the type t. Finally, whenever it is applicable, our algorithm randomly omits type information from local variables' declaration, or type constructor applications.

Representing the generated programs through an IR allows us to test both compilers. In particular, having generated a program in IR, we translate it into a concrete source file (e.g., a Kotlin source file) using language-specific translators. Every translator traverses the input program (written in IR) and converts every declaration / statement / expression into the corresponding one that follows the syntax of the target language. The output of this translation is passed to the compiler under test. When the compiler is unable to compile the given source file (i.e., it crashes or reports a compile-time error), a potential bug is found.

Table 6.1 gives a summary of the bugs found by our program generator. In total, we found 28 previously unknown bugs in `kotlinc` and `groovyc`, of which 16 bugs have been already fixed by developers. Our tool was able to find 19 bugs in `groovyc` and 9 bugs in the Kotlin compiler. Almost all of the reported bugs manifest as unexpected compile-time errors, while all but two are typing-related bugs, i.e., one bug was classified as a back-end bug, and one was classified as a parser bug by the Kotlin developers. These bug detection results demonstrate that the observations of our study can indeed guide the design of

Table 6.1: Summary of the bugs found by our proof-of-concept tool. In total, we have found 28 bugs in `kotlinc` and `groovyc`, of which, 16 have been fixed by developers.

	groovyc		**kotlinc**	
Symptom	**Confirmed**	**Fixed**	**Confirmed**	**Fixed**
Unexpected compile-time error	5	14	7	1
Internal compiler error	0	0	0	1
Total	**5**	**14**	**7**	**2**

future techniques on compiler testing, which 1) are useful for finding typing-related bugs, 2) are applicable to more than one compilers.

7. CONCLUSIONS AND FUTURE WORK

We presented the first empirical study of 320 typing-related bugs found in compilers of four popular JVM languages, that is, Java, Scala, Kotlin, and Groovy. Unlike optimization bugs, typing-related bugs have diverse manifestations: from unexpected compile-time errors to compilation performance degradations. Correctness issues found in the core components of compiler typing processes, such as the type system, inference and resolution engines, and the semantic validation of declarations, are responsible for the majority of the inspected bugs. Moreover, although front-end bugs are typically fixed without requiring extensive modifications in compilers' code base, compilers' developer need a few months to resolve a bug, as they carefully assess the impact of each fix to prevent regression bugs. We also found that a non-trivial number of typing-related bugs is caused by non-compilable test cases, while loops and arithmetic expressions are hardly seen in the bug-revealing programs. These observations conflict with the intuition behind the existing approaches for finding optimization bugs.

We discussed several implications of our study's findings. Future testing techniques should consider diverse test oracles, as typing-related bugs affect both compilation and runtime in various ways. Another interesting future challenge is the generation of subtly invalid programs that are more likely to trigger typing-related bugs and, most notably, soundness issues. Furthermore, the existing program generators for C++ could benefit from the results of our study: their generation strategies could be adapted to include features (e.g., type inference, lambdas, overloading) that can potentially uncover inference or resolution bugs in the C++ compilers.

Finally, we implemented a simple program generator that relies on some of our observations regarding type inference and parametric polymorphism. Our generator was able to reveal 27 unexpected compile-time errors, and one internal compiler error in the Kotlin and Groovy compilers. This demonstrates the practicality of our study: we believe that researchers can build upon our work's findings by creating improved compiler validation methods and tools.

ABBREVIATIONS - ACRONYMS

AOBE	Array Index Out of Bounds Exception
NPE	Null Pointer Exception
CCE	Class Cast Exception

BIBLIOGRAPHY

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[2] Alexander Konovalov. Fuzzball: Scala fuzzer. `https://github.com/alexknvl/fuzzball`, 2021. Online accessed; 05-03-2021.

[3] N. Amin, S. Grütter, M. Odersky, T. Rompf, and S. Stucki. *The Essence of Dependent Object Types*, pages 249–272. Springer International Publishing, Cham, 2016.

[4] M. Bagherzadeh, N. Fireman, A. Shawesh, and R. Khatchadourian. Actor concurrency bugs: A comprehensive study on symptoms, root causes, api usages, and differences. *Proc. ACM Program. Lang.*, 4(OOPSLA), Nov. 2020.

[5] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, page 183–200, New York, NY, USA, 1998. Association for Computing Machinery.

[6] Brian Goetz. State of Valhalla. `https://cr.openjdk.java.net/~briangoetz/valhalla/sov/01-background.html`, 2020. Online accessed; 05-03-2021.

[7] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie. Learning to prioritize test programs for compiler testing. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, page 700–711. IEEE Press, 2017.

[8] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie. Test case prioritization for compilers: A text-vector based approach. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 266–277, 2016.

[9] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang. A survey of compiler testing. *ACM Comput. Surv.*, 53(1), Feb. 2020.

[10] Y. Chen, T. Su, and Z. Su. Deep differential testing of JVM implementations. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 1257–1268. IEEE Press, 2019.

[11] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 85–99, New York, NY, USA, 2016. Association for Computing Machinery.

[12] J. Davis, A. Thekumparampil, and D. Lee. Node.Fz: Fuzzing the server-side event-driven architecture. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 145–160, New York, NY, USA, 2017. Association for Computing Machinery.

[13] K. Dewey, J. Roesch, and B. Hardekopf. Fuzzing the Rust typechecker using CLP. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, page 482–493. IEEE Press, 2015.

[14] A. Di Franco, H. Guo, and C. Rubio-González. A comprehensive study of real-world numerical bug characteristics. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, page 509–519. IEEE Press, 2017.

[15] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct. 2017.

[16] S. Dutta, O. Legunsen, Z. Huang, and S. Misailovic. Testing probabilistic programming systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 574–586, New York, NY, USA, 2018. Association for Computing Machinery.

[17] Gavin Bierman. Pattern matching for instanceof. `https://openjdk.java.net/jeps/305`, 2017. Online accessed; 05-03-2021.

[18] Github Inc. The state of the Octoverse. `https://octoverse.github.com/`, 2021. Online accessed; 05-03-2021.

[19] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. The Java language specification: Java SE 8 edition. `https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf`, 2015.

[20] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, page 38, USA, 2012. USENIX Association.

[21] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 77–88, New York, NY, USA, 2012. Association for Computing Machinery.

[22] D. E. Knuth. The errors of TeX. *Software: Practice & Experience*, 19(7):607–687, July 1989.

[23] F. Křikava, H. Miller, and J. Vitek. Scala implicits are everywhere: A large-scale study of the use of Scala implicits in the wild. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019.

[24] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 216–226, New York, NY, USA, 2014. Association for Computing Machinery.

[25] V. Le, C. Sun, and Z. Su. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, page 386–399, New York, NY, USA, 2015. Association for Computing Machinery.

[26] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 517–530, New York, NY, USA, 2016. Association for Computing Machinery.

[27] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 65–76, New York, NY, USA, 2015. Association for Computing Machinery.

[28] V. Livinskii, D. Babokin, and J. Regehr. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.*, 4(OOPSLA), Nov. 2020.

[29] M. Marcozzi, Q. Tang, A. F. Donaldson, and C. Cadar. Compiler fuzzing: How much does it matter? *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019.

[30] B. Marick. A survey of software fault surveys. Technical Report 1651, Department of Computer Science. University of Illinois at Urbana-Champaign, 1990. Motorola Partnerships in Research.

[31] L. Mastrangelo, M. Hauswirth, and N. Nystrom. Casting about in the dark: An empirical study of cast operations in Java programs. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019.

[32] B. G. Mateus and M. Martinez. On the adoption, usage and evolution of Kotlin features in Android development. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20, New York, NY, USA, 2020. Association for Computing Machinery.

[33] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[34] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[35] A. Moors, F. Piessens, and M. Odersky. Generics of a higher kind. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA '08, page 423–438, New York, NY, USA, 2008. Association for Computing Machinery.

[36] E. Nagai, H. Awazu, N. Ishiura, and N. Takeda. Random testing of C compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*, pages 48–53, 2012.

[37] E. Nagai, A. Hashimoto, and N. Ishiura. Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions. *IPSJ Transactions on System LSI Design Methodology*, 7:91–100, 2014.

[38] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. 2004.

[39] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim. Fuzzing JavaScript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1629–1642, 2020.

[40] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 335–346, New York, NY, USA, 2012. Association for Computing Machinery.

[41] C. Sun, V. Le, and Z. Su. Finding and analyzing compiler warning defects. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 203–213, New York, NY, USA, 2016. Association for Computing Machinery.

[42] C. Sun, V. Le, and Z. Su. Finding compiler bugs via live code mutation. OOPSLA 2016, page 849–863, New York, NY, USA, 2016. Association for Computing Machinery.

[43] C. Sun, V. Le, Q. Zhang, and Z. Su. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 294–305, New York, NY, USA, 2016. Association for Computing Machinery.

[44] TIOBE Software BV. TIOBE index. `https://www.tiobe.com/tiobe-index/`, 2021. Online accessed; 05-03-2021.

[45] J. Wang, B. Chen, L. Wei, and Y. Liu. Superion: Grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 724–735. IEEE Press, 2019.

[46] J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei. A comprehensive study on real world concurrency bugs in Node.js. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, page 520–531. IEEE Press, 2017.

[47] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[48] Wikipedia contributors. Software testing, 2021. [Online; accessed 22-July-2021].

[49] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.

[50] Q. Zhang, C. Sun, and Z. Su. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 347–361, New York, NY, USA, 2017. Association for Computing Machinery.

[51] Z. Zhou, Z. Ren, G. Gao, and H. Jiang. An empirical study of optimization bugs in GCC and LLVM. *Journal of Systems and Software*, 174:110884, 2021.

[52] D. Zubrow. IEEE standard classification for software anomalies. *IEEE Computer Society*, 2009.