TU Delft

Delft University of Technology

Well-typed programs can go wrong
A study of typing-related bugs in JVM compilers

Chaliasos, Stefanos; Sotiropoulos, Thodoris; Drosos, Georgios Petros; Mitropoulos, Charalambos; Mitropoulos, Dimitris; Spinellis, Diomidis

DOI
10.1145/3485500

Publication date
2021

Document Version
Final published version

Published in
Proceedings of the ACM on Programming Languages

Citation (APA)
Chaliasos, S., Sotiropoulos, T., Drosos, G. P., Mitropoulos, C., Mitropoulos, D., & Spinellis, D. (2021). Well-typed programs can go wrong: A study of typing-related bugs in JVM compilers. *Proceedings of the ACM on Programming Languages*, *5*(OOPSLA), Article 123. https://doi.org/10.1145/3485500

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Well-Typed Programs Can Go Wrong: A Study of Typing-Related Bugs in JVM Compilers

STEFANOS CHALIASOS*, Athens University of Economics and Business, Greece
THODORIS SOTIROPOULOS*, Athens University of Economics and Business, Greece
GEORGIOS-PETROS DROSOS, Athens University of Economics and Business, Greece
CHARALAMBOS MITROPOULOS, Technical University of Crete, Greece
DIMITRIS MITROPOULOS, University of Athens, Greece
DIOMIDIS SPINELLIS, Athens University of Economics and Business, Greece and Delft University of Technology, Netherlands

Despite the substantial progress in compiler testing, research endeavors have mainly focused on detecting compiler crashes and subtle 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 (breaking code portability or the type system's soundness), the rejection of correct (e.g. well-typed) programs, and the reporting of misleading errors and warnings.

We conduct, what is to the best of our knowledge, the first empirical study for understanding and characterizing 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. Some representative observations indicate that: (1) more than half of the typing-related bugs manifest as unexpected compile-time errors: the buggy compiler wrongly rejects semantically correct programs, (2) the majority of typing-related bugs lie in the implementations of the underlying type systems and in other core components related to operations on types, (3) parametric polymorphism is the most pervasive feature in the corresponding test cases, (4) one third of typing-related bugs are triggered by non-compilable programs.

We believe that our study opens up a new research direction by driving future researchers to build appropriate methods and techniques for a more holistic testing of compilers.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Software testing and debugging**.

Additional Key Words and Phrases: compiler bugs, compiler testing, static typing, Java, Scala, Kotlin, Groovy

ACM Reference Format:
Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Well-Typed Programs Can Go Wrong: A Study of Typing-Related

*These authors contributed equally.

Authors' addresses: Stefanos Chaliasos, Athens University of Economics and Business, Greece, schaliasos@aueb.gr; Thodoris Sotiropoulos, Athens University of Economics and Business, Greece, theosotr@aueb.gr; Georgios-Petros Drosos, Athens University of Economics and Business, Greece, t8180024@aueb.gr; Charalambos Mitropoulos, Technical University of Crete, Greece, cmitropoulos@isc.tuc.gr; Dimitris Mitropoulos, University of Athens, Greece, dimitro@ba.uoa.gr; Diomidis Spinellis, Athens University of Economics and Business, Greece and Delft University of Technology, Netherlands, dds@aueb.gr.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2021 Copyright held by the owner/author(s).
2475-1421/2021/10-ART123
https://doi.org/10.1145/3485500

Bugs in JVM Compilers. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 123 (October 2021), 30 pages. https://doi.org/10.1145/3485500

Well-typed programs cannot "go wrong". — Robin Milner [1978]

1 INTRODUCTION

Over the past decade, we have witnessed tremendous advances in techniques for improving compiler reliability. Dozens of methods have emerged to validate compilers' correctness or facilitate compiler testing and debugging: from program generators [Livinskii et al. 2020; Nagai et al. 2012, 2014; Yang et al. 2011] and transformation-based techniques [Le et al. 2014, 2015; Sun et al. 2016b; Zhang et al. 2017], to test-case reduction [Regehr et al. 2012] and test-case prioritization approaches [Chen et al. 2017, 2016a]. Although the initial focus was on C/C++ compilers, researchers have also invested much effort on testing other compilers [Dewey et al. 2015; Donaldson et al. 2017; Lidbury et al. 2015], runtime systems [Chen et al. 2019, 2016b], and even dynamic programming languages [Holler et al. 2012; Park et al. 2020; Wang et al. 2019]. 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, Nagai et al. [2012, 2014] craft C programs that exercise optimizations on arithmetic expressions. Another example is the most recent program generator for C/C++ programs [Livinskii et al. 2020], 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 [Moors et al. 2008], parametric polymorphism, or path-dependent types [Amin et al. 2016]), 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 a compiler to effectively deal with certain programs leading to type-safety breaches and compilation of non-portable code, or propagate themselves to other compiler phases.

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.

RQ1 (Symptoms) What are the main symptoms of typing-related compiler bugs? What is the frequency of these symptoms? (Section 3.1)

RQ2 (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 3.2)

RQ3 (Bug Fixes) How are typing-related compiler bugs introduced? What is the size of their fixes? How long does it take to fix these bugs? (Section 3.3)

RQ4 (Test Case Characteristics) What are the main characteristics of the bug-revealing test cases? What language features are prevalent in these test cases? (Section 3.4)

To answer these 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 [Bracha et al. 1998]. 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 [Github Inc. 2021; TIOBE Software BV 2021]. The Scala programming language [Odersky et al. 2004] 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 path-dependent 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 [Mateus and Martinez 2020]. Finally, Groovy is a popular programming language [TIOBE Software BV 2021] that supports both dynamic and static typing, and also provides flow-sensitive typing.

Using carefully-crafted search criteria and some heuristics, we search the issue trackers of the studied languages and obtain 4,101 previously reported typing-related bugs that have been *fixed*. We analyze a random sample of 320 bugs. Specifically, 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.

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 2).
- 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 3).
- We enumerate the implications of our findings, and discuss potential future directions on compiler testing (Section 4).

Summary of findings. Some of our representative findings are: (1) most of fixed typing-related bugs (50.94%) manifest as unexpected compile-time errors: the buggy compiler mistakenly rejects correct programs, (2) the majority of fixed 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: 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.

Implications. 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 Kotlin and Groovy compilers' front-end. Our program generator was able to find 28 previously unknown bugs within two months of testing. More than 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.

Availability. The research artifact is available at https://zenodo.org/record/5411667.

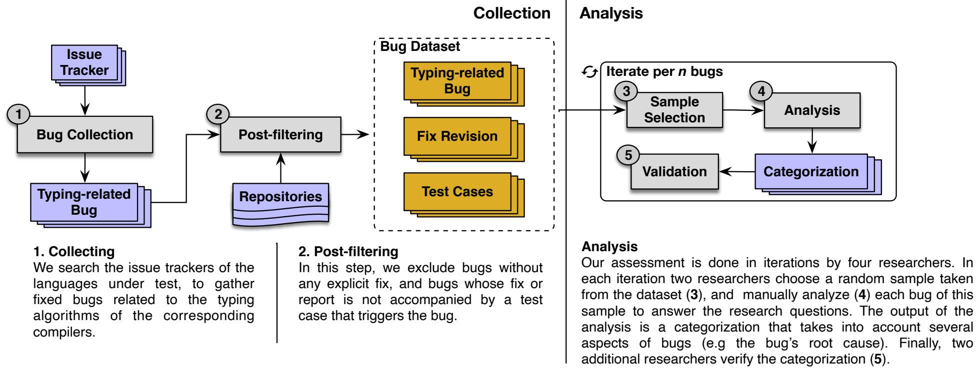


Fig. 1. The overview of our bug collection and bug analysis approach.

2 METHODOLOGY

First, we create a corpus of typing-related bugs taken from the issue trackers of four JVM languages (Section 2.1). Then, we explain how we study and analyze the collected bugs (Section 2.2), and finally, we discuss the limitations and threats to validity of our method (Section 2.3).

Our bug collection and analysis approach is summarized in Figure 1. As a starting point, we take the issue trackers of the languages under study, and we apply language-specific filters in order to obtain an initial list of typing-related bugs (*bug Collection*). Then, in the next step (*post-filtering*), we filter out typing-related bugs that are not accompanied by an explicit fix and a test case. To do so, we search the repositories and the issue trackers of the examined languages for commits, pull requests or bug reports that are linked with any of the bugs included in the output of the previous step. The *bug collection* and *post-filtering* steps are fully automated and constitute our approach for collecting bugs and their fixes (Section 2.1). The final outcome of this approach is a bug dataset consisting of a set of typing-related bugs, their fix revisions, and their test cases.

The resulting dataset is used as an input to our bug analysis approach (Section 2.2). This bug analysis is done in iterations by four researchers. Each iteration involves the examination of a specified number of bugs, n. Specifically, two researchers first choose a random sample of bugs taken from the initial dataset (*sample selection*), and then they manually analyze each bug of this sample to answer our research questions (RQ3 is partly answered through automated means — see Section 3.3). The output of the analysis is a categorization that takes into account several aspects of bugs, including their symptom, root cause, and test case characteristics. Finally, two additional researchers verify the proposed categorization. In case of a conflict, they discuss with the original researchers until reaching consensus (*validation*).

2.1 Collecting Bugs and Fixes

Our bug collection approach consists of two steps, namely, *bug collection* and *post-filtering*. In the first step, 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 proceed as follows. 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

Table 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 creation date of the oldest and the most recent issue considered in the study (Oldest and Most Recent), (3) the number of the selected bugs after running the *bug collection* step (BC), and (4) the number of the remaining bugs after running the *post-filtering* step (PF).

Language	Issue Tracker	REST Endpoint	Total Issues	Oldest	Most Recent	BC	PF
Java	Jira	https://bugs.openjdk.java.net/rest/api/latest/search	10,872	11 Feb 2004	26 March 2021	1,252	873
Scala 2	GitHub	https://api.github.com/repos/scala/bug	12,315	22 May 2003	11 March 2021	1,180	1,067
Scala 3	GitHub	https://api.github.com/repos/lampepfl/dotty	4,286	1 Feb 2014	28 March 2021	429	366
Kotlin	YouTrack	https://youtrack.jetbrains.com/api/issues	40,998	28 Oct 2011	9 April 2021	2,189	1,601
Groovy	Jira	https://issues.apache.org/jira/rest/api/2/search	9,710	25 Sep 2003	9 April 2021	300	246

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/`. 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* step, 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) along with the corresponding fix revisions and test cases.

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 1 shows descriptive statistics of our bug collection effort. After applying the *bug collection* and *post-filtering* steps 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 Kotlin compiler (`kotlinc`), and 246 bugs are in Groovy compiler (`groovyc`). Note that although not all the examined programming languages used the same issue tracker throughout their lifetime (e.g., Scala recently migrated to GitHub from Jira [Tisue 2017]), we were able to consider bugs even from the early days of these languages, as all these historical issues were imported to the current issue trackers. In the following, we discuss some language-specific details related to our bug collection approach.

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 292,059 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.

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. 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”, “typelevel”, “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 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.

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 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* step, we ended up with the final set of Kotlin bugs $\mathcal{B}'_k$. This set contains 1,601 elements.

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

2.2 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.

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 (*sample selection*, Figure 1), 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 began. The first two authors independently studied each of the selected 80 bugs (*analysis*), 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). In particular, regarding RQ1, the authors considered the description of each bug report to identify differences between the compiler's expected and actual behavior. For RQ2, the authors studied the discussion among developers and the fix of each bug to locate which specific compiler procedure was buggy, while for answering RQ4, examining the accompanying test cases was enough.

Moving forward, for each bug, the first two authors together discussed their categorization, until they reached consensus. The procedure described above was repeated four times, i.e., until studying 320 bugs in total. During these four iterations, the first two authors 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. Finally, two additional researchers verified the resulting categorization and discussed their conflicts with the initial authors until agreement. Notably, having this extra validation step was beneficial in cases where the first two authors could not reach consensus. In such cases, there was a discussion during validation, where the researchers considered all possible options, and finally agreed to the most "optimal" one.

2.3 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. Furthermore, our study considered only real bugs rather than enhancements or features (e.g., situations where the typing algorithm is incomplete and can be further improved). To do so, during the selection process, we chose issues explicitly marked with the label "bug" (recall Section 2.1), and avoided issues marked as "enhancement" or "feature". This is in line with prior work [Di Franco et al. 2017; Jin et al. 2012; Sun et al. 2016c], where fixed bugs were also studied.

For selecting bugs related to the typing algorithms of compilers, we carefully 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 by the first two authors, who excluded irrelevant ones.

A threat to external validity is the representativeness of the selected bugs. To mitigate this threat, we picked a random sample of 320 typing-related bugs, which is consistent with the literature of bug studies. Specifically, Jin et al. [2012] have manually analyzed 110 real-world performance bugs, Di Franco et al. [2017] analyzed 269 bugs in numerical libraries, Leesatapornwongsa et al. [2016], and Bagherzadeh et al. [2020] analyzed 104 and 186 concurrency bugs in distributed and actor-based systems respectively. Theoretically, as in the work of Mastrangelo et al. [2019], we can presume that using a random sample of 320 bugs, there is a probability of $(1 - 0.01)^{320} \simeq 0.04 = 4\%$ of missing a category whose relative frequency is at least 1%, and a probability of $(1 - 0.015)^{320} \simeq 0.008 = 0.8\%$ of missing a category whose relative frequency is at least 1.5%. Note that in practice, as we were examining bugs in iterations, we observed that it was easy to categorize bugs coming from the third or the fourth iteration, as most of these bugs fitted well in one of the resulting categories.

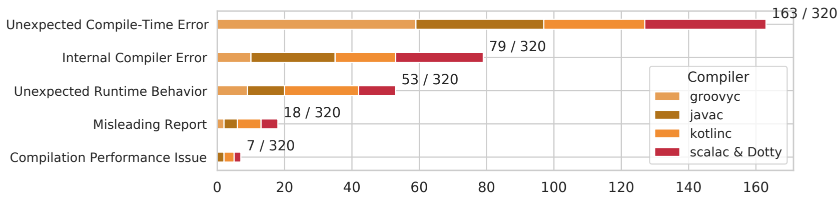


Fig. 2. The distribution of symptoms.

Another 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. 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.

Our manual analysis of bugs may be subjective. To minimize this threat, the first two authors independently studied each bug, and then they had a thorough discussion until agreeing on categorization. To further mitigate this threat, two additional researchers verified the categorization performed by the first two authors. This is consistent with previous empirical studies [Bagherzadeh et al. 2020; Di Franco et al. 2017; Wang et al. 2017], where each bug was inspected and verified by multiple researchers.

3 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.

3.1 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. We manually examined the differences between the compiler's expected and actual behavior, and grouped these differences into categories. We ultimately identified five categories of symptoms, namely, *Unexpected Compile-Time Error*, *Internal Compiler Error*, *Unexpected Runtime Behavior*, *Misleading Report*, and *Compilation Performance Issue*. Figure 2 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.

3.1.1 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.

Example bug: KT-10711

Figure 3 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, 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`.

```
1  class A<T>(val f: T)
2  fun test() {
3    listOf<String>().map(::A)
4  }
```

Fig. 3. KT-10711: A program that triggers a kotlinc bug with an *unexpected compile-time error.*

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).

3.1.2 Internal Compiler Error (or Crash). This 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.

Example bug: GROOVY-7618

Figure 4 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`.

```
1  interface I {
2    int m()
3  }
4  int m2(I x) {
5    x.m()
6  }
7  void test() {
8    m2 { -> 1 }
9  }
```

Fig. 4. GROOVY-7618: A program that triggers a groovyc bug with an *internal compiler error.*

`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. While coercing the type of lambda to a SAM type for type checking the call at line 8, `groovyc` first computes the arity of lambda by accessing the property `params.length`. 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 occur because of the following reasons: (1) operations on unvalidated data that trigger unexpected runtime exceptions (e.g., `NullPointerException`, `ArrayOutOfBoundsException`, `ClassCastException`), (2) failures of assertions included in the compiler's source code or custom exceptions thrown when the compiler validates input data that are in an illegal state, and (3) infinite loops in recursive computations leading to a `StackOverflowException`. We found that 35 internal compiler errors were triggered by unexpected runtime exceptions, 35 by assertion failures and custom compiler exceptions, and 9 by infinite loops. As an example of an assertion failure, consider Dotty-7041, where 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 the JVM. This is not the case in this issue, where after type erasure the program tree contains an illegal type leading to an `AssertionError`.

3.1.3 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 [Milner 1978]. 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.

Example bug: JDK-7041019

Consider the Java program of Figure 5, 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 (lines 12–14).

```
1  interface A<E> {
2    void m(E x);
3  }
4  interface B<Y> extends A<Y[]> { }
5  class C implements B<Integer> {
6    @Override
7    void m(Integer[] x) { }
8    static <T extends B<?>> void m2(T x) {
9      //Boom! ClassCastException at runtime.
10     x.m(new String[]{"s"});
11   }
12   static void main(String[] args) {
13     m2(new C());
14   }
15 }
```

Fig. 5. JDK-7041019: A program that triggers a `javac` bug with an *unexpected runtime behavior.*

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

is because the 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 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.

3.1.4 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 message. 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.

Example bug: KT-5511

Figure 6 shows a Kotlin program triggering a bug (see KT-5511) with a *misleading report* 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>`.

```
1  interface X<T> {
2      inner enum class C : X<T>
3  }
```

Fig. 6. KT-5511: A program that triggers a `kotlinc` bug with a *misleading report.*

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. However, the second error of the compiler is spurious, and it is caused by a bug in the reporting mechanism of `kotlinc`.

3.1.5 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.

Example bug: Dotty-10217

Consider the Scala program of Figure 7, which triggers a bug in Dotty (see Dotty-10217). The program defines 23 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`.

```
1 trait A
2 trait B
3 trait C
4 ...
5 trait W
6 trait Foo[T]
7 val f: Foo[A | B | C | ... | W] = ???
```

Fig. 7. Dotty-10217: A program that triggers a Dotty bug with an *compilation performance issue.*

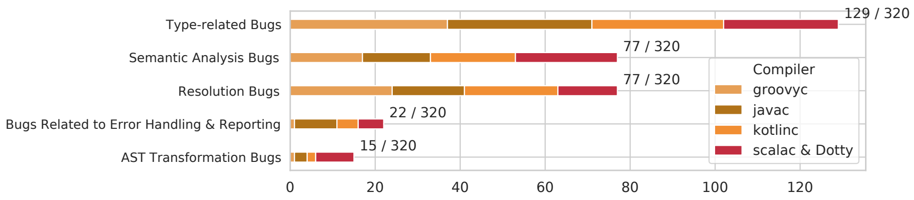
Dotty spends roughly five minutes to compile this program. Specifically, Dotty performs a type optimization on union types: a union type of the form `T | Null` or `Null | T` becomes a regular type `T`. In this context, Dotty examines the union type passed as a type argument of type constructor `Foo` (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 23 terms, Dotty performs 2^{23} calls to `derivesFrom`!

3.1.6 Comparative Analysis. From Figure 2, 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, which may be attributed to Kotlin's immaturity compared to the other compilers. 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.

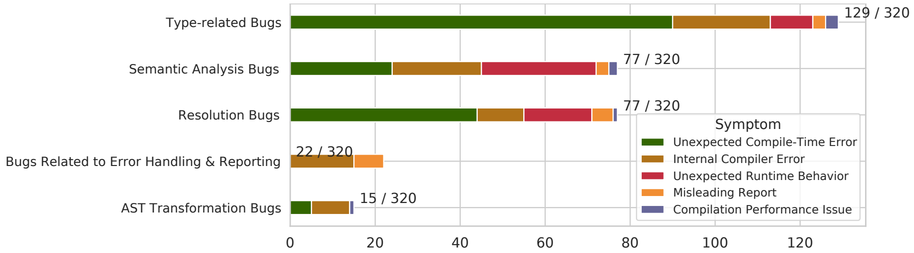
3.2 RQ2: Bug Causes

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*, *Bugs Related to Error Handling & Reporting*, and *AST Transformation Bugs*. Figure 8 illustrates the distribution of our bug causes. In the following, we provide descriptions and examples for every category.

3.2.1 Type-Related Bugs. To type check an input program, a compiler consults the type system of the language, which provides a set of rules of what are the language main types, what operations on these types are valid, how these types relate to each other, and how they can be combined. In this context, a compiler internally represents all types and properties of the underlying type system using specialized data structures. Further, when typing an input program, it applies a



(a) The distribution of bug causes per compiler.



(b) The distribution of bug causes per symptom.

Fig. 8. The distribution of bug causes.

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 8b). 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 groups: (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 the type at all.

Figure 3 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 Figure 5. 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 or type erasure. For example, as shown in Figure 7, 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. 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.

Example bug: KT-9630

As an example, consider Figure 9, 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`.

```
1 interface A
2 interface B
3 class C : A, B
4 fun <T> T.m(): Unit where T : A, T : B { }
5 fun main() {
6   C().foo()
7 }
```

Fig. 9. KT-9630: A Kotlin program that triggers a bug related to an *incorrect type transformation.*

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 included in the type system. For example, a compiler consults the subtyping rules of the type system to check whether a value of type $\mathtt{T}_1$ is assignable to a variable of type $\mathtt{T}_2$. 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.

Example bug: JDK-8039214

Figure 10 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 a 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) [Gosling et al. 2015, §4.1.5]. Unfortunately, the implementation of `javac` does not follow this containment relation to the letter. Hence, it considers that `C<?>` is not a subtype of `I<? extends X, X>`. This makes `javac` reject this well-formed program.

```
1 interface I<X1,X2> {}
2 class C<T> implements I<T,T> {}
3
4 public class Test {
5   <X> void m(I<? extends X, X> arg) {}
6   void test(C<?> arg) {
7     m(arg);
8   }
9 }
```

Fig. 10. JDK-8039214: A Java program that triggers a bug related to *incorrect type comparisons.*

3.2.2 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 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.

Example bug: Scala2-5878

Consider the Scala program of Figure 11, 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.

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

Fig. 11. Scala2-5878: A Scala program that triggers a bug related to *missing validation checks*.

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.

3.2.3 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 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 8b.

Example bug: JDK-7042566

Figure 12 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 [Gosling et al. 2015, §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.

```
1  class Test {
2    void test() {
3      Exception ex = null;
4      error("error", ex);
5    }
6    void error(Object o, Object... p) { }
7    void error(Object o, Throwable t,
8               Object... p) { }
9  }
```

Fig. 12. JDK-7042566: A Java program that triggers a *resolution* bug.

3.2.4 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 Figure 6 triggers a bug related to error handling and reporting. As already discussed, in this program, kotlinc produces an excessive diagnostic message. This message suggests developers to take actions that contradict with previously reported messages.

3.2.5 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 that employs iterators as follows: `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.

Example bug: Scala2-6714

Figure 13 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).

```
1  class A
2  class B {
3    def apply(x: Int)(implicit a: A) = 1
4    def update(x: Int, y: Int) { }
5  }
6  object Test {
7    implicit val a = new A()
8    val b = new B()
9    b(3) += 4 // compile-time error here
10 }
```

Fig. 13. Scala2-6714: A Scala program that triggers an *AST transformation bug*.

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)`. 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.

As already discussed in Section 2.3, we may have missed the identification of categories not included in the sample of analyzed bugs. For example, such a missing category may involve bugs concerning program serialization, a process that translates the AST of a program into another storable format, e.g., see the TASTy files of Dotty.[1] Such representations are useful for inspecting the semantic information of the input program and designing custom program analyses.

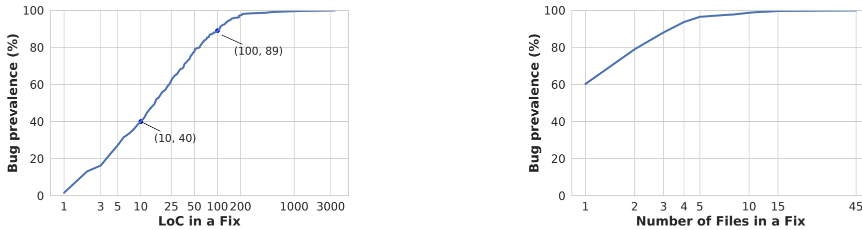
3.2.6 Comparative Analysis. According to Figure 8a, type-related bugs form by far the most common bug cause for all studied compilers. This suggests that reasoning about types is a complex and challenging task for compilers, and that the corresponding type system implementations are susceptible to errors. 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.

3.3 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.

3.3.1 How Are Bugs 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.

[1] https://docs.scala-lang.org/scala3/reference/metaprogramming/tasty-inspect.html



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

(b) Cumulative distribution of files in a fix.

Fig. 14. Size of bug fixes.

Logic errors, which stand for defects in logic, sequencing, or branching of a procedure [Zubrow 2010], are the dominant source of bugs in our dataset (204 / 320). Most of these logic errors are missing cases or steps in the implementation of a routine, or incorrect conditions of an `if` statement. According to Groovy developers, such missing cases are introduced by failing to handle some edge cases (e.g., due to insufficient test coverage) when adding a new compiler feature or making an enhancement. 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 `else if` 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 (Figure 7), in which the implementation of the underlying algorithm has exponential complexity. Algorithmic errors were found in 67 / 320 bugs.

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 the == operator), 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. We encountered 36 / 320 bugs introduced by an error in the compiler/language design.

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 Figure 4 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. We ran into such programming errors in only 13 / 320 bugs.

3.3.2 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 14a 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. [2016c], which 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 14b shows how many files are modified in a bug fix. The majority of fixes change 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.

3.3.3 Duration of Bugs. Considering the plots in Figure 14, 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 and 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.

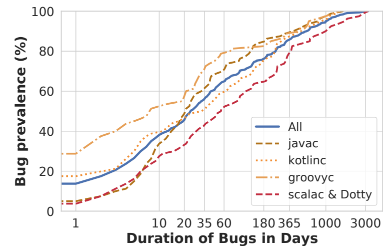


Fig. 15. Cumulative distribution of bugs through time.

Figure 15 shows the bugs' cumulative distribution function over time. The blue plot indicates that over 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 came across 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.

3.3.4 Comparative Analysis. Consider again Figure 15. `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 some parts of `groovyc` (e.g., the type checker) may be less mature than the other compilers, and many `groovyc` bugs are programming errors (e.g., a GROOVY-7618, Figure 4), which can be fixed easily (e.g., by adding a `null` check), rather than defects that require much domain expertise and knowledge. Another

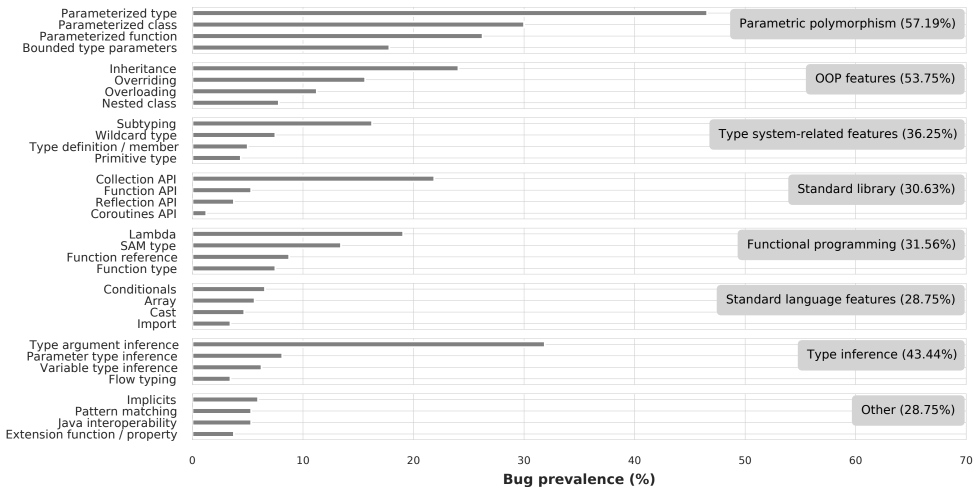


Fig. 16. The classification of the language features that appear in test cases, along with their frequency. For each category, we show the four most frequent features.

explanation may lie in the motivation and resources associated with the project's development team.

We also performed the Mann-Whitney U test on the distributions of bugs' duration. 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.

3.4 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.

3.4.1 General Statistics. Table 2 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 [Le et al. 2014; Livinskii et al. 2020; Yang et al. 2011]), identifying typing-related bugs requires passing *non-compilable* 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.

Table 2. 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)	8.0
Number of class decl. (mean)	2.0
Number of class decl. (median)	2.0
Number of method decl. (mean)	2.9
Number of method decl. (median)	2.0
Number of method call (mean)	2.5
Number of method call (median)	1.0

3.4.2 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,

Table 3. 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	30.00%	Arithmetic expression	1.88%
Parameterized function	26.25%	Loops	1.25%
Inheritance	24.06%	Sealed class	0.94%

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).

For every category of features, Figure 16 presents its frequency along with its four most frequent features. Parametric polymorphism is pervasive in the corresponding bug-revealing programs: more than half of the examined bugs (57.19%) are caused by test cases 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 Figure 3. 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 3 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 [Le et al. 2014; Livinskii et al. 2020] 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 [Jin et al. 2012], 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 3. Moreover, such test cases often let the compiler infer some type information, e.g., signature of lambda expressions, or type argument of a callee parameterized function. Regarding individual features, some interesting combinations are: (1) variable arguments with overloaded methods (e.g., Figure 12) with a lift score of 24, (2) use-site variance with parameterized function (e.g., Figure 5) with a lift score of 17.1, (3) type argument

Table 4. 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	57.50	Parameterized type	36.25	Parameterized type	41.25
Type argument inference	42.50	Parameterized class	42.50	Parameterized class	33.75	Collection API	35.00
Functional interface	37.50	Inheritance	32.50	Type argument inference	32.50	Type argument inference	35.00
Parameterized function	35.00	Implicits	23.75	Parameterized function	26.25	Lambda	25.00
Parameterized class	30.00	Parameterized function	22.50	Inheritance	25.00	Parameterized function	21.25

inference with parameterized function (e.g., Figures 5, 9) 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., Figure 3) 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 [Mastrangelo et al. 2019]. 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 4.2).

3.4.3 Comparative Analysis. Table 4 shows the five most bug-triggering features per language. Again, parametric polymorphism-related features are in the top of every language under study. Another interesting finding is that implicits, a powerful and popular Scala-only feature [Křikava et al. 2019], 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.

4 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. We also demonstrate the value of our study's results through a type-related Kotlin and Groovy test-program generator.

4.1 Lessons Learned and Takeaways

Typing-related bugs have diverse manifestations. Contrary to optimization bugs, which mainly manifest at runtime as errors [Le et al. 2014; Yang et al. 2011], typing-related bugs can potentially affect both compilation and runtime (Section 3.1). 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 14a and 14b (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 3.2, 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 Figure 11.

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 2). This observation comes in contrast to existing compiler testing techniques, which feed compilers with compilable programs [Le et al. 2014; Livinskii et al. 2020; Yang et al. 2011]. 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)* [Zhang et al. 2017] 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 half of the examined bugs (Tables 3, 4). 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. Consequently, 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 3.4.2). 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 Figure 3, line 3). Interestingly, such APIs heavily rely on parametric polymorphism.

Control-flow constructs and arithmetic expressions are not common in bug-revealing test cases. Table 4 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 [Livinskii et al. 2020; Nagai et al. 2014; Yang et al. 2011]. For example, as an effort to uncover optimization bugs, the recent work of Livinskii et al. [2020] 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. [2019], 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

```
1  def gen_program(builtins, max_decls, max_fields,
2                max_methods, max_params)
3      decls = []
4      type_pool = builtins
5      for i in range(randint(1, max_decls)):
6        decl_type = random(
7            ["FUNC", "CLASS", "VAR"])
8        if decl_type == "CLASS":
9          cls = gen_class(max_fields, max_methods,
                max_params)
10         type_pool.append(get_type(cls))
11         decls.append(cls)
12       elif decl_type == "FUNC":
13         func = gen_func(max_params)
14         decls.append(func)
15       else:
16          var_type = random(type_pool)
17          expr = gen_expr(var_type)
18          decls.append(create_var(var_type, expr))
```

```
19 def gen_class(max_fields, max_methods, max_params
       ):
20     fields = []
21     methods = []
22     for i in range(randint(1, max_fields)):
23       field_type = random(type_pool)
24       fields.append(create_field(field_type))
25     for i in range(randint(1, max_methods)):
26       methods.append(gen_func(max_params))
27     return create_cls(fields, funcs)
28
29 def gen_func(max_params):
30     ret_type = random(type_pool)
31     params = []
32     for i in range(randint(1, max_params)):
33       param_type = random(type_pool)
34       params.append(create_param(param_type))
35     body = gen_expr(ret_type)
36     return create_func(params, ret_type, body)
```

Fig. 17. The high-level description of our program generation approach.

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.

4.2 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 3.4.2).

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.

Our algorithm for generating programs written in the IR is summarized in Figure 17. Every program is *well-formed* by construction and consists of a number of randomly generated declarations (i.e., classes, functions or variables), see lines 5–18. Each class declaration includes a random number of methods and fields (lines 19–27), while each function declaration contains a signature (i.e., a list of formal parameters and a return type — see lines 30–34), and a body corresponding to an expression whose type is a subtype of function's return type (line 35).

During generation, we randomly assign types to the signature of functions (e.g., return type, formal parameter type — lines 30, 33), or the signature of class fields (line 23). 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 (line 10). Note that the type pool is initialized with the set of built-in types (line 4). This set is given as input to our generation algorithm and contains the set of built-in types (e.g., `Int`, `String`, `Boolean`) supported by the language under test. Regarding expressions (e.g., variable

Table 5. 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**

```
1  class A<T> {
2    T x;
3    A(T x) { this.x = x; }
4    void test() {
5      m((new A<>("str")).x) // error here
6    }
7    void m(String x) {}
8  }
```

(a) The program that triggers the bug.

```
test.groovy: 5: [Static type checking] -
Cannot find matching method A.m(T).
Please check if the declared type is correct
and if the method exists.

 @ line 5, column 5.
        m((new A<>("str")).x);
               ^1 error
```

(b) The incorrect output of the compiler.

Fig. 18. GROOVY-9963: A Groovy bug found by our program generator. This bug was fixed in Groovy 4.0.

references, function calls, assignments, etc.), the algorithm generates a random expression whose type is a subtype of the given 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 kotlinc and groovyc. 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 then passed as an input to the compiler under test. Generating programs that are well-formed by construction gives us the test oracle: 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 detected.

Table 5 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. 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 future techniques on compiler testing, which (1) are useful for finding typing-related bugs, (2) are applicable to more than one compilers.

As an example of bug found by our tool, consider the Groovy program of Figure 18a (minimized for space reasons). The program defines a parameterized class A that expects one type parameter T (lines 1–8). This class defines a field called x whose type is given by T (line 2). The code later initializes an instance of class A, and passes the value of the field x as an argument to a function named m (line 5). Although this program is type-correct, groovyc rejects it by producing the error message shown in Figure 18b. This is caused by a bug in the type substitution algorithm of groovyc. Specifically, the compiler fails to substitute the type variable corresponding to the type of field x with the concrete type String, with which the type constructor A is instantiated during the constructor call A<>("str").

Remark. Our program generator does not come with a test-case reducer. This is because our program generator mainly found unexpected compile-time errors. Therefore, by inspecting the *incorrect* compiler message, it was easy for us to locate the specific program expressions that triggered the bug (e.g., see the error message of Figure 18b). Apparently, for other kinds of symptoms

(e.g., crashes), manually reducing the test case is a tedious task. Since test-case reduction is beyond the scope of this paper, we leave this as future work.

5 RELATED WORK

Understanding Compiler Bugs. The closest bug study to our work is that conducted by Sun et al. [2016c]. The authors collected and automatically analyzed 52,732 bugs and 31,399 revisions from the GCC and LLVM compilers. 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. [2021] recently repeated the study of Sun et al. [2016c], but this time, 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. [2016c]. In a different spirit, Marcozzi et al. [2019] 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. It provides the first insights into understanding the nature of typing-related bugs, a category of bugs that is currently overlooked.

Other Bug Studies. Here we briefly present recent studies that are closely related to our work. Trying to investigate the characteristics of distributed concurrency bugs, Leesataporнwongsa et al. [2016] 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. Bagherzadeh et al. [2020] 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.

Numerical bugs form another category of bugs that has been examined by previous empirical studies. Di Franco et al. [2017] selected 269 numerical bugs from five popular numerical libraries and classified them into four categories based on their patterns and root causes. In a subsequent study, Dutta et al. [2018] 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. [2012] 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.

Compiler Testing. Testing compilers through randomized testing is a research topic that has received much attention (see the survey of Chen et al. [2020] for more details). Csmith [Yang et al. 2011] is the most well-known program generator for C programs. A characteristic of Csmith is that it generates programs that are free from undefined behavior. Thanks to Csmith hundreds of bugs in GCC and LLVM compilers have been uncovered. Since then, more program generators [Livinskii et al. 2020; Nagai et al. 2014] for C/C++ programs have been introduced with a focus on detecting optimization bugs. Following C's success, researchers have developed fuzzing tools and program generators for testing (1) other compilers such as OpenCL [Lidbury et al. 2015], graphics shader compilers [Donaldson et al. 2017], (2) runtime systems [Chen et al. 2019, 2016b], and (3) dynamic programming languages, such as PHP and JavaScript [Holler et al. 2012; Park et al. 2020; Wang et al. 2019]. Some of these tools have also been used in a production setting [Donaldson et al. 2020].

Most of the generators and fuzzers above target compiler crashes. For finding other functional errors (i.e., miscompilations), researchers have applied *differential testing* and *equivalence modulo inputs (EMI)* testing in combination with program generators. Differential testing has been successfully applied to C/C++ compilers [Livinskii et al. 2020; Sun et al. 2016a; Yang et al. 2011], JVM implementations [Chen et al. 2019, 2016b] and OpenCL compilers [Lidbury et al. 2015]. Beyond C compilers [Le et al. 2014], EMI testing has been recently applied to finding bugs in simulation software [Chowdhury et al. 2020]. Another interesting method for testing optimizing compilers is *skeletal program enumeration (SPE)* [Zhang et al. 2017]. SPE enumerates all programs that expose different variable usage patterns, while preserving the given program structure.

Currently, there is a research gap in automated testing of compiler front-ends. Although "traditional" fuzzing techniques, such as the AFL-based (American Fuzzy Lop) methods introduced in [M. Zalewski 2013; Wang et al. 2019], are effective in testing lexers and parsers, they struggle to construct semantically-valid inputs that get past the very front-end of a compiler and exercise the implementation of typing procedures. Dewey et al. [2015] have proposed a method for finding bugs in the type checker of Rust. Using constraint logic programming, they generate well-typed (or ill-typed) programs to detect various types of bugs (e.g., precision or soundness bugs). Sun et al. [2016a] have employed differential testing to check how different are the warnings produced by the GCC and LLVM compilers. We argue that our work can assist researchers in designing new testing methods, specifically targeting compiler front-ends.

6 CONCLUSION

We presented the first empirical study of 320 fixed 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. We also found that a non-trivial number of typing-related bugs is caused by non-compilable test cases, whereas 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.

ACKNOWLEDGMENTS

We thank Alex Delis and the anonymous reviewers for their constructive comments. We also thank the Groovy developers, Eric Miles and Paul King, for fixing our bug reports quickly and providing feedback in an earlier draft of this paper. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 825328.

REFERENCES

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. *The Essence of Dependent Object Types*. Springer International Publishing, Cham, 249–272. https://doi.org/10.1007/978-3-319-30936-1_14

Mehdi Bagherzadeh, Nicholas Fireman, Anas Shawesh, and Raffi Khatchadourian. 2020. Actor Concurrency Bugs: A Comprehensive Study on Symptoms, Root Causes, API Usages, and Differences. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 214 (Nov. 2020), 32 pages. https://doi.org/10.1145/3428282

Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. 1998. 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* (Vancouver, British Columbia, Canada) *(OOPSLA '98)*. Association for Computing Machinery, New York, NY, USA, 183–200. https://doi.org/10.1145/286936.286957

Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to Prioritize Test Programs for Compiler Testing. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 700–711. https://doi.org/10.1109/ICSE.2017.70

Junjie Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie. 2016a. Test Case Prioritization for Compilers: A Text-Vector Based Approach. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 266–277. https://doi.org/10.1109/ICST.2016.19

Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1, Article 4 (Feb. 2020), 36 pages. https://doi.org/10.1145/3363562

Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep Differential Testing of JVM Implementations. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 1257–1268. https://doi.org/10.1109/ICSE.2019.00127

Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016b. Coverage-Directed Differential Testing of JVM Implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 85–99. https://doi.org/10.1145/2908080.2908095

Shafiul Azam Chowdhury, Sohil Lal Shrestha, Taylor T. Johnson, and Christoph Csallner. 2020. SLEMI: Equivalence modulo Input (EMI) Based Mutation of CPS Models for Finding Compiler Bugs in Simulink. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 335–346. https://doi.org/10.1145/3377811.3380381

Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 482–493. https://doi.org/10.1109/ASE.2015.65

Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A Comprehensive Study of Real-World Numerical Bug Characteristics. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 509–519. https://doi.org/10.1109/ASE.2017.8115662

Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 93 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133917

Alastair F. Donaldson, Hugues Evrard, and Paul Thomson. 2020. Putting Randomized Compiler Testing into Production (Experience Report). In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 22:1–22:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.22

Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. 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* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 574–586. https://doi.org/10.1145/3236024.3236057

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

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2015. The Java Language Specification: Java SE 8 Edition. https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Bellevue, WA) *(Security'12)*. USENIX Association, USA, 38.

Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-World Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/2254064.2254075

Filip Křikava, Heather Miller, and Jan Vitek. 2019. Scala Implicits Are Everywhere: A Large-Scale Study of the Use of Scala Implicits in the Wild. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 163 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360589

Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216–226. https://doi.org/10.1145/2594291.2594334

Vu Le, Chengnian Sun, and Zhendong Su. 2015. 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* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 386–399. https://doi.org/10.1145/2814270.2814319

Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. 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* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 517–530. https://doi.org/10.1145/2872362.2872374

Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-Core Compiler Fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 65–76. https://doi.org/10.1145/2737924.2737986

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (Nov. 2020), 25 pages. https://doi.org/10.1145/3428264

M. Zalewski. 2013. American fuzzy lop. https://lcamtuf.coredump.cx/afl/. Online accessed; 05-08-2021.

Michaël Marcozzi, Qiyi Tang, Alastair F. Donaldson, and Cristian Cadar. 2019. Compiler Fuzzing: How Much Does It Matter? *Proc. ACM Program. Lang.* 3, OOPSLA, Article 155 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360581

Luis Mastrangelo, Matthias Hauswirth, and Nathaniel Nystrom. 2019. Casting about in the Dark: An Empirical Study of Cast Operations in Java Programs. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 158 (Oct. 2019), 31 pages. https://doi.org/10.1145/3360584

Bruno Gois Mateus and Matias Martinez. 2020. 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)* (Bari, Italy) *(ESEM '20)*. Association for Computing Machinery, New York, NY, USA, Article 15, 12 pages. https://doi.org/10.1145/3382494.3410676

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375.

Adriaan Moors, Frank Piessens, and Martin Odersky. 2008. Generics of a Higher Kind. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications* (Nashville, TN, USA) *(OOPSLA '08)*. Association for Computing Machinery, New York, NY, USA, 423–438. https://doi.org/10.1145/1449764.1449798

Eriko Nagai, Hironobu Awazu, Nagisa Ishiura, and Naoya Takeda. 2012. Random testing of C compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*. 48–53.

Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. 2014. 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 (2014), 91–100. https://doi.org/10.2197/ipsjtsldm.7.91

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. An overview of the Scala programming language. (2004).

S. Park, W. Xu, I. Yun, D. Jang, and T. Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1629–1642. https://doi.org/10.1109/SP40000.2020.00067

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

Chengnian Sun, Vu Le, and Zhendong Su. 2016a. Finding and Analyzing Compiler Warning Defects. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 203–213. https://doi.org/10.1145/2884781.2884879

Chengnian Sun, Vu Le, and Zhendong Su. 2016b. Finding Compiler Bugs via Live Code Mutation *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 849–863. https://doi.org/10.1145/2983990.2984038

Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016c. Toward Understanding Compiler Bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 294–305. https://doi.org/10.1145/2931037.2931074

TIOBE Software BV. 2021. TIOBE index. https://www.tiobe.com/tiobe-index/. Online accessed; 05-03-2021.

Seth Tisue. 2017. Bye bye JIRA — Scala issues migrated to GitHub scala/bug. https://contributors.scala-lang.org/t/bye-bye-jira-scala-issues-migrated-to-github-scala-bug/715.

Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 724–735.

https://doi.org/10.1109/ICSE.2019.00081

Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. 2017. A Comprehensive Study on Real World Concurrency Bugs in Node.js. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 520–531. https://doi.org/10.1109/ASE.2017.8115663

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 347–361. https://doi.org/10.1145/3062341.3062379

Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. 2021. An empirical study of optimization bugs in GCC and LLVM. *Journal of Systems and Software* 174 (2021), 110884. https://doi.org/10.1016/j.jss.2020.110884

David Zubrow. 2010. IEEE Standard Classification for Software Anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)* (2010), 1–23. https://doi.org/10.1109/IEEESTD.2010.5399061