Finding Typing Compiler Bugs

Anonymous Author(s)

Abstract

We propose a testing framework for validating static typing procedures in compilers. Our core component is a program generator suitably crafted for producing programs that are likely to trigger typing compiler bugs. One of our main contributions is that our program generator gives rise to transformation-based compiler testing for finding typing bugs. We present two novel approaches (*type erasure mutation* and *type overwriting mutation*) that apply targeted transformations to an input program to reveal type inference and soundness compiler bugs respectively. Both approaches are guided by an intra-procedural type inference analysis used to capture type information flow.

We implement our techniques as a tool, which we call Hephaestus. The extensibility of Hephaestus enables us to test the compilers of three popular JVM languages: Java, Kotlin, and Groovy. Within nine months of testing, we have found 153 bugs (128 confirmed and 71 fixed) with diverse manifestations and root causes in all the examined compilers. Most of the discovered bugs lie in the heart of many critical components related to static typing, such as type inference.

1 Introduction

Compiler reliability has a tremendous impact on the entire software ecosystem. To this end, *compiler testing* has substantially thrived since the beginning of the last decade [8], when Csmith [47], the most well-known program generator for C programs, first appeared. Csmith has paved the way for advanced program generation [18, 26, 28], transformation-based compiler testing [15, 21, 22, 26, 44], test-case reduction [39, 41], and test-case prioritization [6, 7]. The result from this research is far beyond prominent: (1) discovery of thousands of (critical) bugs in well-established compilers, such as GCC and LLVM, and (2) enhancements on the compiler testing pipelines [2].

State-of-the-art research endeavors primarily focus on finding bugs in optimizing compilers. Indeed, optimizations is a source of problems that justifiably keeps researchers preoccupied with verifying and testing the implementation of optimizations [21, 24, 25, 29, 30, 47, 48]. However, optimization issues is not the only challenge when working with compilers: a recent study [5] has showed that compilers, and especially those of languages that feature rich type systems (e.g., Java), suffer from bugs in static typing and semantic analysis procedures. Notably, such procedures examine if the input program is error free, thus it is very important that they are implemented correctly. Unfortunately, the ongoing language evolution and the difficulty of harmonizing new language features with type systems [20, 32] render the implementation of the corresponding typing algorithms notoriously challenging.

Typing bugs degrade the reliability of programs and developers' productivity. Specifically, such bugs can (1) lead to the rejection of well-formed programs making developers waste time on debugging their correct programs, (2) violate the safety provided by type systems [35] and can potentially cause security issues at runtime, or (3) invalidate subsequent compiler phases, such as optimizations.

Despite the importance of typing bugs, testing static typing procedures has been barely the goal of the existing testing campaigns. To our knowledge, the only relevant work is the fuzzer introduced by Dewey et al. [14] in 2014, which has found only a couple of bugs in the Rust's type-checker using a form of constraint logic programming.

Approach: We introduce a systematic and extensible approach for detecting typing compiler bugs. Our approach is motivated and guided by the findings and observations of a study [5] on 320 typing bugs in the compilers of four JVM languages, namely Java, Scala, Kotlin, and Groovy:

- **F1** Typing bugs mainly (51%) manifest as unexpected compile-time errors, meaning that the buggy compiler mistakenly rejects a well-formed program.
- **F2** An important portion (40%) of typing bugs lie in the implementation of type inference engines and in other type-related operations (e.g., subtyping checks).
- **F3** One third of typing bugs are triggered by non-compilable (e.g., ill-typed) code.
- **F4** Language features related to parametric polymorphism (e.g., use of parameterized classes, bounded polymorphism) are important for uncovering typing bugs, while unlike optimization bugs [28], loops and complex arithmetics do not exhibit high bug-revealing capability.
- **F5** Many aspects of typing bugs (i.e., symptoms, root causes, and test case characteristics) are uniformly distributed across the studied compilers.

Our approach is based on both *program generation* and *transformation-based compiler testing*. The first component of our approach is a *program generator* that comes with three important characteristics. First, it produces *semantically valid* programs, because typing bugs mainly cause the compiler to reject *well-formed* programs (**F1**). Rejecting a well-formed program produced by our generator indicates a potential compiler bug (test oracle). Second, the resulting programs rely heavily on parametric polymorphism (**F4**), while we avoid generating complex arithmetics or nested loops, because such features are irrelevant to the types of bugs we aim to detect. Third, to test compilers of different languages (**F5**), our generator yields programs at a higher-level of abstraction, and then uses translation mechanisms to convert the "abstract" programs into the target language.

The second component is based on our design of two novel transformation-based methods: *type erasure mutation* and *type overwriting mutation* whose goal is to exercise compilers' type inference algorithms and other type-related operations (**F2**). Given an input program P, the type erasure mutation removes declared types from variables and parameters, or type arguments from parameterized constructor and method calls, while preserving the well-formedness of P. The type overwriting mutation adopts a fault-injecting approach (**F3**), and introduces a type error in P by replacing a type t_1 with another incompatible type t_2. The type overwriting mutation updates the test oracle, as compiling a program obtained from this mutation indicates a potential soundness bug in the compiler under test. To perform sound transformations with respect to the test oracle, both mutations rely on a model underpinned by an intra-procedural type inference analysis that captures (1) the declared and inferred type of each variable, (2) how each type parameter is instantiated, and (3) dependencies among type parameters.

Testing campaign: Our tool implementation called Hephaestus[1] is currently able to test compilers for three different languages: Java (`javac`), Kotlin (`kotlinc`), and Groovy (`groovyc`). All selected languages are statically-typed, object-oriented languages, feature advanced type systems, and support parametric polymorphism via the Java generics framework [4]. Java is consistently on the list of the top five most popular languages [19, 45]. Kotlin has become the de-facto language for Android development [33]: already over 80% of the top-1000 Android applications use Kotlin [3]. Finally, Groovy is a popular hybrid language that supports both dynamic and static typing.

Over a period of nine months, we have found 153 bugs in all the examined compilers, of which 71 bugs were subsequently fixed by developers. Thanks to type erasure and type overwriting mutations, we have uncovered 50 inference and 22 soundness bugs, which we were unable to detect by using our program generator by itself. Our results further indicate that our mutations are able to increase coverage of compiler code. For example, type erasure mutation covers up to 5,431 more branches and invokes up to 217 more functions, when compared to our generator.

Contributions: We make the following contributions.

- A program generator carefully designed to find typing bugs in compilers of diverse languages.
- Two novel transformation-based testing techniques, namely *type erasure mutation* and *type overwriting mutation* used for finding type inference and soundness issues. Both methods rely on an intra-procedural type inference analysis for capturing type information flow.

[1] In Greek mythology, Hephaestus was the smithing god.

```
1  public class Test {
2    void test() {
3      def closure = { new B<>(new A<Long>()); }
4      A<Long> x = closure().f
5    }
6  }
7  class A<T> {}
8  class B<T> {
9    T f;
10   B(T f) { this.f = f; }
11 }
```

Figure 1. GROOVY-XXXX: This well-typed program is rejected by the Groovy compiler.

- An openly available implementation called Hephaestus, which is the first tool that is capable of testing different JVM compilers: `javac`, `kotlinc` and `groovyc`.
- A thorough evaluation of Hephaestus in terms of both bug-finding capability and code coverage improvement. From February 2021 to mid-November 2021, Hephaestus has found 153 bugs in total, of which 11 bugs are in `javac`, 32 are in `kotlinc`, and 110 in `groovyc`.

Availability: We plan to make our research artifact publicly available, concurrently with this paper's publication.

2 Illustrative Examples

To motivate the importance of typing bugs, we present two real examples detected by our tool.

Unexpected compile-time error in `groovyc`: Figure 1 shows a Groovy program that leads to this error in `groovyc`. Unexpected compile-time error are cases where the bug makes the compiler mistakenly reject a well-formed program. This bug had affected `groovyc` since version 2.0.0. For almost a decade (from December 2011 until May 2021 when we reported it), this bug had slipped the thorough testing efforts applied by the Groovy development team. Notably, this long-latent issue was resolved within days after reporting it.

The program declares two parameterized classes, namely `A` and `B`. Class `B` defines a field whose type is given by the type parameter `T`. On line 3, the code declares a lambda that returns an object of type `B<A<Long>>`. Although the type argument of `B` is omitted on line 3 (via the diamond operator <>), the compiler should be able to infer the corresponding type parameter from the type of the constructor's argument, which is `A<Long>`. However, a type inference bug causes the compiler to report a type mismatch on line 4. `groovyc` incorrectly infers the type of `closure().f` as `Object` instead of `B<A<Long>>`. Surprisingly, replacing `A<Long>` with `Long` at line 3 successfully compiles the program.

Erroneous compilation of ill-typed program in `kotlinc`: Figure 2 presents another bug detected by Hephaestus. The development and the stable versions of `kotlinc` fail to detect a type error in this program. This bug is a regression introduced by a major refactoring in the

```
1 fun <T1: Number> foo(x: T1) {}
2 fun <T2: String> bar(): T2 { return "" as T2 }
3 fun test() {
4  foo(bar())
5 }
```

Figure 2. KT-XXXX: This ill-typed program is compiled by the Kotlin compiler.

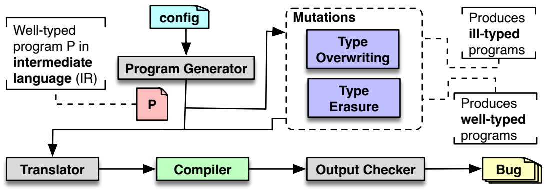

Figure 3. Our approach for detecting typing compiler bugs.

type inference algorithm of Kotlin, shipped with version 1.4, which appeared in August 2020. The bug remained undetected until we reported it in September 2021, and was classified as *"major"* by developers.

The program defines two parameterized functions: `foo` and `bar`. The first function declares a type parameter `T1` bounded by `Number`, while the second one introduces `T2` bounded by `String`. When calling `bar` at line 4, `kotlinc` instantiates it as `Number => Number`, because the return value of `bar` flows to a parameter whose type is bounded by `Number`. However, this type substitution is not valid, as `T2` cannot be a `Number`. Hence, instead of accepting the program, `kotlinc` should have raised a type error of the form: *"type parameter bound for T2 is not satisfied: inferred type Number is not a subtype of String"*.

The two examples demonstrate that both well-typed and ill-typed programs can uncover typing bugs. Furthermore, the bug-revealing programs combine multiple language features, e.g., mix of parametric polymorphism, lambdas, type inference, etc. Finally, both examples highlight that the process of static typing is hard to get right.

3 Techniques

Figure 3 summarizes our approach for detecting typing compiler bugs. The core component of our approach is a program generator (Section 3.2) designed to produce *well-formed* programs written in an *intermediate representation (IR)* (Section 3.1), an object-oriented language supporting parametric polymorphism, functional programming constructs, and type inference. As our approach tests multiple compilers, we use this IR to abstract away differences of target languages. Our generator takes as input a configuration that can either disable certain features (e.g. bounded polymorphism), enable them, or affect their probability distribution. Language-aware translators then convert a program written in the IR into a corresponding source file, which is ultimately passed as an input to the compiler under test.

We have also designed two transformation-based approaches (Section 3.4), namely: a *type erasure mutation* and a *type overwriting mutation* for detecting type inference and soundness bugs respectively. The type erasure mutation is a semantics-preserving transformation that removes type-related information from an input program. The type overwriting mutation replaces a type t with another type t' in a way that this replacement invalidates the program's correctness. Hence, unlike a program produced by our generator or through the type erasure mutation, we expect the compiler to reject the output of type overwriting mutation.

In contrast to previous work [28, 43, 47, 48], which requires differential testing [34], our approach does not need to employ it, as each program derived either from the generator or our mutations also acts as an oracle, based on the way it was derived. In the following, we present the technical details behind each component of our approach.

$$
\begin{aligned}
\langle p \in \mathit{Program}\rangle &::= \bar{d} \\
\langle d \in \mathit{Decl}\rangle &::= \texttt{class}\ C\ \texttt{extends}\ e\ \bar{d} \\
&\quad\mid \texttt{fun}\ m(x : t \ldots) : t = e \\
&\quad\mid \texttt{var}\ x : t = e \mid \Lambda\alpha.d \\
\langle e \in \mathit{Expr}\rangle &::= \mathit{val}(t) \mid x \mid e.f \mid e \oplus e \mid \{d \ldots e \ldots\} \\
&\quad\mid (e.m\ t)(e \ldots) \mid (\texttt{new}\ C\ t)(e \ldots) \\
&\quad\mid e.x = e \mid \texttt{if}(e)\ e\ \texttt{else}\ e \mid e :: m \\
&\quad\mid \lambda x : t.e \\
\langle \oplus \in \mathit{BinaryOp}\rangle &::= == \mid != \mid \&\& \mid || \mid > \mid \geq \mid < \mid \leq \\
\langle x \in \mathit{VariableName}\rangle &::= \textit{is the set of variable and field names} \\
\langle m \in \mathit{MethodName}\rangle &::= \textit{is the set of method names} \\
\langle C \in \mathit{ClassName}\rangle &::= \textit{is the set of class names}
\end{aligned}
$$

(a) Syntax of the IR.

$$
\begin{aligned}
\langle t \in \mathit{Type}\rangle &::= \top \mid \bot \mid \alpha \mid \mathcal{T} : t \\
&\quad\mid \Lambda\alpha.t \mid (\Lambda\alpha.t)\ t \\
\langle \alpha \in \mathit{TypeParameter}\rangle &::= \phi : t \\
\langle \mathcal{T} \in \mathit{TypeName}\rangle &::= \textit{is the set of type names}
\end{aligned}
$$

(b) Types in the IR.

Figure 4. The syntax and the types in the IR.

3.1 IR and Preliminary Definitions

Figure 4a shows the syntax of the IR. Use- and declaration-site variance, interfaces, and abstract classes are omitted from the figure for the sake of simplicity. A program in the IR consists of a sequence of declarations. A declaration is either a class, a function, or a variable. The language also supports parameterized declarations by introducing a type parameter in the body of the declaration (see $\Lambda\alpha.d$). The IR contains constant values of a type t ($\mathit{val}(t)$) and the typical expressions encountered in an object-oriented language (e.g., conditionals, parameterized method and constructor calls), along with functional features, i.e., method references and lambdas. Arithmetic expressions, loops, exceptions, access modifiers (e.g., `public`, `private`) are not supported.

Regarding types (Figure 4b), the language involves a nominal type system. A type is either (1) a regular type ($\mathcal{T} : t$) labeled with a name $\mathcal{T}$ and a supertype t, (2) a type parameter ($\phi : t$) with an upper bound t, (3) a type constructor, or (4) a type application that instantiates a type constructor with a set of concrete types.

In the following definitions, we use the symbol <: to denote the subtyping relation, and the operation $S(t)$ to give the supertype of type t (e.g., $S(\mathcal{T} : t) = t$)).

Definition 3.1. (Type substitution) The substitution $[\alpha \mapsto t] : \mathit{Type} \longrightarrow \mathit{Type}$, where α is a type parameter, and t is a type $t \in \mathit{Type}$, is inductively defined by:

$$\begin{aligned} [\alpha \mapsto t]\alpha &= t \\ [\alpha_1 \mapsto t]\alpha_2' &= \alpha_2 \qquad\qquad \alpha_1 \neq \alpha_2' \\ [\alpha \mapsto t_1]\mathcal{T} : t_2 &= \mathcal{T} : [\alpha \mapsto t_1]t_2 \\ [\alpha \mapsto t_1]\Lambda\alpha.t_2 &= [\alpha \mapsto t_1]t_2 = (\Lambda\alpha.t_2)t_1 \\ [\alpha_1 \mapsto t_1](\Lambda\alpha_2.t_2)t_3 &= (\Lambda\alpha_2.t_2)[\alpha_1 \mapsto t_1]t_3 \end{aligned}$$

A type substitution replaces all occurrences of a type parameter α in a type t_1 with another type t_2.

Definition 3.2. (Type unification) Type unification ($\mathit{Type} \times \mathit{Type} \longrightarrow S$) is an operation that takes two types ($t_1, t_2 \in \mathit{Type}$) and returns a type substitution σ so that σt_1 <: t_2:

$$\begin{aligned} \mathit{unify}(\alpha, t) &= [\alpha \mapsto t] \\ \mathit{unify}((\Lambda\alpha.t)t_1, (\Lambda\alpha.t)t_2) &= \mathit{unify}([\alpha \mapsto t_1]t, [\alpha \mapsto t_2]t) \\ \mathit{unify}(t_1, t_2) &= \mathit{unify}(t_1, S(t_2)) \quad t_1 \notin \mathit{TypeParam} \end{aligned}$$

Type unification identifies a substitution σ so that the type σt_1 is a subtype of t_2. We explain how we use the above definitions, when detailing our techniques.

3.2 Program Generation

We now describe the internals of our program generator used to produce programs written in the IR.

Context: Our program generator maintains a data structure called *context*, which stores all declarations and types in their namespace. Specifically, we use context to store the following entities: class-, method- and variable declarations (i.e., local variables, class fields, and method parameters), as well as type parameters and lambdas. Every time our generator uses a declaration or a type (e.g., initializing an instance of a class), it consults the context to determine which declarations are available in the current scope.

Generating declarations: The entry point of our program generator is the creation of random declarations (i.e., either a class, a method, or a variable) in the top-level scope. The maximum number of these top-level declarations is given as an input. When our generator constructs a declaration, it adds it to the context so that subsequent declarations and expressions can refer to the initial one.

Generating types: To generate a type, our generator first computes the set of available types in the current scope, and then picks one type at random. This set contains types from three different sources, namely, (1) built-in types (e.g., `Int`, `String`, `Array`) supported by the language under test, (2) types derived from previously generated classes, and (3) type parameters that are available in the current scope. Notably, for obtaining the second and the third source of types, our generator consults the context, while the first set is a constant given as an input to the generator. If the selected type is a type constructor, our generator instantiates it by recursively generating types with which the corresponding type parameters are instantiated.

Generating expressions: To avoid producing ill-typed expressions and programs, our program generator adopts a *type-driven* approach for generating expressions. This means that it first constructs a random type t, and then creates a random expression e of a type t', where t' <: t. Generating such expressions helps us exercise the implementation of subtyping rules in the compiler under test.

Object initialization: Expression generation is done up to a certain depth provided as input to the generator. However, infinite loops may occur, especially when initializing objects of classes with circular dependencies [27]. To prevent this from happening, after reaching the maximum depth, the generator initializes objects with constant values (i.e., $\mathit{val}(t)$), which are typically translated into cast `null` expressions.

Resolving matching methods and fields: When constructing a method call, a method reference, or a field access of a type t, the generator examines the context to resolve existing methods and fields that match the given type t.

Algorithm 1 illustrates the resolution process performed when generating a method call of a type t. When dealing with field accesses and method references, the resolution process works in a similar manner. Specifically, resolution involves three steps. In the first step, we inspect the current scope to find methods whose return type is either a subtype of t (line 2), or live objects containing at least one instance method whose signature matches t (line 3).

If the above search fails (i.e., *methods* = **nil**), we examine all previously declared classes and check whether there is any class containing such a method (line 5). To answer this question we use the `resolveMatchingClass` procedure (lines 11–23). For every class c and method m, our resolution algorithm unifies the return type r of m with type t (line 16), and if σr <: t, it instantiates the corresponding receiver type that stems from class c using the (partial) substitution obtained by type unification (line 18). Finally, the procedure picks a receiver type *rt*, and a method m at random (line 24), and generates an expression of type *rt* corresponding to the receiver of method call (line 25).

When the search of `resolveMatchingClass` fails (line 23), `resolveMethod` ultimately produces a fresh method with a return type t, and adds it to the current scope

Algorithm 1: Resolve methods by return type t.

```
1  fun resolveMethod(t, context, n) =
2      methods ← resolveMatchingFunctions(t, context, n)
3      methods ←
           methods ∪ resolveMatchingObjects(t, context, n)
4      if methods = nil then
5          methods ← resolveMatchingClass(t, context, n)
6      if methods = nil then
7          (rt, method) ← generateMatchingMethod(t, context, n)
8      else (rt, method) ← random(methods)
9      return (rt, method)
10
11 fun resolveMatchingClass(t, context, n) =
12     methods ← []
13     for c ∈ getClasses(context, n) do
14         for m ∈ getMethods(c) do
15             r ← getRetType(m)
16             σ ← unify(r, t)
17             if σr <: t then
18                 rt ← instantiate(toType(c), types, σ)
19                 σ′ ← σ ∪ getTypeSubstitution(rt)
20                 if isParameterized(m) then
21                     m ← instantiate(toType(m), types, σ′)
22                 methods ← methods ∪ (rt, m)
23     if methods = nil then return []
24     (rt, method) ← random(methods)
25     return generateExpr(rt, context, n), method
```

or to an existing class (line 7). Otherwise, it randomly selects a pair of a receiver and a method, which is the output of the algorithm (line 9). Our generator then uses this output to finish the generation of method call accordingly.

3.3 Modeling Type Information

We introduce a model for reasoning about type information in a program written in the IR. The model is based on the notion of a *type graph* (Sections 3.3.1), a program representation that captures how type information flows between declarations and type parameters. We present an intra-procedural type inference analysis for building type graphs (Section 3.3.2). Finally, based on type graph, we introduce the properties of *type preservation* and *type relevance* (Section 3.3.3) which, as we will show in Section 3.4, our testing approaches are based on.

3.3.1 Type Graph Formulation. The type graph captures (1) the declared and inferred type of program declarations, (2) how each type parameter is instantiated, and (3) the inter-dependencies between type parameters. We define a type graph as $G = (V, E)$. There are nodes of two kinds: a node $n \in V$ is either a declaration $d \in Decl$, or a type $t \in Type$. The set of edges $E \subseteq V \times V \times L$, where $L = \{\texttt{decl}, \texttt{inf}, \texttt{def}\}$ indicate the following: given a type graph G, the edge $n \xrightarrow{\texttt{decl}} t$ denotes that the type of node n is

TYPE APPLICATION

$$\frac{t = (\Lambda\alpha.t_1)t_2}{A(G,t) \Rightarrow G \cup \{(\Lambda\alpha.t_1)t_2 \xrightarrow{\texttt{def}} \alpha, \quad \alpha \xrightarrow{\texttt{decl}} t_2\}}$$

VAR DECL

$$\frac{e = \texttt{val}\ \ x : t_1 = e \qquad t_2 = getType(e)}{A(G,e) \Rightarrow G' \cup \{x \xrightarrow{\texttt{decl}} t_1, \quad x \xrightarrow{\texttt{inf}} t_2\}}$$

VAR PARAM CONSTRUCTOR

$$\frac{\begin{array}{c} e = \texttt{val}\ \ x : t_1 = \texttt{new}\ \ (Ct_2)() \\ \Lambda.\alpha.t = toType(C) \qquad \sigma = unify'(t_1, (\Lambda.\alpha.t)t_2) \end{array}}{A(G,e) \Rightarrow G \cup \{x \xrightarrow{\texttt{decl}} t_1,\ \ x \xrightarrow{\texttt{inf}} (\Lambda\alpha.t)t_2\} \cup \{\alpha \xrightarrow{\texttt{inf}} \sigma(\alpha) | \alpha \in \sigma\}}$$

VAR PARAM METHOD CALL

$$\frac{\begin{array}{c} e = \texttt{val}\ \ x : t_1 = e_1.(mt)() \\ t_2 = getRetType(e_1, m) \\ typeParam(m) \in t_2 \qquad \sigma = unify'(t_1, t_2) \end{array}}{A(G,e) \Rightarrow G \cup \{x \xrightarrow{\texttt{decl}} t_1,\ x \xrightarrow{\texttt{inf}} getType(e)\} \cup \{\alpha \xrightarrow{\texttt{inf}} \sigma(\alpha) | \alpha \in \sigma\}}$$

PARAM CALL

$$\frac{\begin{array}{c} e = e_1.(mt)(e_2) \\ t_1 = getType(e_2) \qquad t_2 = getParamType(e_1, m) \\ typeParam(m) \in t_2 \qquad \sigma = unify'(t_1, t_2) \end{array}}{A(G,e) \Rightarrow G \cup \{\alpha \xrightarrow{\texttt{inf}} \sigma(\alpha) | \alpha \in \sigma\}}$$

Figure 5. Analysis rules for building type graphs.

explicitly declared in the program as t. For example, for a variable declaration of the form `String x = ...`, there is a $\texttt{x} \xrightarrow{\texttt{decl}} t$ edge, where $t = \texttt{String}$. The edge $n_1 \xrightarrow{\texttt{inf}} n_2$ indicates that the type of node n_1 is inferred by node n_2. For example, for an assignment of the form `String x = "str"`, beyond a $\xrightarrow{\texttt{decl}}$ edge, there is also an $\texttt{x} \xrightarrow{\texttt{inf}} t$ edge, where $t = \texttt{String}$. This is because the type of variable `x` is inferred as `String`, which is the type of the constant at the right-hand side of the assignment. Finally, the edge $t_1 \xrightarrow{\texttt{def}} t_2$ shows that type t_1 consists of another type t_2. For example, for each type application of the form $t_1 = \texttt{A<String>}$, we have the edges $t_1 \xrightarrow{\texttt{def}} \texttt{T}$ and $\texttt{T} \xrightarrow{\texttt{decl}} t_2$, where $t_2 = \texttt{String}$. These edges indicate that parameterized type `A<String>` contains type parameter `T`, and the corresponding type argument is `String`.

3.3.2 Constructing Type Graphs. To construct type graphs, we design an intra-procedural, flow-sensitive analysis that operates on programs written in the IR. The analysis $A(G, n)$ constructs a type graph G by visiting every declaration and expression n of the given program. Figure 5 summarizes our analysis rules. For what follows, *unify*′ is a variant of type unification that adds the following rules:

$$unify((\Lambda\alpha.t)t_1, (\Lambda\alpha)t_2) = [\alpha \mapsto \alpha] \ \text{ if } t_1 = t_2$$

$$\begin{aligned} unify'((\Lambda\alpha_1.t_1)t_2, (\Lambda\alpha_2.t_3)t_4) &= [\alpha_2 \mapsto \alpha_1] \\ &\text{if } unify(S(t_3), [\alpha_1 \mapsto t_1]t_2) \neq \emptyset \land \\ &\land [\alpha_2 \mapsto t_4]t_3 <: [\alpha_1 \mapsto t_2]t_1 \end{aligned}$$

In essence, this modification finds dependent type parameters between two type applications. For example, suppose we have (1) two parameterized classes: `class A<T>` and `class B<T> extends A<T>`, and (2) two type applications derived from these classes, namely, t_1 = `A<String>` and t_2 = `B<String>`. In this scenario, $unify'(t_1, t_2)$ returns $[\alpha_2 \mapsto \alpha_1]$, where α_1 and α_2 are the type parameters of type constructors `A` and `B` respectively. This dependency information indicates that instantiating the type parameter α_2 with a type t also instantiates α_1 with the same type, as α_2 flows to the type parameter of superclass.

[TYPE APLICATION]: For each type application $t = (\Lambda\alpha.t_1)t_2$, the resulting type graph contains two edges. The first edge ($\xrightarrow{\text{def}}$) connects type application with the underlying type parameter α, and the second edge ($\xrightarrow{\text{decl}}$) connects α with type argument t_2.

[VAR DECL]: For a variable declaration, we connect variable `x` with two types: t_1 is the declared type of `x`, and t_2 is the type inferred by the right-hand side of declaration.

[VAR PARAM CONSTRUCTOR]: For a variable initialized by a parameterized constructor (e.g., `A<String> x = new A<String>()`), beyond adding the edges for the declared and inferred type of variable `x`, we unify the type of the right-hand side with that of the left-hand side to identify any dependent type parameters. If this is the case, we add the corresponding $\xrightarrow{\text{inf}}$ edges. This rule models the case where the type parameter of a constructor invocation is instantiated by using type information from the variable's declared type (e.g., `A<String> x = new A<>()`).

[VAR PARAM METHOD CALL]: When initializing a variable with the value of a parameterized method call, the type graph contains an $\xrightarrow{\text{inf}}$ edge, which connects the method's type parameter (in case it appears in the method's return type) with any dependent type or type parameter included in the declared type of `x`. This edge captures the case where a method's type parameter is instantiated based on the declared type of the target variable `x`.

[PARAM METHOD CALL]: When calling a parameterized method with arguments, the method's type parameter included in the type of the formal parameter is instantiated by the type of the expression e_2 passed as a call argument.

We treat any other case using one of the rules above. For example, a return value of a method is treated as the initial value of a virtual variable named `ret`. We model this using one of the [VAR .*] rules depending on the body of the method. Invoking a parameterized constructor with call arguments (i.e., `A<String>("f")`) is modeled as calling a parameterized method (i.e., [PARAM METHOD CALL]).

Example Type Graph: Figure 6 shows an example program and its type graph. The program consists of two parameterized classes with a parent-child relationship (lines 1–4), and a function `m` that returns a value of type `A<String>`. The produced type graph contains two declarations depicted with red color. The one declaration stands for the return value of function `m`, and the other corresponds to the field `f`, after initializing its receiver object at line 7. Observe the dependencies between type parameters. For example, the edge from node `B.T:7` to node `A.T:3` demonstrates that the type parameter of the parameterized constructor call on line 7 is instantiated by the type parameter coming from the call argument at line 8. This edge is captured by the [PARAM METHOD CALL] rule.

```
1  open class A<T>
2  class B<T>(
3    val f: A<T>
4  ): A<T>()
5
6  fun m(): A<String> {
7    return B<String>(
8      A<String>()
9    )
10 }
```

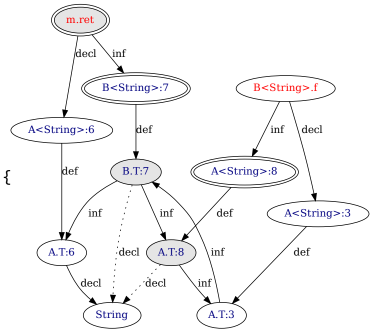

Figure 6. A Kotlin program and its type graph. Red nodes represent declarations and blue nodes are types. Types are annotated the line they come from. Double circled nodes are candidate nodes for the type erasure mutation, and shadowed nodes are candidate nodes for the type overwriting mutation.

3.3.3 Type Preservation and Type Relevance.

Assume that $\sqcup$ is the least upper bound operator, and *visitedTypes(G, n)* returns all *type nodes* in G that are reachable from the given node n through either $\xrightarrow{\text{decl}}$ or $\xrightarrow{\text{inf}}$ edges.

Definition 3.3. (Type inference) Type inference ($G \times V \longrightarrow Type$) is an operation that takes a type graph $G = (V, E)$ and a node $n \in V$, and returns a type. It is defined as:

$$infer(G, n) = \bigsqcup_{t \in visitedTypes(G,\ n)} t$$

This definition gives the type of a particular node n. This type stands for the least upper bound of all types that are reachable from n.

Definition 3.4. (Type erasure) Type erasure ($G \times V \longrightarrow G$) operates on a type graph $G = (V, E)$ and a node $n \in V$ and returns a new type graph. It is defined as:

$$\begin{aligned}
erasure(G, \alpha) &= G \setminus \{\alpha \xrightarrow{\text{decl}} n\}, \quad n \in G \\
erasure(G, (\Lambda.\alpha.t_1)t_2) &= erasure(G, \alpha) \\
erasure(G, t) &= G \quad \text{if } t \notin TypeParam \\
erasure(G, d) &= (G \setminus \{d \xrightarrow{\text{decl}} t\}) \cap erasure(G, t), \ t \in G
\end{aligned}$$

Type erasure takes a node n and a graph G, and removes all $\xrightarrow{\text{decl}}$ edges associated with the given node n. Conceptually,

Algorithm 2: Algorithm for type erasure mutation.

```
1  fun typeErasureMutation(P)=
2    for m ∈ Methods(P) do
3      G ← A(∅, m)                          // Builds type graph
4      nodes ← findCandidateNodes(G)
5      nodes ← {n ∈ nodes|n preserves its type on G}
6      for k = len(nodes)to1 do
7        for ⟨n1, n2, . . . , nk⟩ ∈ combination(nodes, k) do
8          if ⟨n1, n2, . . . , nk⟩ preserve their type on G then
9            erase ⟨n1, n2, . . . , nk⟩
10           break
```

type erasure removes variables' declared types, or types used as type arguments from the corresponding type parameters.

Based on the *infer* and *erasure* operations, we now present the *type preservation* and *type relevance* properties.

Definition 3.5. (Type preservation) Given a type graph $G = (V, E)$, and a node $n \in V$, we say that n *preserves* type $t \in Type$, when $infer(G, n) = t$, $G' = erasure(G, n)$ and $infer(G', n) = t$.

Definition 3.5 says that a node n preserves its type t, when even after erasing type information from n (e.g., a variable's declared type), the inferred type for n is still t. We can generalize the type preservation property for multiple nodes.

Definition 3.6. (Generalized type preservation) Given a type graph $G = (V, E)$ and nodes $n_1, n_2 \dots n_k \in V$, we say that these nodes *preserve* their type when $t_1 = infer(G, n_1), \dots, t_k = infer(G, n_k)$, $G' = \bigcap_{i=0}^{k} erasure(G, n_i)$ and $t_1 = infer(G', n_1), \dots, t_k = infer(G', n_k)$.

Definition 3.7. (Type relevance) Given a type graph $G = (V, E)$ and a node $n \in V$, we say that n is relevant to type $t \in Type$, when $G' = erasure(G, n)$ and $infer(G', n) <: t$.

The definition above says that a type t is relevant to a node n, when, after performing type erasure, t is a supertype of the inferred type of n.

3.4 Mutations

We now introduce our novel testing approaches for detecting inference and soundness bugs. The input of both approaches is a program produced by the generator. Our techniques mutate the input program by leveraging the type preservation and type relevance properties presented earlier.

3.4.1 Type Erasure Mutation. The insight of the *type erasure mutation* (hereafter TEM) is that omitting types (wherever is possible) exercises the implementation of compilers' inference algorithms. Given an input program P, TEM removes type information from P in a way that this modification does *not* change the semantics of P. Our IR supports type removal for the following cases: (1) removing a variable's declared type (e.g., `var x = 1`), (2) removing type arguments from a parameterized constructor or method call (e.g., `new A<>("")`), (3) removing a return type from a method's signature, (e.g., `fun m() = "f"`), and (4) removing a type from a parameter of a lambda (e.g., `(x) -> x + 1`).

Removing types is not always benign, as it may lead to cases where type inference is impossible or the compiler infers a different type from the one initially declared, something that may cause type errors. For example, consider the following code snippet:

```
1 class A<T>(val f: T)
2 val x: Any = "str"
3 val y: A<Any> = A<Any>(x)
```

Removing the declared type of variable `x` (i.e., `val x = "str"`), and the type argument of the constructor call (i.e., `val y: A<Any> = A(x)`) makes the program ill-typed. This is because the compiler now infers the type of `x` as `String` and the type of the right-hand side of line 3 as `A<String>`. Therefore, there is a type mismatch while type-checking line 3, as we assign something of type `A<String>` to a variable of type `A<Any>`.

To prevent such situations from happening, we need to identify which types and which combinations of them can be safely disregarded. To answer this question, TEM leverages the type graph of the input program. In particular, TEM chooses to erase the types of nodes for which the type preservation property (Definition 3.6) holds.

Algorithm 2 summarizes the implementation of TEM, which we describe using the example program and the type graph shown in Figure 6. The algorithm takes an input program P, and for every method in P, TEM builds the corresponding type graph (lines 2, 3). On line 4, the algorithm examines the type graph to identify candidate nodes where type erasure is permissible (recall the four cases enumerated in the beginning of Section 3.4.1). In the example of Figure 6, there are three candidate nodes shown with double circles. Next, TEM excludes every candidate node that does not preserve its type based on Definition 3.5 (line 5). In our example, TEM filters out node `m.ret`, as after type erasure the return type of method `m` becomes `B<String>`. For the remaining set of nodes, our algorithm finds the *maximal* set of nodes that is omittable, meaning that the generalized type preservation property holds for the included nodes (lines 6–9). The intuition is that removing the maximal set of types allows us to explore more paths in the compiler, as there is much hidden type information that the compiler needs to infer.

Back to our example, TEM checks whether the combination of nodes `B<String>:7` and `A<String>:8` can be erased. Indeed, this is the case, as after type erasure both `B.T:7` and `A.T:8` are still instantiated with type `String` (observe that type node `String` is reachable from both nodes, using the graph produced by the *erasure* operation, where dotted edges are removed). As a result, TEM mutates the program accordingly, namely, it transforms the body of method `m` from `return B<String>(A<String>())` to `return B(A())`.

Remarks: By construction, TEM yields well-typed programs. Based on the type preservation property (Definitions 3.5 and 3.6), TEM considers only types for which it knows that their removal does not affect the typing of declarations and type parameters.

TEM has a worst case exponential complexity, as it enumerates the combinations of candidate nodes of any size k (Algorithm 2, lines 6–7). However, such an exponential behavior does cause performance problems in practice, because (1) our algorithm disregards any candidate node that is trivially non-omittable (i.e., the node does not preserve its type on its own), (2) our algorithm stops the enumeration when it finds the maximal combination of nodes.

3.4.2 Type Overwriting Mutation.

The goal of the *type overwriting mutation* (hereafter TOM) is to find soundness compiler bugs. To trigger such bugs, TOM provides the compiler under test with wrongly-typed programs. Specifically, TOM takes a well-typed program as input and mutates it by injecting a type error. Accepting and compiling such an invalid program indicates a potential soundness bug in the compiler. In particular, TOM introduces a type error in the input program by replacing a type t_1 with another type t_2 so that type mismatches arise. TOM performs these types of replacements on either the declared types of variables, or upper bounds and type arguments of type parameters.

The algorithm of TOM is similar to Algorithm 2. It starts by randomly picking one method from P and producing its type graph G. As in the case of the TEM algorithm, TOM examines G to identify nodes where the mutation is applicable. In the context of TOM, such nodes reflect either variable declarations or type parameters. Next, TOM selects a node n at random, and exploits the type relevance property (Definition 3.7) to generate a type t so that the selected node n is *not* relevant to type t. Rather than creating an incompatible type from scratch (i.e., creating `class A {}`), our algorithm generates t at random using the available types at the current scope. In this way, the compiler compares types with diverse shapes and characteristics, which in turn, triggers more subtyping checks and type-related operations in the compiler codebase. After generating such a type, TOM substitutes the declared type of n, with the newly created type t. When n is a type parameter, this replacement occurs in the type parameter's upper bound or explicit type argument.

Consider again the example in Figure 6, and suppose that among candidate nodes (shadowed nodes), TOM chooses to mutate node `A.T:8`. TOM generates a random type t (e.g., type `Int`) such that the type relevance property does not hold for the selected node (i.e., *infer(G, n)* = `String` $\not<:$ `Int`). The output of TOM is then an updated program where the body of `m` is `return B<String>(A<Int>())`. We expect the compiler to reject the mutated program by raising a diagnostic message of the form: *"type mismatch: inferred type is A<Int> but A<String> was expected"*.

3.5 Implementation

We have implemented our techniques as a tool named Hephaestus, which contains roughly 15k lines of Python code.

We observed that most of the testing time is spent on compiling the generated test programs. To mitigate this bottleneck, instead of generating and compiling one program at a time, Hephaestus generates and compiles programs in batches, where each batch contains a user-specified number of programs. Compiling programs in batches significantly boosts the performance of testing, as we avoid bootstrapping a JVM per generated program. Finally, for better throughput, Hephaestus generates and compiles programs using multiple processes via the `multiprocessing` module of Python.

Hephaestus can be extended with only little engineering effort. Specifically, supporting a new target language can be achieved by implementing: (1) a translator to convert a program in IR into a concrete program written in the target language, and (2) a regular expression that distinguishes compiler crashes from compiler diagnostic messages. Hephaestus currently supports Java, Kotlin, and Groovy, and each translator consists of around 800 LoC.

4 Evaluation

Our evaluation is based on the following research questions:

RQ1 Is Hephaestus effective in finding typing bugs in JVM compilers? (Section 4.2)

RQ2 What are the characteristics of the discovered bugs and the bug-revealing test cases? (Section 4.3)

RQ3 Are the type erasure and type overwriting mutations effective in detecting inference and soundness bugs respectively? (Section 4.4)

RQ4 Can Hephaestus improve code coverage? (Section 4.5)

To answer these questions, we used Hephaestus between February 2021 and mid-November 2021 to systematically test the selected JVM compilers. During this period, we ran Hephaestus for three months of CPU time, in total.

Result summary: Our key experimental results are

- **RQ1:** Hephaestus *has found many bugs.* Within nine months of testing, Hephaestus has detected 153 bugs, of which 128 are confirmed, and 71 have been already fixed by developers. Interestingly, Hephaestus was able to find bugs in *all* the examined compilers: 110 bugs in `groovyc`, 32 bugs in `kotlinc`, and 11 bugs in `javac`.
- **RQ2:** Hephaestus *finds typing bugs.* Hephaestus found 144 typing bugs, 2 parser/lexer bugs, and 7 back-end bugs. Most of these bugs are defects in the implementation of parametric polymorphism and type inference.
- **RQ3:** *The type erasure and type overwriting mutations are effective in revealing type inference and soundness bugs.* TEM has found 50 type inference-related bugs, while TOM discovered 24 bugs. Moreover, our mutations can exercise deep compiler code associated with type inference and other type-related operations, e.g., TEM has covered up

to 5,431 more branches, and invoked up to 217 more functions, when compared with our program generator.

- **RQ4:** Similarly to prior work [28, 47], the incremental code coverage improvement due to Hephaestus is small.

4.1 Experimental Setup

Compiler versions: To avoid reporting previously known bugs, our efforts have focused on testing the *latest development* version of each compiler. Note that our testing efforts were incremental, i.e. we concurrently developed Hephaestus and tested the compilers. Hence, we have run Hephaestus in its full capabilities for only one month.

Baseline: There is no relevant baseline to which we could compare Hephaestus. The fuzzer presented by Dewey et al. [14], which detects bugs in the type-checker of Rust, is the closest related work. Still, their tool is language-specific and probably outdated. Stepanov et al. [42] have developed a tool focusing on back-end crashes in the Kotlin compiler. However, their tool is not publicly available. Similarly, AFL [31] and the AFL compiler fuzzer [1] can only detect crashes.

Test case reduction: Hephaestus produces programs that trigger compiler bugs with three different manifestations [5]: unexpected-compile time error (UCTE), unexpected runtime behavior (URB), and compiler crashes. Unlike prior work [21, 28, 42, 47, 48], in most cases, test programs generated by Hephaestus are easy to reduce. For UCTE, the compilers emit informative diagnostic messages that help us locate the expression that is responsible for the bug, and reduce the test case effortlessly. URB errors are an outcome of the type overwriting mutation. Henceforth, Hephaestus logs the mutated program points; thus, we know precisely what line and instruction introduces the error. Nevertheless, minimizing programs that cause compiler crashes could benefit from an automated program reducer, such as C-Reduce [41].

Interaction with compiler developers: Groovy developers responded to most of our bug reports soon after reporting them, and typically patched easy-to-fix bugs within a week. Kotlin developers were also very responsive. Despite Kotlin developers being more interested in compiler crashes (as they fixed them immediately), they also answered other bug reports within a few days. For the OpenJDK's Java compiler, bug reports were verified within a week by developers. Unfortunately, OpenJDK's issue tracker is not open to the public. Although we tried to contact OpenJDK developers through email, we could not get any details beyond what is visible on their Jira deployment [37]. Furthermore, we could not interact directly with the bug tracker and comment on the reports. Therefore, we focused our testing efforts on Groovy and Kotlin compilers.

4.2 RQ1: Bug-Finding Results

Figure 7a summarizes the bugs we identified during our testing campaign. Overall, we reported 153 bugs. The developers confirmed most of them (128/153) as previously unknown, real bugs, while they have already fixed 71 bugs. This highlights the correctness and importance of the reported issues. As shown in the study of Chaliasos et al. [5], the relatively high number of unfixed bugs could be attributed to the fact that some of the submitted bugs required much time to be resolved, as they are challenging and need careful examination. Notably, one compiler developer commented on our bug reports: *"The generics bugs are tough and so it was like working on a difficult crossword or Sudoku puzzle every day."*

Before submitting a new bug report, we always performed two steps. First, we waited for developers to fix existing bugs that may had the same root cause as the bug we wanted to report. Second, we searched in the issue trackers to find potential duplicate bugs. Overall, 4 out of 153 reported bugs were marked as duplicates. Specifically, two of them have already been opened by other users, and the other two had the same root causes with bugs we have already submitted.

Finally, only nine bugs were marked by developers as "not an issue" or "won't fix". Most of these "won't fix" issues are associated with cases where either the corresponding type inference engines are *underimplemented*, or there are decidability issues in the underlying type systems [20, 32, 40]. We discuss one such example in Section 4.6 (Figure 11c).

Importance of bug-finding results: In general, compiler developers welcomed our testing efforts and bug reports. A developer mentioned that: "*Thanks for your high-quality bug reports. I have been finding them quite complete in terms of recreating the issue. And the inclusion of variations that work as expected gives a nice basis of comparison when investigating the root cause*". Furthermore, we identified issues in fundamental compiler components. For example, seven out of 32 bugs in `kotlinc` were classified as *"major"* by developers. A `groovyc` developer commented on our reports: "*Static compilation and static type-checking is, for me, one of the most important features that must work with most other features of the language*". All the above demonstrate the practical implications of our testing efforts.

Affected compiler versions: We also ran the test cases that accompany our bug reports on all stable compiler versions. Figure 8 presents how many stable compiler versions are affected by the discovered bugs. It is clear that Hephaestus is able to find both *long-standing* and *regression* bugs. In particular, 35 `groovyc` and 14 `kotlinc` bugs occur in *all* stable compiler versions, while there is also a non-trivial number of bugs that affect numerous versions (i.e., 10–12 affected versions). Such long-standing issues remain unnoticed for years, as the `groovyc` bug we discussed in Section 2. Also observe that a large portion of `groovyc` bugs (50/110—45%) are triggered *only* in the master branch of the compiler. These issues indicate that a feature that worked properly in previous versions, is broken in the current implementation `groovyc`. Regression bugs are often introduced by fixes of

Status	groovyc	kotlinc	javac	Total
Reported	0	12	0	12
Confirmed	46	8	3	57
Fixed	60	9	2	71
Duplicate	2	1	1	4
Won't fix	2	2	5	9
Total	**110**	**32**	**11**	**153**

(a)

Symptom	groovyc	kotlinc	javac	Total
UCTE	77	17	7	101
URB	19	3	0	22
Crash	14	12	4	30

(b)

Component	groovyc	kotlinc	javac	Total
Generator	54	16	7	77
TEM	35	12	3	50
TOM	20	3	1	24
TEM & TOM	1	1	0	2

(c)

Figure 7. (a) Status of the reported bugs in `groovyc`, `kotlinc`, and `javac`, (b) number of bugs with unexpected compile-time error (UCTE), unexpected runtime behavior (URB), and crash symptom, (c) bugs revealed by the generator, the type erasure mutation (TEM), the type overwriting mutation (TOM), and their combination (TEM & TOM).

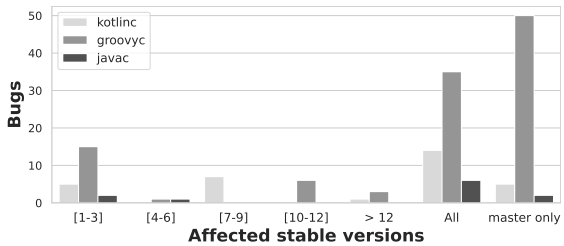

Figure 8. Number of bugs along with the number of stable versions they affect.

other bugs; their discovery reduces the friction and risk of development work.

4.3 RQ2: Bug and Test Case Characteristics

Figure 7b characterizes the bugs by their symptoms. Most of the discovered bugs (101/153) result in a UCTE, followed by 30 crashes, and 22 URBs. UCTE errors are triggered by well-formed programs produced by either our generator or the type erasure mutation. URB errors are an outcome of the type overwriting mutation that yields ill-typed programs. Finally, 26 crashes are caused by well-formed test cases, while four crashes are triggered by wrongly-typed code. The above results indicate that our approach enables the discovery of bugs with diverse manifestations.

We also identified what language features are involved in every minimized bug-revealing program that accompanies our bug reports. Features related to parametric polymorphism (e.g., parameterized class) are in the list of features with the most bug-revealing capability. In total, 104/153 bugs are caused by programs containing at least one such feature. This confirms a comment by a compiler developer who wrote: "*generics are the feature with the most latent concerns*". Type inference is another category of features that is hard to get right, as type inference features appear in 61 test cases. We further observed that some features are often combined with other individual features. For instance, in 47% of test cases that use conditionals, type inference features are also included. These results (1) validate our design decision to focus our efforts on parametric polymorphism and type inference (Section 1), and (2) are consistent with the study of Chaliasos et al. [5] who first pointed out the impact of specific language features on triggering typing bugs.

Compiler		Line Coverage	Function Coverage	Branch Coverage
groovyc	Generator	42.68 %	41.77 %	42.07 %
	TEM change	+167 (0.46 %)	+27 (0.37 %)	+752 (0.45 %)
	TOM change	+99 (0.27 %)	+10 (0.14 %)	+447 (0.27 %)
	TEM `stc.*`	+106 (4.6%)	+13 (3.6%)	+531 (4.58%)
kotlinc	Generator	30.92 %	30.60 %	30.32 %
	TEM change	+787 (0.46 %)	+217 (0.39 %)	+5,431 (0.46 %)
	TOM change	+572 (0.33 %)	+166 (0.30 %)	+4,171 (0.35 %)
	TEM `resolve.calls.inference.*`	+238 (17.8%)	+63 (14.9%)	+1,865 (20.1%)
	TEM `resolve.*`	+572 (3.93%)	+135 (3.3%)	+4,086 (4.2%)
	TEM `types.*`	+147 (4.5%)	+69 (6.5%)	+957 (4.3%) %
javac	Generator	36.99 %	39.68 %	34.56 %
	TEM change	+396 (0.68 %)	+87 (0.81 %)	+2,150 (0.62 %)
	TOM change	+362 (0.62 %)	+79 (0.74 %)	+1,990 (0.57 %)
	TEM `comp.Resolve`	+100 (14.1%)	+27 (14.2%)	+613 (15.7%)
	TEM `comp.*`	+204 (2.67%)	+47 (3.6%)	+1,200 (3.1%)
	TEM `code.Types`	+113 (8.1%)	+23 (7.7%)	+ 558 (7.5%)
	TEM `code.*`	+131 (3.3%)	31 (3.2%)	636 (3.3%)

Figure 9. Coverage increase by type erasure (TEM) and type overwriting (TOM) mutations.

4.4 RQ3: Effectiveness of Mutations

Figure 7c shows the number of bugs triggered by the generator, the mutators, and their combination. Our mutations led to the identification of 76 out of 153 bugs, about half of the total discovered bugs. Our generator fails to detect these 76 bugs, as they are all related to either type inference issues or other issues triggered by wrongly-typed code. TEM is an effective approach, able to identify 50 type inference bugs. This suggests that beyond compiler optimizations, type inference is another compiler procedure that can cause problems and deserves the attention of researchers. Finally, TOM either by itself or in combination with TEM has uncovered 26 bugs, of which 22 bugs are soundness issues. Detecting soundness bugs is of particular importance because such bugs can lead to unexpected runtime errors and security issues.

We also conducted an experiment to estimate the impact of mutations on code coverage. To do so, (1) we instrumented each compiler using the JaCoCo code coverage library [17], (2) we generated 10k random programs via Hephaestus, (3) for each generated program, we produced two mutants using TEM and TOM respectively, and (4) we measured the code coverage increase that comes from compiling each mutant.

Figure 9 shows the results of this experiment. In all compilers, TEM and TOM increase line, function, and branch coverage when compared to our generator. In all cases, TEM is more effective in exercising new code than TOM. Also, `kotlinc` testing exhibits the most noticeable increase

Compiler		Line Coverage	Function Coverage	Branch Coverage
groovyc	test suite	82.00 %	71.77 %	78.38 %
	test suite & random	82.06 %	71.79 %	78.44 %
	% change	+0.06 %	+0.02 %	+0.05 %
kotlinc	test suite	75.80 %	64.11 %	70.01 %
	test suite & random	75.88 %	64.17 %	70.08 %
	% change	+0.08 %	+0.06 %	+0.07 %
javac	test suite	80.82 %	80.14 %	81.29 %
	test suite & random	81.00 %	80.16 %	81.50 %
	% change	+0.18 %	+0.02 %	+0.21 %

Figure 10. Coverage on compilers' test suites plus 10K randomly-generated programs.

in terms of absolute numbers. For example, TEM covers 787 (0.46%) additional lines of code, triggers 5,431 (0.46%) additional branches, and calls 217 (0.39%) more functions.

At a first glance, the percentage increase may seem low (<1%). However, we should clarify that the goal of our mutations is to exercise the inference engines and other type-related operations, and *not* explore the entire compiler codebase. To validate this we further investigated the results. Indeed, when examining `kotlinc` results, we observe that TEM mostly exercises code in `resolve.*` and `types.*` packages, e.g., 204 / 217 (94%) of the additionally invoked functions belong to one of these packages. Specifically, these packages contain code responsible for inferring types and resolving method calls by building and solving a type constraint problem (e.g., see `resolve.calls.inference` package). In `groovyc`, TEM mostly covers code in the package responsible for static typing (namely, `stc.*`). Finally, in `javac`, TEM exercises much code in the `code.*` and `comp.*` packages, which among other things, contain the implementation of (1) `javac`'s name resolution algorithm (`comp.Resolve`), and (2) type-related operations, such as type variable substitution (`code.Types`). Similarly, TOM mainly exercises code in the aforementioned packages.

The above results clearly suggest that our mutations can effectively find bugs through increased coverage of relevant compiler procedures, such type inference.

4.5 RQ4: Code Coverage

To answer this research question, we employed the JaCoCo code coverage tool. Specifically, we measured for each compiler the code coverage of its test suite, plus 10K programs produced by Hephaestus. Figure 10 summarizes our results. We observe that in all cases, the code coverage improvement is negligible. Nevertheless, Hephaestus is still able to trigger numerous bugs in all studied compilers. For example, although the line coverage improvement on `groovyc` is only +0.06 %, Hephaestus was able to find 110 `groovyc` bugs. Thus, we find that traditional code coverage metrics are too *shallow* to capture the efficacy of our approach (as also observed in testing optimizing compilers [28, 47]).

4.6 Examples of Reduced, Bug-Triggering Programs

We discuss a selection of bugs discovered by Hephaestus.

Figure 11a: While type-checking the variable declaration on line 7, `groovyc` checks whether the call of the parameterized method `foo` returns a subtype of `C<String>`. The problem here is that due to a bug in its inference algorithm, `groovyc` fails to infer the correct type for instantiating type variable `T` of function `foo`. Consequently, `groovyc` infers the return type of `foo` as `Object` instead of `C<String>`. This bug was found by TEM.

Figure 11b: This program triggers a bug in the Kotlin compiler that leads to a compiler crash. The program defines a parameterized class `B` that contains a parameterized function `m`, which in turn declares a bounded type parameter `X`. When calling method `m` at line 3, we instantiate it with `C<out number>` as the type argument (note that `out Number` is the equivalent of `? extends Number` in the Java world). The compiler then tries to compute the captured type for `X` but it crashes due to a missing condition in the implementation of type capturing. This bug was found by TOM.

Figure 11c: This program presents a "wont'fix" `javac` issue. Although the least upper bound of the conditional (line 7) is type `T`, the compiler infers the type of local variable `v` as type `double`. This in turn causes a type mismatch as a `double` cannot be converted to type `T` (see line 7). A Java developer commented that type inference is not possible in this case, as the target variable `v` does not contain all required constraints to compute an "optimal" solution. Using `T extends Double` (line 1) is the cause of this issue, as using `T extends Number` or any other type leads to a successful compilation. Beyond that, replacing the expression at line 7 with `(true) ? (T) null : (K) null` results in a correct compilation. All the above suggest that this is a broader issue in `javac`'s type inference algorithm design and implementation; this issue was found by TEM.

5 Related Work

Program Generators. Csmith [47] is a program generator for C programs, that has found hundreds of bugs in GCC and Clang. Csmith generates programs that are free from *undefined behavior*. Relying on Csmith, several other program generators have emerged for (1) testing other compilers (e.g., OpenCL) [26], or link-time optimizers [23], and (2) generating more expressive programs [18].

Epiphron [43] is a program generator for C that aims to uncover defects in the error reporting mechanisms of compilers. Unlike Csmith, Epiphron does not necessarily generate programs that are free from undefined behavior. Targeting optimizations bugs, Orange [36] creates programs that involve longer and more complex arithmetic expressions, such as floating-point arithmetics. YARPGen [28] is a program generator for C/C++ programs that comes with a set of generation policies aiming to trigger certain optimizations.

Most of the aforementioned program generators focus on the detection of crashes or miscompilations caused by optimization bugs. Finding miscompilations requires *differential*

```
class A {
 static <T> T foo(C<T> t) { ... }
}
class C<T> {}
class B {
 void test() {
  C<String> x = A.foo(new C<>())
 }
}
```

(a) Groovy-XXXX: A bug in the inference engine of `groovyc` that leads to an UCTE.

```
class A<T: B<out Number>>(val x: T) {
 fun test() {
  val y: Int = x.m<C<out Number>>()
 }
}
class B<T> {
 fun <X: C<T>> m(): Int = 1
}
class C<T>
```

(b) KT-XXXX: A crash found in `kotlinc` due to a bug in type constructor projection.

```
class A<T extends Double,
        K extends T> {

 public T test() {
  T foo = null;
  var v = (true) ? foo : (K) null;
  return v;
 }
}
```

(c) JDK-XXXX: `javac` infers the type of `v` as `double`. This leads to an UCTE.

Figure 11. Sample test programs that trigger typing bugs.

testing [34]. Contrary to this work, our program generator does not involve differential testing and focuses on typing compiler bugs where test cases act as their own oracle.

Dewey et al. [13, 14] have introduced a *constraint logic programming (CLP)* approach for synthesizing programs for JavaScript and Rust. The idea of the CLP-based program generation is to encode all syntactic and semantic rules (e.g., type system) of the language to logic predicates, and then use a constraint solver to generate test programs. Like Hephaestus, their fuzzing approach finds precision and soundness bugs. However, as stated by the authors, one of the fundamental shortcomings of CLP-based program generation and encoding typing rules into logic predicates, is poor performance. Moreover, our approach is (1) adaptable (already applied to three languages), (2) more effective (it identified more bugs than the fuzzer of Dewey et al. [14]), and (3) the first to validate type inference algorithms.

Transformation-Based Compiler Testing: *Equivalence Modulo Inputs (EMI)* [21, 22, 44] is an effective metamorphic testing [9] approach for finding bugs in optimizing compilers. EMI transforms a given program in a way that does not change its output under the same input. This is achieved by deleting dead statements [21], inserting code in dead regions [22], or even updating live parts [44]. EMI testing has been also ported to testing OpenCL compilers [26], and simulation software [12].

GLFuzz and spirv-fuzz [15, 16] repeatedly apply a set of semantics-preserving transformations (e.g., dead code injection) to an initial corpus of programs for finding bugs in graphics shader compilers. classfuzz [11] and classming [10] employ a set of transformations on existing Java bytecode programs to test JVM implementations through differential testing. Given a specific program structure, *skeletal program enumeration (SPE)* [48] enumerates all variant programs that expose different variable usage patterns.

SPE is complementary to our mutations. For example, instead of removing the maximal set of types, our type erasure mutation could employ SPE to enumerate all variant programs that manifest different patterns of omitted type information. Similarly, we could combine SPE with fault-injecting mutations (e.g., TOM), to identify what program points are promising to inject the error.

Inspired by SPE, Stepanov et al. [42] have designed *type-centric enumeration (TCE)*. TCE produces variants by assigning different values to variables or call arguments, while preserving the same type information as the original program. Unlike our work, TCE is effective in primarily finding crashes caused by back-end bugs. Another similar approach to TCE is *generative type-aware mutation* [38], which has been recently used for testing SMT solvers. Like TCE, generative type-aware mutation replaces an expression of an SMT formula with a newly-generated expression of the same type. A variant of this is *type-aware operator mutation* [46], which substitutes an SMT operator with another compatible operator. Instead of replacing expressions and operators, our type overwriting mutation replaces types. Also, the existing approaches (e.g., TCE) respect the semantics of the input program, while TOM is the first to adopt a fault-injecting approach, as an effort to find soundness bugs.

6 Conclusion

We have presented a systematic and extensible approach for finding typing bugs in diverse JVM compilers. We have introduced a program generator that constructs programs that are more likely to trigger typing bugs. Based on this generator, we have designed two novel transformation-based approaches for uncovering type inference and soundness compiler bugs. Within nine months of testing, our implementation, Hephaestus, has found 153 bugs (128 confirmed and 71 fixed) in the compilers of Java, Kotlin, and Groovy.

To reveal soundness or other types of bugs, additional sophisticated mutators can be developed on top of Hephaestus utilizing our analysis for capturing type information flow. For example, a promising direction could be the development of a mutation that targets bugs in the resolution algorithms of compilers, a category of bugs that is quite frequent [5]. Also, it would be interesting to apply our approach to other compilers, e.g., we already plan to extend Hephaestus to test the Scala and TypeScript compilers.

Our work is the first step towards more holistic compiler testing, as it fills the research gap in automated testing of static typing, a compiler procedure that deserves more attention by researchers.

References

[1] 2019. AFL compiler fuzzing. https://github.com/agroce/afl-compiler-fuzzer. Online accessed; 23-10-2021.

[2] 2021. Fuzzing LLVM libraries and tools. https://llvm.org/docs/FuzzingLLVM.html. Online accessed; 07-11-2021.

[3] 2021. Kotlin development stories. https://developer.android.com/kotlin/stories?linkId=94116374. Online accessed; 07-11-2021.

[4] 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

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

[6] 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

[7] Junjie Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie. 2016. 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

[8] 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

[9] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 1998. Metamorphic testing: a new approach for generating next test cases. *Technical Report HKUST-CS98-01* (1998).

[10] 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

[11] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. 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

[12] 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

[13] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2014. Language Fuzzing Using Constraint Logic Programming. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 725–730. https://doi.org/10.1145/2642937.2642963

[14] 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

[15] 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

[16] Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. 2021. Test-Case Reduction and Deduplication Almost for Free with Transformation-Based Compiler Testing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1017–1032. https://doi.org/10.1145/3453483.3454092

[17] EclEmma. 2021. EclEmma Jacoco. https://www.eclemma.org/jacoco/. Online accessed; 26-10-2021.

[18] Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. 2020. Closer to the Edge: Testing Compilers More Thoroughly by Being Less Conservative about Undefined Behaviour. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1219–1223. https://doi.org/10.1145/3324884.3418933

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

[20] Radu Grigore. 2017. Java Generics Are Turing Complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 73–85. https://doi.org/10.1145/3009837.3009871

[21] 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

[22] 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

[23] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Randomized Stress-Testing of Link-Time Optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 327–337. https://doi.org/10.1145/2771783.2771785

[24] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. 2018. Reconciling High-Level Optimizations and Low-Level Code in LLVM. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 125 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276495

[25] Xavier Leroy. 2006. Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant *(POPL '06)*. Association for Computing Machinery, New York, NY, USA, 42–54. https://doi.org/10.1145/1111037.1111042

[26] 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

[27] Fengyun Liu, Ondřej Lhoták, Aggelos Biboudis, Paolo G. Giarrusso, and Martin Odersky. 2020. A Type-and-Effect System for Object Initialization. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 175 (Nov. 2020), 28 pages. https://doi.org/10.1145/3428243

[28] 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

[29] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 65–79. https://doi.org/10.1145/3453483.3454030

[30] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. 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, 22–32. https://doi.org/10.1145/2737924.2737965

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

[32] Julian Mackay, Alex Potanin, Jonathan Aldrich, and Lindsay Groves. 2020. Syntactically Restricting Bounded Polymorphism for Decidable Subtyping. In *Programming Languages and Systems*, Bruno C. d. S. Oliveira (Ed.). Springer International Publishing, Cham, 125–144.

[33] 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

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

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

[36] 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

[37] OpenJDK. 2021. OpenJDK Jir deployment. https://bugs.openjdk.java.net. Online accessed; 26-10-2021.

[38] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. Generative Type-Aware Mutation for Testing SMT Solvers. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 152 (Oct. 2021), 19 pages. https://doi.org/10.1145/3485529

[39] Moritz Pflanzer, Alastair F. Donaldson, and Andrei Lascu. 2016. Automatic Test Case Reduction for OpenCL. In *4th International Workshop on OpenCL (IWOCL'16)* (Vienna, Austria).

[40] Benjamin C. Pierce. 1992. Bounded Quantification is Undecidable. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico, USA) *(POPL '92)*. Association for Computing Machinery, New York, NY, USA, 305–315. https://doi.org/10.1145/143165.143228

[41] 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

[42] Daniil Stepanov, Marat Akhin, and Mikhail Belyaev. 2021. Type-Centric Kotlin Compiler Fuzzing: Preserving Test Program Correctness by Preserving Types. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 318–328.

[43] Chengnian Sun, Vu Le, and Zhendong Su. 2016. 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

[44] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 849–863. https://doi.org/10.1145/2983990.2984038

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

[46] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 193 (Nov. 2020), 25 pages. https://doi.org/10.1145/3428261

[47] 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

[48] 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