Finding Typing Compiler Bugs

Stefanos Chaliasos∗
Imperial College London
United Kingdom
s.chaliasos21@imperial.ac.uk

Thodoris Sotiropoulos∗
Athens University of Economics and Business
Greece
theosotr@aueb.gr

Diomidis Spinellis
Athens University of Economics and Business
Greece
dds@aueb.gr

Arthur Gervais
Imperial College London
United Kingdom
a.gervais@imperial.ac.uk

Benjamin Livshits
Imperial College London
United Kingdom
b.livshits@imperial.ac.uk

Dimitris Mitropoulos
University of Athens
Greece
dimitro@ba.uoa.gr

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 156 bugs (137 confirmed and 85 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.

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

Keywords: compiler bugs, compiler testing, static typing, Java, Kotlin, Groovy

∗These authors contributed equally.

ACM Reference Format:

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 156 bugs in all the examined compilers, of which 85 bugs were subsequently fixed by developers. Thanks to type erasure and type overwriting mutations, we have uncovered 52 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.
- 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.

\(^1\)In Greek mythology, Hephaestus was the smithing god.
public class Test {
    void test() {
        def closure = { new B<> {(new A<Long>())} }
        A<Long> x = closure().f
        5 }
    6 }
    class A<T> {}
    class B<T> {
        T f;
        B<T f> { this.f = f; }
    }
}

Figure 1. GROOVY-10080: This well-typed program is rejected by the Groovy compiler.

From February 2021 to mid-November 2021, HEPHAESTUS has found 156 bugs in total, of which 11 bugs are in javac, 32 are in kotlinc, and 113 in groovyc.

Availability: HEPHAESTUS is available as open-source software under the GNU General Public License v3.0 at https://github.com/hephaestus-compiler-project/hephaestus. The research artifact is available at https://doi.org/10.5281/zenodo.6410434.

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<\(\text{A<\text{Long}}\)>. Although the type argument of B is omitted on line 3 (via the diamond operator \(<\text{>\text{>})\text{, the compiler should be able to infer the corresponding type parameter from the type of the constructor's argument, which is A<\text{Long}}\). 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<\(\text{A<\text{Long}}\)>. Surprisingly, replacing A<\text{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 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.

### 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 \( t \) (\( \text{val}(t) \)) 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 \( (T : t) \) labeled with a name \( T \) 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 \( \prec \) to denote the subtyping relation, and the operation \( S(t) \) to give the supertype of type \( t \) (e.g., \( S(T : t) = t) \).

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

\[
\begin{align*}
[\alpha \mapsto t] & \; \alpha = t \\
[\alpha_1 \mapsto t_1][\alpha_2 \mapsto t_2] & = [\alpha_1 \mapsto [\alpha_2 \mapsto t_2]]
\end{align*}
\]

\( \alpha_1 \neq \alpha_2 \prime \)

\[
\begin{align*}
[\alpha \mapsto t_1]T & = T' : [\alpha \mapsto t_1]t_2 \\
[\alpha \mapsto t_1][\Lambda \alpha . t_2] & = [\alpha \mapsto t_1]t_2 = (\Lambda \alpha . t_2)t_1 \\
[a_1 \mapsto t_1][\Lambda \alpha_2 . t_2]t_3 & = ([\Lambda \alpha_2 . t_2][a_1 \mapsto t_1])t_3
\end{align*}
\]

A type substitution replaces all occurrences of a type parameter \( \alpha \) in a type \( t_1 \) with another type \( t_2 \).

\[
\begin{align*}
\langle p \in \text{Program} \rangle & ::= \hat{d} \\
\langle d \in \text{Decl} \rangle & ::= \text{class } C \text{ extends } e \; \hat{d} \\
& \quad | \text{fun } m(x: t \ldots) : t = e \\
& \quad | \text{var } x : t : e ; \Lambda \alpha . d \\
\langle e \in \text{Expr} \rangle & ::= \text{val}(t) \mid x \mid e.f \mid e @ e \mid \{d \ldots e \ldots \} \\
& \quad | (e.m t)(e \ldots) \mid (\text{new } C t)(e \ldots) \\
& \quad | e.x = e \mid \text{if}(e) \text{else } e \mid e : m \\
& \quad | \lambda x : t e
\end{align*}
\]

\( (\oplus \in \text{BinaryOp}) \) is the set of variable and field names, \( (m \in \text{MethodName}) \) is the set of method names, \( (C \in \text{ClassName}) \) is the set of class names.

(a) Syntax of the IR.

\[
\begin{align*}
\langle t \in \text{Type} \rangle & ::= T \mid \bot \mid \alpha \mid T : t \\
& \quad | \Lambda \alpha . t \mid (\Lambda \alpha . t) \; t
\end{align*}
\]

(b) Types in the IR.

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

**Definition 3.2.** (Type unification) The type unification \( \text{Type} \times \text{Type} \rightarrow S \) is an operation that takes two types \( (t_1, t_2) \in \text{Type} \) and returns a type substitution \( \sigma \) so that \( \sigma t_1 \prec t_2 \):

\[
\begin{align*}
\text{unify}(\alpha, t) & = [\alpha \mapsto t] \\
\text{unify}((\Lambda \alpha . t)t_1, (\Lambda \alpha . t)t_2) & = \text{unify}([\alpha \mapsto t_1]t, [\alpha \mapsto t_2]t) \\
\text{unify}(t_1, t_2) & = \text{unify}(t_1, S(t_2)) \; t_1 \notin \text{TypeParam}
\end{align*}
\]

Type unification identifies a substitution \( \sigma \) so that the type \( \sigma t_1 \) 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., `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 `r < 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 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 ∈ V` is either a declaration `d ∈ Decl`, or a type `t ∈ Type`. The set of edges \( E \subseteq V \times V \times L \), where
TYPE APPLICATION
\[ t = (\Lambda a.t_1)_{t_2} \]
\[ A(G, t) \Rightarrow G \cup \{(\Lambda a.t_1)_{t_2} \rightarrow a \}_{decl}, a \rightarrow t_1 \}_{def} \]

VAR DECL
\[ e = \text{val } x : t_1 = e \quad t_2 = \text{getType}(e) \]
\[ A(G, e) \Rightarrow G' \cup \{(x \rightarrow t_1, x \rightarrow t_2) \}_{decl} \]

VAR PARAM CONSTRUCTOR
\[ e = \text{val } x : t_1 = \text{new } (C_{t_2})(\cdot) \]
\[ \Lambda a.t = \text{toType}(C) \quad \sigma = \text{unify}'(t_1, (\Lambda a.t)_{t_2}) \]
\[ A(G, e) \Rightarrow G \cup \{(x \rightarrow t_1, x \rightarrow (\Lambda a.t)_{t_2}) \cup \{a \rightarrow \sigma(a) | a \in \sigma \} \}_{decl} \]

VAR PARAM METHOD CALL
\[ e = \text{val } x : t_1 = e_1.(m)() \]
\[ t_2 = \text{getType}(e_1, m) \]
\[ \text{typeParam}(m) \in t_2 \quad \sigma = \text{unify}'(t_1, t_2) \]
\[ A(G, e) \Rightarrow G \cup \{(x \rightarrow t_1, x \rightarrow \text{getType}(e_1)) \cup \{a \rightarrow \sigma(a) | a \in \sigma \} \}_{decl} \]

PARAM CALL
\[ e = e_1.(m)(e_2) \]
\[ t_1 = \text{getType}(e_2) \quad t_2 = \text{getParamType}(e_1, m) \]
\[ \text{typeParam}(m) \in t_2 \quad \sigma = \text{unify}'(t_1, t_2) \]
\[ A(G, e) \Rightarrow G \cup \{a \rightarrow \sigma(a) | a \in \sigma \} \}_{decl} \]

Figure 5. Analysis rules for building type graphs.

\[ L = \{\text{decl}, \text{inf}, \text{def}\} \]
indicate the following: given a type graph \( G \), the edge \( n \rightarrow t \) denotes that the type of node \( n \) is explicitly declared in the program as \( t \). For example, for a variable declaration of the form \( x = \ldots \), there is a \( x \rightarrow t \) edge, where \( t = \text{String} \). The edge \( n_1 \rightarrow n_2 \) indicates that the type of node \( n_1 \) is inferred by node \( n_2 \). For example, for an assignment of the form \( \text{String} = "\text{str}" \), there is also an \( a \rightarrow t \) edge, where \( a = \text{String} \). This is because the type of variable \( x \) is inferred as \( \text{String} \), which is the type of the constant at the right-hand side of the assignment. Finally, the edge \( t_1 \rightarrow t_2 \) shows that type \( t_1 \) consists of another type \( t_2 \). For each type application of the form \( t_1 = \text{String} \), we have the edges \( t_1 \rightarrow \text{T} \) and \( \text{T} \rightarrow t_2 \), where \( t_2 = \text{String} \). These edges indicate that parameterized type \( \text{String} \) contains type parameter \( \text{T} \), and the corresponding type argument is \( \text{String} \).

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, \( \text{unify}' \) is a variant of type unification that adds the following rules:

\[ \text{unify}'(\langle \Lambda a.t \rangle_{t_1}, \langle \Lambda a.t \rangle_{t_2}) = [a \rightarrow a] \text{ if } t_1 = t_2 \]
\[ \text{unify}'(\langle \Lambda a.t \rangle_{t_1}, \langle a_a, t_2 \rangle_{t_2}) = [a_a \rightarrow a] \]
\[ \text{if } \text{unify}(S(t), \langle a_1 \rightarrow t_1 \rangle) \neq 0 \land \]
\[ \langle a_2 \rightarrow t_2 \rangle \]

In essence, this modification finds dependent type parameters between two type applications. For example, suppose we have (1) two parameterized classes: \( \text{class } A<T> \) and \( \text{class } B<T> \) extends \( A<T> \), and (2) two type applications derived from these classes, namely, \( t_1 = A<\text{String}> \) and \( t_2 = B<\text{String}> \). In this scenario, \( \text{unify}'(t_1, t_2) \) returns \( [a_2 \rightarrow a_1] \), where \( a_1 \) and \( a_2 \) are the type parameters of type constructors \( A \) and \( B \) respectively. This dependency information indicates that instantiating the type parameter \( a_2 \) with a type \( t_2 \) also instantiates \( a_1 \) with the same type, as \( a_2 \) flows to the type parameter of superclass.

\[ \text{[TYPE APPLICATION]}: \text{For each type application } t = \langle \Lambda a.t \rangle_{t_2}, \text{the resulting type graph contains two edges. The first edge } (\rightarrow) \text{ connects type application with the underlying type parameter } a, \text{and the second edge } (\rightarrow) \text{ connects } a \text{ with type argument } t_2. \]

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

\[ \text{[VAR PARAM CONSTRUCTOR]}: \text{For a variable initialized by a parameterized constructor (e.g., } A<\text{String}> x = \text{new } A<\text{String}>(\cdot)\), \text{by adding the edges for } a \text{ connected to the declared } \text{and inferred types of variable } x, \text{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 } (\rightarrow) \text{ 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<\text{String}> x = \text{new } A<>()\).} \]

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

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

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
open class A<T>
class B<T>(
val f: A<T>
):
A<T>(
)

fun m(): A<String> {
    return B<String>(
        A<String>()
    )
}
returns a new type graph. It is defined as:
\[
erasure(G, α) = G \setminus \{ α \rightarrow n \}, \ n \in G
\]
\[
erasure(G, (A.α.t₁)₁₂) = erasure(G, α)
\]
\[
erasure(G, t) = G \text{ if } t \notin TypeParam
\]
Type erasure takes a node \( n \) and a graph \( G \), and removes all \( \rightarrow \) edges associated with the given node \( n \). Conceptually, type erasure removes variables’ declared types, or types used as type arguments from the corresponding type parameters.

Based on the \textit{infer} and \textit{erasure} operations, we now present the \textit{type preservation} and \textit{type relevance} properties.

\textbf{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 \( \text{infer}(G, n) = t \), \( G' = \text{erasure}(G, n) \) and \( \text{infer}(G', n) = t \).

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.

\textbf{Definition 3.6.} (Generalized type preservation) Given a type graph \( G = (V, E) \) and nodes \( n₁, n₂ \ldots nₙ \in V \), we say that these nodes preserve their types when \( t₁ = \text{infer}(G, n₁), \ldots, tₙ = \text{infer}(G, nₙ) \), \( G' = \bigcap_{i=0}^{n} \text{erasure}(G, n_i) \) and \( t₁ = \text{infer}(G', n₁), \ldots, tₙ = \text{infer}(G', nₙ) \).

\textbf{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 \( G' = \text{erasure}(G, n) \) and \( \text{infer}(G', n) \prec t \).

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.

\textbf{3.4.1 Type Erasure Mutation.} The insight of the \textit{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., \texttt{var x = 1}), (2) removing type arguments from a parameterized constructor or method call (e.g., \texttt{new A<>\(\text{"f"}\)}), (3) removing a return type from a method’s signature (e.g., \texttt{fun m() = \text{"f"}}), and (4) removing a type from a parameter of a lambda (e.g., \( x \rightarrow x + 1 \)).


Algorithm 2: Algorithm for type erasure mutation.

```plaintext
fun typeErasureMutation(P) =
    for m ∈ Methods(P) do
        G ← A(∅, m) // Builds type graph
        nodes ← findCandidateNodes(G)
        nodes ← {n ∈ nodes | n preserves its type on G}
        for k = len(nodes) to 1 do
            for (n1, n2, ..., nk) ∈ combination(nodes, k) do
                if (n1, n2, ..., nk) preserve their type on G then
                    erase (n1, n2, ..., nk)
                    break
    Algorithm 2: Algorithm for type erasure mutation.
```

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:

```plaintext
class A<T> { val x: Any = A<String>()
2   val y: A<String> = A(x)
3   fun A<T>() = A<String>() // B<T>:
4     return B<String>(A<String>())
5   }
6
7   fun typeErasureMutation(P) =
8       for m ∈ Methods(P) do
9           G ← A(∅, m) // Builds type graph
10          nodes ← findCandidateNodes(G)
11          nodes ← {n ∈ nodes | n preserves its type on G}
12          for k = len(nodes) to 1 do
13              for (n1, n2, ..., nk) ∈ combination(nodes, k) do
14                  if (n1, n2, ..., nk) preserve their type on G then
15                      erase (n1, n2, ..., nk)
16                      break
17     }

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 t₁ with another type t₂ 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., $\text{infer}(G, n) = \text{String} \neq \text{Int}$). The output of TOM is then an updated program where the body of $m$ is return $\text{B<String>}<(A<\text{Int}>())$. 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. All programs of each batch are then passed as input to the compiler under test. Note that to avoid conflicting declarations, every program in a batch is placed in a dedicated package, i.e., the corresponding translator puts a package statement at the beginning of each source file. 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.

### 3.6 Discussion

**Generalizability:** Currently, our implementation, generates programs written in three popular languages: Java, Kotlin, and Groovy. However, our approach is not limited to JVM languages. Our techniques also apply to statically-typed, object-oriented languages that support type inference or parametric polymorphism/generics, such as Scala, C#, TypeScript, Swift, and more. Supporting a new target language requires little engineering effort. Specifically, two components are necessary: (1) a translator to convert a program written in IR into a concrete program written in the target language (existing translators contain roughly 800 LoC), and (2) a regular expression that distinguishes compiler crashes from compiler diagnostic messages. For testing language-specific features (e.g., Scala implicits, pattern matching), Hephaestus’ IR should be extended as well.

Although our initial focus is to test the type checkers of statically-typed languages, Hephaestus’s program generator can also be used to produce programs written in dynamic languages (e.g., Python, JavaScript) by implementing the corresponding translators.

**Necessity of program generation:** A reader may wonder whether our program generator is actually necessary or not. Actually, our program generator is not necessary for using our mutations. That is, one can apply the mutations directly to existing code (e.g., compiler test suites). However, this would require a parser implementation for each target language because the mutations operate on our IR. Furthermore, as we will observe in our bug-finding results (Section 4.2, Figure 7c), our program generator (1) is very efficient in discovering typing bugs on its own, and (2) improves the effectiveness of mutations by providing them with good seeds.

### 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 156 bugs, of which 137 are confirmed, and 85 have been already fixed by developers. Interestingly, Hephaestus was able to find bugs in all the examined compilers: 113 bugs in groovyc, 32 bugs in kotlin, and 11 bugs in javac.
- **RQ2:** Hephaestus finds typing bugs. Hephaestus found 147 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 52 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.

Settings of test program generation: Producing very large programs can decrease throughput, as compilers require more time to process input programs and Hephaestus’s internal algorithms (e.g., type inference analysis) take longer to proceed. Based on our exploratory experiments, generating programs with up to ten top-level declarations, and generating expressions with depth up to seven give us a good balance between bug-finding results and performance. Other settings considered during our testing campaign include the maximum number of type parameters per parameterized class/function (three), the maximum number of local variable declarations (three), and the maximum number of parameters per method (two). Using the above settings, Hephaestus results in programs consisting of 500–1000 LoC.

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 156 bugs. The developers confirmed most of them (137/156) as previously unknown, real bugs, while they have already fixed 85 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, 7 out of 156 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 11f).

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
shows the number of bugs triggered by the genera-
characterizes the bugs by their symptoms. Most
versions they affect.

4.3 RQ2: Bug and Test Case Characteristics

Overall, developers classified 147 out of our 156 (94%) dis-
covered bugs, as bugs whose root cause reside in compilers’
static typing procedures. We also identified seven back-end
bugs. All of them were assertion failures or other runtime ex-
ception (e.g., NullPointerException) that were observed
during code generation or optimizations, e.g., mishandling
of bounded type parameters in code generation, erroneous
optimizations corrupted the stack. Finally, Hephaestus
detected two parser/lexer bugs.

Figure 7b characterizes the bugs by their symptoms. Most
of the discovered bugs (104/156) 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 pro-
grams. 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 polymor-
phism (e.g., parameterized class) are in the list of features
with the most bug-revealing capability. In total, 106/156 bugs
are caused by programs containing at least one such fea-
ure. 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 63 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 in-
fERENCE (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.

4.4 RQ3: Effectiveness of Mutations

Figure 7c shows the number of bugs triggered by the genera-
tor, the mutators, and their combination. Our mutations led
to the identification of 78 out of 156 bugs, about half of the
total discovered bugs. Our generator fails to detect these 78
bugs, as they are all related to either type inference issues or
other issues triggered by wrongly-typed code. TEM is an ef-
fective approach, able to identify 52 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. Note that,
even though soundness bugs may stem from either deep
issues in the language or type system design [5], all the dis-
covered soundness bugs point to implementation-related
errors.

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 compi-
lers, 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 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

---

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

---

**Table 4.** Coverage increase by type erasure (TEM) and type overwriting (TOM) mutations.

---

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

---

is only +0.06%, HEPHAESTUS was able to find 113 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:** Groovy supports flow typing as an extension of type inference. The idea behind flow typing is that the compiler infers types of variables based on the flow of a program. In this program, the type of variable x is A<String> at line 6 and Object at line 8. Nevertheless, when groovyc type checks line 7, it erroneously uses Object as the declared type of x, thereby rejecting the program. This bug was found by TEM.

**Figure 11d:** In this example, the inferred type of the expression if (true) B() else E() is the intersection type A & !<out Any>. Since kotlinc uses intersection types only internally, the intersection type is transformed into a type that is representable in the program. In this context, kotlinc first approximates this intersection type to type Any (Any is the counterpart of Object), and then checks it against the return type of the overridden method, which is A. This makes the compiler reject that program, as Any is not compatible with A. Notably, in the discussion of this
Figure 11: Sample test programs that trigger typing bugs.

(a) GROOVY-10324: A bug in the inference engine of groovyc leads to an UCTE.

(b) KT-49101: A crash found in kotlinc due to a bug in type constructor projection.

(c) GROOVY-10308: A bug in the flow typing algorithm of groovyc that causes an UCTE.

(d) KT-44082: kotlinc mistakenly approximates the return type of the overriding method foo.

(e) GROOVY-10127: groovyc does not catch the error at line 8. This results in a URB.

(f) JDK-8269348: javac infers the type of v as double. This leads to an UCTE.

Figure 11e: groovyc erroneously accepts the program of Figure 11e, and produces a binary that breaks the type safety of the language. In particular, the compiler should have reported a compile-time error at line 8, which would indicate that A cannot be converted to type T (as A is not a subtype of T). At runtime, this incorrect compilation leads to a MissingMethodException, when executing the call at line 12. This bug was found by TOM.

Figure 11f: 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 any 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 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-aware 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 156 bugs (137 confirmed and 85 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.

Acknowledgments
We would like to thank our shepherd John Regehr, the anonymous PLDI reviewers, and Alastair Donaldson for their insightful feedback on previous versions of the paper. We also want to thank the Groovy developers, Eric Miles and Paul King, for addressing our bug reports quickly, and assisting us in incorporating JaCoCo coverage reporting into Groovy build scripts. Finally, we want to thank the Kotlin developer, Victor Petukhov, and the various Kotlin support engineers for fixing and triaging our bugs reports. This work has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 825328.

References


