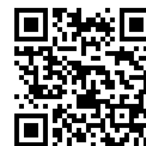
软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007572] [CSTR: 32375.14.jos.007572]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于语法和类型变异的 TypeScript 编译器缺陷检测*



任志磊, 高 越, 周志德, 敖 伟, 江 贺

(大连理工大学 软件学院, 辽宁 大连 116024)
通信作者: 任志磊, E-mail: zren@dlut.edu.cn

摘　要: TypeScript 作为 JavaScript 的超集, 提供了静态类型支持和面向对象等多种特性, 被 Angular、Vue、React 等众多主流框架广泛采用, 成为构建大型应用的核心技术之一. 其编译器负责将 TypeScript 代码编译为标准的 JavaScript 代码. 然而, TypeScript 编译器本身可能存在缺陷, 导致生成的 JavaScript 代码包含难以预料的错误. 尽管在 JavaScript 引擎测试方面已有诸多研究, 但尚未有针对 TypeScript 编译器的系统性测试研究. 现有的 JavaScript 引擎测试方法既难以生成大量包含 TypeScript 特定类型的程序, 也无法有效变异这些类型, 导致难以检测 TypeScript 编译器中与复杂类型系统相关的缺陷. 为此, 提出一种基于语法和类型变异的 TypeScript 编译器测试框架 TscFuzz. 为了获取大量包含 TypeScript 特定类型的种子程序, TscFuzz 针对 TypeScript 相较于 JavaScript 的特殊类型设计了一组提示词, 并引导大语言模型生成一系列包含特定类型的程序. 然后, 设计了一组类型特定的变异算子, 旨在通过类型变异对 TyppScript 的类型系统进行针对性的测试. 最后, TscFuzz 基于交叉版本策略的差分测试, 比较不同版本的 TypeScript 编译器的输出结果来检测其缺陷, 并通过 Node.js 验证编译器输出 JavaScript 程序的语义正确性. 实验结果显示, TscFuzz 在 72 h 内发现了 5 个缺陷, 比基线方法 DIE 和 FuzzJIT 分别多检测了 2 个和 3 个 bug. TscFuzz 的故障检测效果显著优于基线方法. 同时, 经过 3 个月的测试, TscFuzz 发现了 12 个真实的 TypeScript 缺陷, 其中 8 个已被开发者确认, 7 个已被修复.
关键词: TypeScript 编译器测试; 大语言模型; 类型系统; 差分测试
中图法分类号: TP311

中文引用格式: 任志磊, 高越, 周志德, 敖伟, 江贺. 基于语法和类型变异的TypeScript编译器缺陷检测. 软件学报. http://www.jos.org.cn/1000-9825/7572.htm
英文引用格式: Ren ZL, Gao Y, Zhou ZD, Ao W, Jiang H. TypeScript Compiler Bug Detection Based on Syntax and Type Mutation. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7572.htm

TypeScript Compiler Bug Detection Based on Syntax and Type Mutation

REN Zhi-Lei, GAO Yue, ZHOU Zhi-De, AO Wei, JIANG He

(School of Software Technology, Dalian University of Technology, Dalian 116024, China)

Abstract: As a superset of JavaScript, TypeScript provides a rich set of features, such as static type support and object-oriented programming capabilities. It is widely adopted by many mainstream frameworks such as Angular, Vue, and React, and has become a core technology for building large-scale applications. Its compiler is responsible for compiling TypeScript codes into standard JavaScript codes. However, the TypeScript compiler itself may contain bugs, resulting in unexpected errors in the generated JavaScript code. Although numerous studies have been conducted on JavaScript engine testing, there has been no systematic study dedicated to testing the TypeScript compiler. Existing JavaScript engine testing methods have difficulty in generating a large number of TypeScript programs with specific types and effectively mutating these types, thus making it difficult to detect bugs related to complex type systems in the TypeScript compiler. To this end, a TypeScript compiler testing framework based on syntax and type mutation TscFuzz is proposed. To obtain a large

* 基金项目: 国家自然科学基金 (62132020, 62032004); 国家自然科学基金青年科学基金 (62302077); 中央高校基本科研业务费专项资金 (DUT24LAB126)
收稿时间: 2025-05-13; 修改时间: 2025-09-23; 采用时间: 2025-11-03; jos 在线出版时间: 2026-02-04

number of seed programs containing specific types of TypeScript, TscFuzz designs a set of prompts tailored to the unique type system of TypeScript compared to JavaScript, with the large language model (LLM) guided to generate a series of programs featuring these specific types. Next, a set of type-specific mutation operators are designed to conduct targeted testing on the type system of TypeScript via type mutation. Finally, based on differential testing of the cross-version strategy, TscFuzz compares the outputs of different versions of the TypeScript compiler to detect bugs. Additionally, Node.js is employed to verify the semantic correctness of the JavaScript programs output by the compiler. Experimental results demonstrate that TscFuzz detects five bugs within 72 hours, two and three bugs more than the baseline methods DIE and FuzzJIT, respectively. The bug detection effect of TscFuzz is significantly better than that of the baseline methods. Meanwhile, after three months of testing, TscFuzz successfully identifies 12 real TypeScript bugs, eight of which have been confirmed and seven have been repaired.

Key words: TypeScript compiler testing; large language model (LLM); type system; differential testing

TypeScript 是微软开发的开源编程语言, 作为 JavaScript 的超集, 它通过引入静态类型和面向对象特性, 有效弥补了 JavaScript 的不足, 极大增强了代码的可维护性和可读性, 显著提升了开发大型应用程序的效率和可靠性[1,2], 在现代软件开发中占据着重要地位. TypeScript 编译器负责将 TypeScript 代码转换为标准的 JavaScript 代码 (如图 1 所示), 使其能够在支持 JavaScript 的平台上运行, 并与众多流行的前端和后端框架 (如 Angular、React、Vue.js、Node.js 等) 兼容[3–7].

```
const greet = (name: string): string => { return `Hello, ${name}!`; };
console.log(greet("Alice"));
```

(a) 编译前的TypeScript代码

```
var greet = function (name) { return "Hello, " + name + "!"; };
console.log(greet("Alice"));
```

(b) 编译后的JavaScript代码

图 1　TypeScript 编译器编译前后代码示例

然而, TypeScript 编译器可能存在缺陷, 导致无法生成 JavaScript 代码或生成错误的 JavaScript 代码, 从而引发一系列下游问题, 影响与 JavaScript 工具和框架的兼容性. 这不仅会导致开发过程中的错误难以排查, 还可能降低整个软件系统的稳定性和可靠性. 因此, 保障 TypeScript 编译器的质量对于提高软件系统的可靠性和提升与 JavaScript 的兼容性至关重要.

编译器测试在发现和修复错误、提升软件系统质量方面发挥着关键作用[8,9]. 但目前针对 TypeScript 编译器的测试, 主要借鉴 JavaScript 引擎的测试方法[10–38], 其中一些典型的方法包括 DIE[22]、Montage[23]和 FuzzJIT[24]. DIE 提出了名为"方面保留 (aspect-preserving)"的变异方法, 通过在生成新测试用例时随机保留原始种子输入中的有益属性和条件来获取高质量的测试用例[22]. 虽然该方法能够在一定程度上保留原始代码的类型和结构, 但它无法有效地获取一组包含丰富静态类型的 TypeScript 种子程序, 从而无法生成能够充分覆盖 TypeScript 类型系统的测试程序. Montage 将抽象语法树 (AST) 转换为 AST 子树序列, 通过神经网络语言模型 (neural network language model, NNLM) 学习 AST 片段之间的关系, 并通过替换部分代码来变异 JavaScript 代码生成新的测试用例[23]. 然而, Montage 的变异策略并不针对 TypeScript 特有的静态类型, 且其变异算法较为简单, 无法有效变异 TypeScript 代码. FuzzJIT 设计了一种输入包装模板和一组与即时编译器 (just in time, JIT) 相关的变异策略, 主动激活 JIT 编译器以捕获由 JIT 编译器引起的缺陷[24], 然而, 由于 TypeScript 编译器没有 JIT 编译器组件, 且 FuzzJIT 的变异算法依赖一个特定于 JavaScript 的工具——Fuzzilli[36]实现, 因此, 该方法也不适用于 TypeScript 编译器的测试.

由于 TypeScript 编译器与 JavaScript 引擎的结构存在显著差异, 因此两者的测试工作及流程也存在一些重要的区别, 具体如下.

(1) 测试重点不同. JavaScript 作为一种动态语言, 其测试重点通常集中在解析器和 JIT 编译器上[24]; 而 TypeScript 作为静态类型语言, 具有复杂的类型系统, 包含泛型、联合类型、交叉类型和命名空间等多种区别于 JavaScript 的特性, 应重点关注 TypeScript 编译器在处理复杂类型时可能产生的缺陷.

(2) 测试用例目标不同. 现有的 JavaScript 引擎测试方法以生成语义正确的测试程序为目标, 但这些程序并未针对 TypeScript 的静态类型进行设计. 为了更好地覆盖 TypeScript 的类型特性, 需要获取大量包含 TypeScript 特

定类型的测试程序.

(3) 验证方式不同. JavaScript 引擎通常通过直接运行程序, 观察执行结果或崩溃现象来发现缺陷. 然而, TypeScript 编译器的输出是 JavaScript 代码, 仅观察输出结果无法全面验证编译过程的完整性和正确性, 还需要进一步验证生成的 JavaScript 代码能否正确执行.

基于目前 JavaScript 引擎测试方法的局限性以及 TypeScript 编译器与 JavaScript 引擎在测试方面的显著差异, 本文提出了一种基于语法和类型变异的 TypeScript 编译器检测方法 TscFuzz, 主要解决以下两个方面的挑战.

挑战 1. 如何获取包含特定类型的 TypeScript 种子程序. 获取包含丰富静态类型的 TypeScript 种子程序是测试 TypeScript 编译器的基础. 然而, 现有的种子程序生成方法难以系统性地提取涵盖 TypeScript 静态类型 (如泛型、联合类型和交叉类型等) 的程序, 导致测试范围受限, 无法对 TypeScript 的类型系统进行充分的验证. 因此, 如何结合 TypeScript 的语法特性, 自动化地获取包含特定类型的高质量种子程序, 成为亟需解决的问题.

挑战 2. 如何对 TypeScript 程序进行变异以生成具有特定类型的多样化测试程序. TypeScript 编译器的类型系统比 JavaScript 更为复杂, 变异策略需要在保留程序静态类型正确性的同时, 生成能够充分挑战类型检查逻辑的测试用例. 因此, 如何设计一组专门的变异算法, 能够结合 TypeScript 特性对种子程序进行细粒度变异, 确保生成的变异程序能够更全面地覆盖 TypeScript 编译器的类型检查逻辑, 是一个核心挑战.

为了解决上述挑战, 本文设计了针对性的策略并实现了 TscFuzz 方法. 首先, 针对第 1 个挑战, 本文通过对比 TypeScript 和 JavaScript 语法, 归纳出 11 种 TypeScript 特有的静态类型特性, 并设计了一套用于引导大语言模型生成种子程序的提示词模板. 通过填充特定语法类型, 生成了 11 个精确提示词, 成功引导大语言模型生成高质量、覆盖特定类型的 TypeScript 种子程序. 其次, 针对第 2 个挑战, TscFuzz 设计了两组共计 21 种变异规则, 分别适用于常规语法和特定类型语法的变异处理. 具体而言, 对于给定的种子程序, TscFuzz 首先使用 ANLTR4[39]将其转化为解析树, 并通过遍历解析树中的节点, 识别适合变异的目标节点. 为确保变异操作的有效性, TscFuzz 设计了一种动态变异算法: 若当前解析树无法匹配指定的变异目标, 则在限定的迭代次数内调整变异目标或变异规则, 以生成有效的变异程序. 这种策略显著提高了种子程序变异的效率和针对性.

为了评估 TscFuzz, 我们在 TypeScript 编译器上进行了为期 3 个月的实验. 实验结果显示, TscFuzz 发现了 12 个缺陷, 其中 8 个已被开发者确认, 7 个已被修复. 此外, 我们还将 TscFuzz 与目前最先进的两个 JavaScript 引擎测试工具进行了对比. 实验结果表明, TscFuzz 在 72 h 内发现了 5 个缺陷, 比基线方法 DIE 和 FuzzJIT 分别多检测了 2 个和 3 个 bug. TscFuzz 的故障检测效果显著优于基线方法. 此外, 本文还通过实验验证了 TscFuzz 每个组件的有效性以及 TscFuzz 生成程序的复杂度和多样性.

综上所述, 本文的主要贡献总结如下.

(1) 针对 TypeScript 静态类型系统进行 TypeScript 编译器缺陷检测, 研究成果为 TypeScript 工具链分析提供了新的研究方向.

(2) 提出一种有效的测试方法来验证 TypeScript 编译器, 将大语言模型应用到 TypeScript 测试程序生成中, 提出一组类型特定的变异算法, 并实现了 TscFuzz 框架.

(3) 实验评估表明, 相较于主流的 JavaScript 引擎测试工具, 本文方法能够生成具有特定类型的测试用例, 更有效地检测 TypeScript 编译器缺陷. 在为期 3 个月的测试中, 共发现了 12 个 TypeScript 编译器缺陷, 其中 8 个已被开发者确认, 7 个已被修复, 为开源社区的发展做出了贡献.

本文第 1 节介绍研究背景. 第 2 节介绍本文提出的方法框架 TscFuzz. 第 3 节通过对比实验验证所提方法的有效性. 第 4 节讨论和分析相关工作. 第 5 节总结全文.

1 研究背景

1.1 术语解释

TypeScript 编译器: 本文的核心测试目标, 它是一个将 TypeScript 程序转换为标准 JavaScript 程序的工具, 包含语法分析、类型检查和代码生成等关键模块.

TypeScript 语言: 作为 TypeScript 编译器的输入, 它是一种包含了静态类型语法的 JavaScript 超集.

TypeScript 静态类型系统: TypeScript 编译器的核心功能模块之一, 它负责对输入代码进行类型检查、类型推断与类型验证, 是本文缺陷检测的重点关注对象.

TypeScript 测试程序: 用于验证编译器功能的输入用例, 本质是使用 TypeScript 语言编写的代码. 本文的测试程序来源于两个方面: 一是从 GitHub 开源平台收集的真实项目用例; 二是通过大语言模型生成的包含特定类型特征的用例.

1.2 TypeScript 编译器缺陷

在 TypeScript 编译器将 TypeScript 代码转换为标准 JavaScript 代码的过程中, TypeScript 编译器会首先进行语法分析和类型检查, 只有通过语法和类型检查的程序才能生成对应的 JavaScript 代码. TypeScript 编译器应确保编译前后程序片段的语义一致. 图 1 给出了一个编译成功的代码示例, 编译前后的代码均定义了一个名为 greet 的函数, 接受一个 name 参数, 并返回一个问候语 (字符串), 同时将函数调用的结果输出到控制台. 在定义 greet 函数时, TypeScript 代码使用了箭头函数, 并给变量和函数添加了类型注解, 而 JavaScript 代码则使用了常规函数且没有类型注解, 但两者的语义完全相同.

然而, 在编译过程中, TypeScript 编译器难免会存在一些缺陷. 根据其工作流程, 这些缺陷主要分为语法分析缺陷、类型检查缺陷和代码生成缺陷这 3 类. 语法分析缺陷通常表现为编译器错误地拒绝语法正确的程序, 或未能正确解析和输出与语法分析相关的信息; 类型检查缺陷则主要体现在未能正确进行类型检查或类型推导; 代码生成缺陷则表现为编译器拒绝为语义正确的 TypeScript 程序生成相应的 JavaScript 程序, 或生成的 JavaScript 程序在语法和语义上存在错误.

图 2 给出了类型检查缺陷和代码生成缺陷的示例. 对于图 2(a) 中的代码片段, TypeScript 编译器未能正确进行类型检查. 在 TypeScript 中, Record<Key, T> 是一个标准类型, 它表示键的类型必须是 Key 类型的一个值, 且每个键对应的值的类型为 T. 然而, 图 2(a) 中的代码段在给 index4 和 index5 常量赋值时, 分别使用了“[k]:[]”和“[k]:0”这两个不满足 Record<Key, string> 要求的类型, 而 TypeScript 编译器没有报错, 表明其存在类型检查缺陷 (详见 https://github.com/microsoft/TypeScript/issues/38663). 图 2(b) 则展示了 TypeScript 在生成 JavaScript 代码时, 错误地将“const”关键字转换为“var”关键字, 表现为代码生成缺陷 (详见 https://github.com/microsoft/TypeScript/issues/59877).

```
type Key = 'a' | 'b';
// 按预期报错
const index1: Record<Key, string> = { a: '', b: 0 };
// 正确
const index2: Record<Key, string> = { a: '', b: '' };
const b: Key = 'b';
// 按预期报错
const index3: Record<Key, string> = { ...index2, [b]: 0 };
declare var k: Key;
// 未报错, 允许将任何内容赋值给字符串类型变量
const index4: Record<Key, string> = { ...index2, [k]: [] };
const index5: Record<Key, string> = { a: '', b: '', [k]: 0 };
```

(a) TypeScript bug #38663

```
// 输入的 TypeScript 代码
export const cilBlurLinear: string[] = [,]; // 定义一个空数组
const [,] = cilBlurLinear;                  // 尝试解构赋值
// 输出的 JavaScript 代码（错误翻译）
"use strict";
Object.defineProperty(exports, "__esModule", { value: true });
exports.cilBlurLinear = void 0;
exports.cilBlurLinear = [,];
var ; // 错误: `const` 被错误翻译成了 `var`, 并生成了无效语法
```

(b) TypeScript bug #59877

图 2 TypeScript 缺陷示例

TypeScript 编译器的缺陷可能会误导开发人员. 由于 TypeScript 编译器本身并不直接执行 TypeScript 代码, 而是将其编译为 JavaScript 代码后由 JavaScript 执行工具执行. 因此, 当生成的 JavaScript 代码存在缺陷时, 开发人员通常难以将这些缺陷与 TypeScript 编译器关联起来. 此外, TypeScript 编译器的缺陷还可能导致下游工具或框架的异常. 许多流行的前端框架和工具集成了 TypeScript 编译器, 如果 TypeScript 编译器存在缺陷, 这些框架和工具可能会出现难以追踪的错误. 例如, 图 2(a) 中展示的 TypeScript 的类型检查缺陷可能会导致下游工具 (如 React) 无法正确处理类型信息. 类似的, 当 TypeScript 编译器出现装饰器应用错误的缺陷时, 可能会导致下游框架在应用程序初始化时出现错误或功能失效, 尤其是依赖装饰器功能的框架, 如 Angular 和 NestJS.

基于上述原因, 提出有效的 TypeScript 编译器缺陷检测方法显得尤为重要和必要.

2 TscFuzz

本节介绍用于 TypeScript 编译器缺陷检测的工具 TscFuzz 的技术细节. 如图 3 所示, TscFuzz 主要由 3 部分组成: 基于大语言模型的语法程序数据集构建、基于语法和类型的程序变异和缺陷检测. TscFuzz 的基本思想是对 TypeScript 程序进行基于解析树的程序变异, 并利用差分测试来检测 TypeScript 编译器的缺陷.

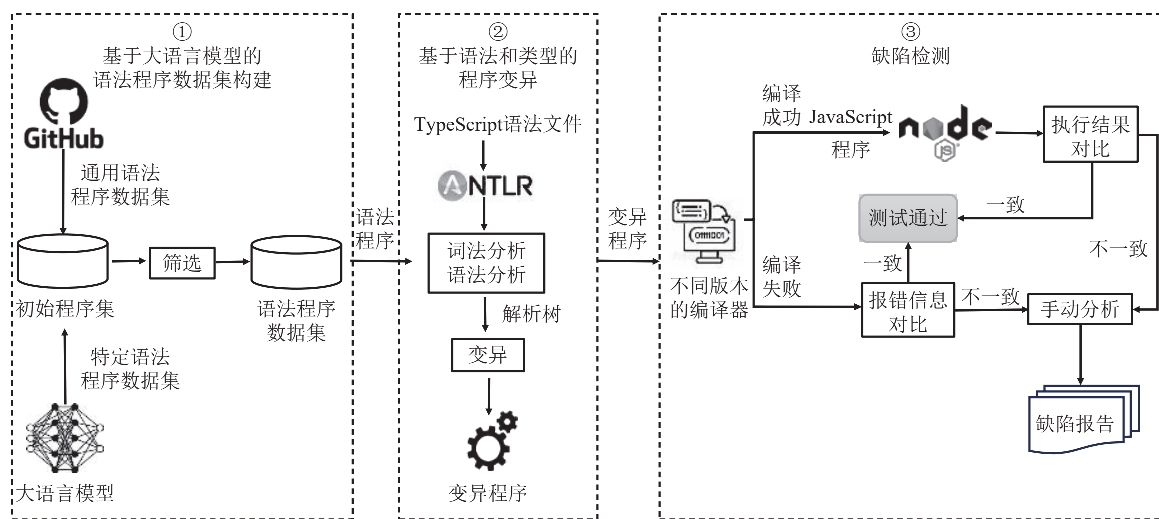


图 3 TscFuzz 方法框架

TscFuzz 首先借助 GitHub 开源社区和大语言模型共同构建一个包含通用语法特征和特定语法特征的语法程序数据集. 然后, 在程序变异阶段, TscFuzz 设计了一组包含通用变异算法和类型特定变异算法的算法集, 对语法程序数据集中的程序进行基于解析树的变异. 通过程序变异, 可以生成大量多样化的测试用例. 接下来, TscFuzz 将得到的变异程序输入不同版本的 TypeScript 编译器进行差分测试, 并将编译成功后输出的 JavaScript 程序输入 Node.js 进行执行. 通过分析编译和执行结果来判断 TypeScript 编译器是否存在缺陷. 此外, 本文还提出了一种代码复杂度的表示方法, 使用 6 个不同的指标衡量变异程序的代码复杂度和多样性. 为了不断丰富程序数据集的多样性, 我们将代码复杂度较高的变异程序加入语法程序数据集中, 从而增加测试用例的复杂性和覆盖面.

2.1 基于大语言模型的语法程序数据集构建

由于 TscFuzz 通过程序变异生成新的 TypeScript 程序用于 TypeScript 编译器的测试, 因此, 首先需要构建一个初始种子程序集, 即图 3 中的语法程序数据集. 本文从两个角度出发, 一方面, 为了针对 TypeScript 编译器中特定语法结构及相关特性进行专门测试, 利用大语言模型收集包含特定语法特征的程序, 以增强测试的针对性. 另一方面, 考虑大语言模型受生成机制和提示词导向的影响, 生成的程序在语法结构上通常较为相似, 难以涵盖丰富多样的语法特征. 因此, 为提高测试程序包含语法特征的全面性, 本文从开源平台收集包含通用语法特征的程序, 这

些程序来源于实际项目开发, 涵盖各类在真实场景中频繁使用的语法结构, 将它们纳入能显著提升测试程序所包含语法特征的全面性, 确保测试覆盖更广泛的语法应用场景. 这两种类型的程序相互补充, 共同构成语法程序数据集.

图 4 给出了语法程序数据集的构建流程. 具体来说, 为了收集包含通用语法特征的程序, 我们从 GitHub 上检索与“TypeScript”相关的开源仓库, 并按照受欢迎程度选择排名前 2000 的仓库作为语料库, 提取其中的 TypeScript 程序来构成通用语法程序数据集. 为了收集包含 TypeScript 特定语法特征的程序, 本文首先对比分析了 TypeScript 和 JavaScript 的语法, 确定了 TypeScript 中 11 种特有的语法类型, 包括“接口、枚举、元组、联合类型、交叉类型、类型别名、类型断言、类型注解、never 类型、可选类型和只读类型”. 这 11 种语法类型均为 TypeScript 常见类型中相较 JavaScript 新增的核心语法特性, 可以参考 TypeScript 官方文档 (https://www.typescriptlang.org/docs/handbook/2/everyday-types.html) 中对 TypeScript 常见类型的详细说明. 接着, 本文设计了一个待填充的提示词模板: “请生成一个不少于 m 行 (m>20) 的 TypeScript 程序, 程序中应包含<特有语法类型>, 并确保代码复杂且语义正确.” 通过将上述 11 种特定语法类型分别填充到提示词模板中, 生成了 11 个精确的自然语言提示词, 然后输入到大语言模型中执行多次, 从而得到包含特定语法类型的 TypeScript 程序, 构成特定语法程序数据集.

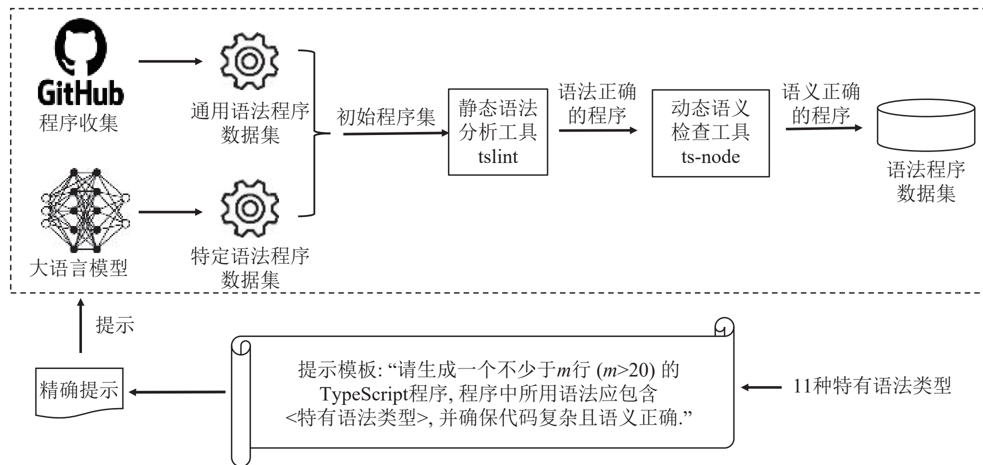


图 4 语法程序数据集构建流程

将通用语法程序数据集与特定语法程序数据集合并后, 得到初始程序集. 为了保证初始程序集中的种子程序的质量, 我们设计并实施了一个严格的两阶段过滤验证流程. 第 1 阶段采用静态语法验证方法. 我们将初始程序集中的原始代码片段, 逐一输入到业界广泛使用的静态分析工具 tslint 中. tslint 会严格检查代码是否遵循 TypeScript 的语法规则. 任何未能通过此语法检查的程序都会被立即丢弃. 这一步骤有效地过滤掉所有存在语法错误的代码, 确保了我们种子程序的语法正确性. 第 2 阶段采用动态语义验证方法. 所有通过了第 1 阶段静态检查的程序, 会被进一步输入动态执行工具 ts-node 中执行. ts-node 能够在 Node.js 环境中直接执行 TypeScript 代码. 如果一个程序在执行过程中抛出任何异常或导致运行时崩溃, 它同样被认为是语义不正确的, 并被立即丢弃. 这一步骤确保了种子程序不仅语法正确, 而且在逻辑上是可执行和语义完整的.

2.2 基于语法和类型的程序变异

2.2.1 程序变异流程

为了生成多样化的测试程序以检测 TypeScript 编译器的缺陷, 本文采用了基于语法和类型变异的算法对第 2.1 节中获得的语法程序数据集中的程序进行基于解析树的变异. 流程如图 5 所示. 具体来说, 首先, 从 GitHub 上的 antlr/grammars-v4 仓库下载 TypeScript 的语法文件, 包括词法规则文件 TypeScriptLexer.g4 和语法规则文件 TypeScriptParser.g4. 其次, 将这两个语法文件输入 ANTLR 4 工具中, 构建 TypeScript 语言对应的词法解析器 TypeScriptLexer 和语法解析器 TypeScriptParser. 然后, 对于语法程序数据集中的每个程序文件, 使用 ANTLR 4

工具将其解析为字符流, 并通过词法解析器 TypeScriptLexer 生成相应的 token 流. 接着, 将 token 流输入语法解析器 TypeScriptParser, 生成对应的解析树. 最后, 从本文设计的变异算法中选择一种变异规则对解析树进行变异, 并将变异后的解析树转换为 TypeScript 程序, 从而得到一系列变异程序. 图 6 给出了一个 TypeScript 代码片段“const a:number=1”对应的解析树.

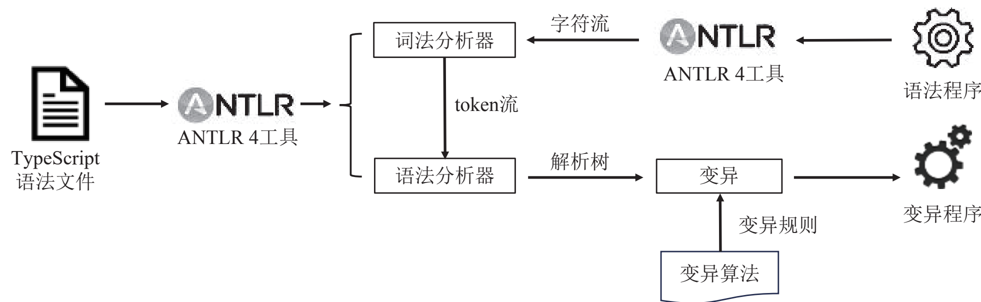


图 5 程序变异流程

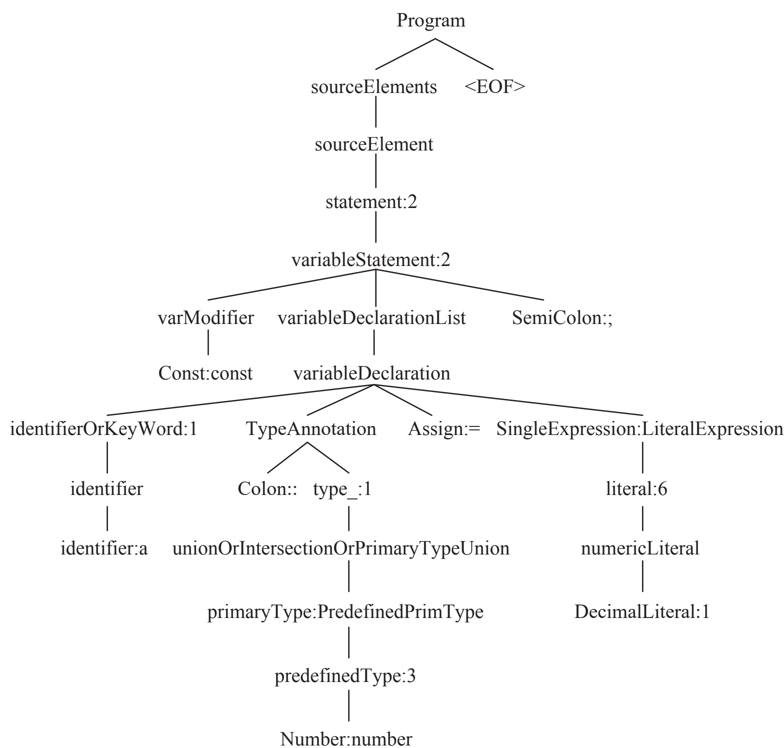


图 6 一个 TypeScript 代码片段的解析树

2.2.2 变异算子

为了提高变异程序的多样性并更好地测试 TypeScript 的类型系统, 我们设计了一组常规变异算法和一组类型特定变异算法, 具体内容如表 1 所示.

常规变异算法基于语法对程序进行变异, 聚焦于 TypeScript 语言的通用特性, 通过不同级别的变异操作, 确保生成的测试用例能够广泛覆盖各种语法和语义情景, 从而检测出编译器中可能存在的各类潜在缺陷. 主要包括以下 4 种变异级别.

(1) 子树级别变异. 子树在程序的语法结构中具有相对独立性和功能完整性, 其结构变化会对多个相关语法元

素产生影响. 通过对子树进行随机删除或替换操作, 能够显著改变程序的局部结构, 有效模拟不同模块之间的交互变化. 这有助于测试编译器在处理复杂语法结构时, 在函数参数传递、返回值处理等方面的稳定性和正确性, 可覆盖较大范围的语法规则, 极大地提高测试用例的多样性.

(2) 表达式级别变异. 表达式作为程序计算值的基本单元, 包含多种操作符和操作数, 其计算逻辑直接影响程序的执行结果. 对表达式进行变异, 如随机删除表达式、修改条件表达式内容或用 True/False 替换条件判断等, 能够直接改变程序的计算逻辑. 这可以全面测试编译器对运算符优先级、类型转换等计算逻辑的处理能力. 由于表达式在程序中广泛存在, 表达式级别变异能够广泛覆盖程序的各个部分, 增强测试的全面性.

(3) 语句级别变异. 语句是程序执行的基本单位, 不同类型语句的执行逻辑存在较大差异. 选择语句级别变异, 例如随机删除语句、随机删除函数体、进行相似 API 替换或随机重复一条语句等, 可从整体上改变程序的执行流程. 这有助于测试编译器对程序控制结构的处理能力, 包括条件判断、循环终止条件等关键方面. 通过模拟程序在不同执行场景下的情况, 能有效提高测试用例对程序执行逻辑的覆盖程度.

(4) 节点级别变异. 节点作为语法树的基本元素, 代表着单个语法成分. 选择节点级别变异, 如随机删除节点、替换 number/string 类型节点的值、布尔类型值取反或用同类型运算符/操作符替换原始符号等, 能够对程序的细微之处进行精准修改. 这可以深入检测编译器在词法和语法分析阶段对基本语法元素的识别和处理能力, 能够有效发现编译器处理基本语法元素时的潜在缺陷, 提升测试的精准度.

表 1 变异算子列表

类别	变异目标	变异规则	例子	
			变异前	变异后
常规变异算法	子树	随机删除或替换子树	let *obj*={*a*:1, *b*:2}	let *obj*={}
	表达式	随机删除表达式; 条件表达式的内容置空; 用True/False替换条件判断	if(*x*>10){*a*++;} if(*x*>10){*a*++;}	if(true){*a*++;} if(*x*>10){}
	语句	随机删除语句; 随机删除函数体; 相似API替换; 随机重复一条语句	str.*toUpperCase*() function *add*(*a*:number, *b*:number){return *a*+*b*;}	str.*toLowerCase*() function *add*(*a*:number, *b*:number){}
	叶子节点	随机删除节点; 替换number/string类型节点的值; 布尔类型值取反; 用同类型运算符/操作符替换原始符号	let *a*:number=1; const *flag*:boolean=true *a*>=*b*	let *a*:number=32; const *flag*:boolean=false *a*<=*b*
类型特定变异算法	对象	增加一条对象解构赋值语句; 子类对象赋值给父类变量	const *person*={name: “Alice”, age: 30};	const *person*={name: “Alice”, age: 30}; const {name, age} = *person*;
	数组	增加一条数组解构赋值语句	const *numbers*=[1,2,3,4];	const *numbers*=[1,2,3,4]; const [*a*, *b*, ...rest]= *numbers*;
	元组	随机删除一个或多个元素	*person*=[‘Alice’, 30]	*person*=[30]
	箭头函数	基本函数和箭头函数互相转换	function *fun*(*x*: number):*number* {return *x**2;}	const *fun*=(*x*: number):*number*=> *x**2
	联合类型	选取其他可选类型值替换原始变量	let *myVar*:number|string; *myVar*=10;	let *myVar*:number|string; *myVar*=“str”;
	类型注解	随机删除类型注解	const *a*:number=1;	const *a*=1;

注: *表示Python语法中的乘法操作

类型特定变异算法基于类型对程序进行变异, 此类算法集中于 TypeScript 的静态类型特性, 旨在挖掘 JavaScript 工具难以检测的类型相关缺陷. 通过对静态类型系统中的关键结构进行变异, 能够有效提升生成程序的复杂性, 更好地测试编译器的类型推断和兼容性处理等功能. 类型特定变异算法主要针对以下几种数据类型.

(1) 对象. 对象在 TypeScript 编程中应用广泛, 其结构和属性访问方式多样. 增加对象解构赋值语句或进行子类对象赋值给父类变量的变异操作, 能够测试编译器对对象结构解析、属性访问以及类型兼容性的处理能力.

(2) 数组. 数组在 TypeScript 中具有特定的类型规则. 增加数组解构赋值语句进行变异, 能够测试编译器对数组元素的提取、类型识别以及剩余元素处理等方面的能力. 不同长度和元素类型的数组在进行解构赋值时, 编译

器需要正确处理各种情况, 这种变异方式有助于发现潜在的类型处理缺陷.

(3) 元组. 元组是 TypeScript 中特有的类型, 它对元素数量和类型有明确规定. 随机删除元组中的一个或多个元素进行变异, 可测试编译器在处理元组元素变化时, 对类型一致性和元素数量限制的检查能力. 确保在元组结构发生变化时, 编译器能准确进行类型检查, 避免出现类型错误.

(4) 箭头函数. 箭头函数与基本函数在语法和作用域等方面存在差异. 在变异时进行基本函数和箭头函数互相转换, 能够测试编译器对不同函数定义方式的解析能力, 包括函数参数传递、返回值类型推断以及作用域处理等方面. 这有助于发现编译器在处理不同函数类型转换时可能出现的错误.

(5) 联合类型. 联合类型允许变量具有多种可能的类型, 增加了类型的灵活性和复杂性. 选取其他可选类型值替换原始变量进行变异, 能够测试编译器对联合类型变量在不同类型赋值情况下的类型检查能力, 确保编译器在处理联合类型时能正确判断变量类型, 避免类型错误的发生.

(6) 类型注解. 类型注解在 TypeScript 中用于明确变量或函数的类型, 对类型检查和代码可读性至关重要. 随机删除类型注解进行变异, 可以测试编译器在缺少明确类型注解时的类型推断能力, 检查编译器能否根据上下文正确推断变量类型, 以及在类型推断过程中是否存在错误.

综上所述, 常规变异算法具有较高的普适性, 能够快速生成大量覆盖面广的测试用例; 而类型特定变异算法则针对性更强, 能够提高编译器测试的全面性和针对性, 二者结合能够为 TypeScript 编译器缺陷检测提供语法和语义信息丰富且复杂度高的测试用例.

2.2.3 基于解析树的程序变异算法

算法 1 展示了基于解析树的程序变异算法的流程. 在程序变异之前, 我们首先从第 2.1 节构建的语法程序数据集中随机选取若干程序, 通过遍历其解析树收集字符串数据集 *Strs*、数字数据集 *Nums* 和子树数据集 *SubTrees*, 这些数据集将为表 1 中的部分变异算法提供替换值或替换子树. 算法 1 将这 3 个集合作为输入, 并接收一个待变异的 TypeScript 程序 *prog*、一个最大迭代次数 *max_iterations*, 以及表 1 所列的变异算子总数 *mutation_count*. 算法的输出是 *prog* 变异后的程序 *mutated_prog*. 在算法开始时, 我们首先初始化变异程序 *mutated_prog* 和当前迭代次数 *iteration_count* (第 1, 2 行). 接着, 我们尝试将待变异程序 *prog* 解析为解析树, 若解析失败则退出; 若解析成功, 则得到解析树 *prog_tree* (第 3–6 行). 随后, 我们使用 *getRandomNumber* 函数生成一个介于 0 和 *mutation_count* 之间的随机数 *randomNum*, 并在 *mutate* 函数中用第 *randomNum*+1 个变异算法对解析树 *prog_tree* 进行变异, 变异后的解析树 *mutated_tree* 通过 *transform* 函数转换为变异程序 *mutated_prog* (第 8–10 行). 接下来, 我们对比变异前后的程序 *prog* 和 *mutated_prog*, 若两个程序完全相同 (*prog* 中没有当前变异规则的目标节点), 则重复第 7–11 行的步骤, 重新选择变异算法对 *prog* 进行变异. 若在达到最大迭代次数 *max_iterations* 后, 仍然未能得到一个与 *prog* 不同的变异程序, 则停止变异这个程序 (第 12–16 行). 当获取到变异程序 *mutated_prog* 后, 分别计算原始程序 *prog* 和变异后程序 *mutated_prog* 的代码复杂度, 并进行比较. 若变异程序的代码复杂度较高, 则将其加入语法程序数据集中用于丰富数据集的多样性 (第 18–22 行). 代码复杂度的计算方法在第 2.2.4 节中给出具体介绍.

算法 1. 基于解析树的程序变异算法.

输入: 待变异程序 *prog* 字符串数据集 *Strs*, 数字数据集 *Nums*, 子树数据集 *SubTrees*, 最大迭代次数 *max_iterations*, 变异算子总数 *mutation_count*;
输出: 变异程序 *mutated_prog*.

1. *mutated_prog* = null
2. *iteration_count* = 0
3. parse *prog* into a parse tree *prog_tree*
4. **if** there are any parsing errors **then** // 解析失败则退出对当前程序的处理
5.   **return**

```
6. end if
7. while iteration_count < max_iterations do // 在最大迭代次数内对 prog 进行变异
8.     randomNum = getRandomNumber(0, mutation_count)
9.     mutated_tree = mutate(prog_tree, randomNum)
10.    mutated_prog = transform(mutated_tree)
11.    interation_count ++
12.    if mutated_prog is identical to prog then // prog 中没有当前选取变异规则的目标节点
13.       continue
14.    else
15.       return mutated_prog
16.    end if
17. end while
18. prog_complexity = getCodeComplex(prog, prog_tree)
19. mutated_complexity = getCodeComplex(mutated_prog, mutated_tree)
20. if mutated_complexity > prog_complexity then // 对比变异前后程序的复杂度
21.    copy mutated_prog to syntax program dataset
22. end if
23. return mutated_prog
```

2.2.4 代码复杂度

代码复杂度是衡量程序结构和计算逻辑复杂性的重要指标, 通常用于评估代码的可读性、复杂性和多样性. 已有研究提出了多种用于量化代码复杂度的度量方法, 包括代码行数、圈复杂度、Halstead 复杂度、语法树的结构等[40−44]. 这些指标从不同角度刻画程序的复杂性, 例如 LOC 反映代码规模[41], 圈复杂度衡量程序的控制流复杂度[40−42], Halstead 复杂度关注运算符和操作数的分布[41−43], 而语法树的结构则刻画了程序的层次和语法特性[44]. 结合多个复杂度指标能够提供更全面的评估, 帮助分析代码的语法和结构方面的多样性.

为了不断丰富语法程序数据集的多样性, 我们将代码复杂度较高的变异程序添加到语法程序数据集中, 从而增加测试用例的复杂性和多样性. 为此, 我们设计了一个综合代码复杂度的度量公式, 将 6 个指标融合起来表示代码复杂度. 具体指标和计算方法如下.

(1) 代码行数: 使用 cloc 工具计算程序的代码行数.

(2) 圈复杂度: 圈复杂度定义为判定节点数加 1. 判定节点数指程序中所有控制结构的判定节点总数, 包括 if 语句、while 语句、case 语句、三元运算、with 语句、for 语句、for…in 语句以及 for…of 语句中的所有判定节点. 通过遍历解析树可以获取程序的圈复杂度.

(3) 解析树最大深度: 指解析树的最大递归深度. 具体而言, 深度 *Depth* 指从解析树根节点到叶节点的最长路径的长度. 最大深度 *maxDepth* 通过递归遍历解析树计算得到, 具体如公式 (1) 所示, 其中, *child* 为当前解析树节点的子节点, *currentDepth* 为当前深度, *calculateDepth* 为计算解析树深度的递归函数:

$$maxDepth = \max(maxDepth, calculateDepth(child, currentDepth + 1)) \tag{1}$$

(4) 最大嵌套深度: 最大嵌套深度是指所有控制语句 (如 if、while、switch、with、for、for…in、for…of 语句以及三元运算) 的嵌套深度中的最大值. 通过遍历这些语句对应的子树, 可以得到该值.

(5) Halstead 复杂度: Halstead 复杂度根据程序中的操作符和操作数的数量计算程序复杂度, 即以下 4 个指标: n_1 (不同操作符的数量)、n_2 (不同操作数的数量)、N_1 (操作符的总出现次数)、N_2 (操作数的总出现次数), 这 4 个值可以通过遍历程序解析树来统计. 我们用公式 (2) 所示的计算方式来综合表示 Halstead 复杂度, 其中 $n = n_1 + n_2$, $N = N_1 + N_2$, E 为编程工作量, 其计算方式如公式 (3) 所示:

$$Halstead_score = 0.3n + 0.4N + 0.5E \tag{2}$$

$$E = \frac{N\log_2 n}{\frac{2}{n_1} \times \frac{n_2}{N_2}} \tag{3}$$

(6) 类和方法的数量: 此指标计算解析树中所有类和函数的数量之和, 通过遍历解析树可以得到.

我们将这 6 个指标对应的数值组成一个列表 (ArrayList 类型). 对于所有变异程序, 我们计算其和原始程序的复杂度. 在比较时, 若 6 个指标中, 指标值有所提升的指标超过 1 个, 我们便认为变异程序的复杂度提高, 从而将其加入语法程序数据集中. 通过这种方式, 我们能更清晰地度量变异程序的复杂度并持续丰富数据集的多样性, 确保生成的变异程序具有更高的复杂性和更广泛的测试覆盖面.

2.3 缺陷检测

由于 TypeScript 编译器的输出是 JavaScript 程序, 因此我们采用差分测试方法来检测 TypeScript 编译器的潜在缺陷. 为了实现对 TypeScript 编译器的全面测试, 不仅需要观察编译过程中的行为, 还必须验证编译器输出的 JavaScript 程序在 Node.js 中的执行情况, 以确保其语义上的正确性.

如第 1 节所述, TypeScript 编译器的缺陷根据其工作流程主要分为 3 类: 语法分析缺陷、类型检查缺陷和代码生成缺陷. 图 7 所示的差分测试流程及可能结果示意图中, 详细描述了如何检测这 3 类缺陷.

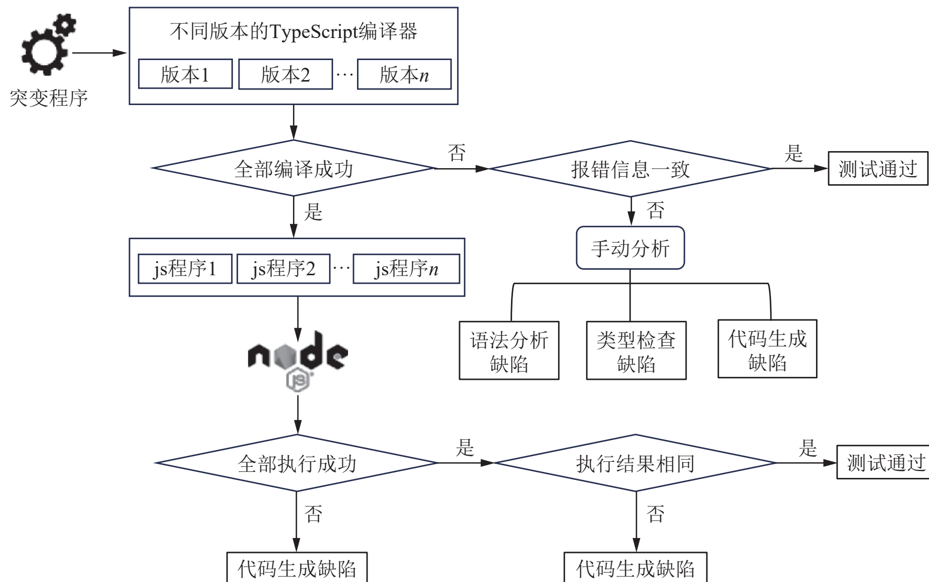


图 7 差分测试流程及可能结果

针对第 2.2 节中生成的变异程序, 我们首先使用多个不同版本的 TypeScript 编译器对其进行编译, 并对比编译结果. 如果所有版本的编译器均能成功编译, 则将编译生成的 JavaScript 程序分别输入 Node.js 中执行, 并对比执行结果. 在编译和执行过程中, 可能产生 4 种不同的结果: 第 1 种结果是测试通过, 当不同版本编译器在编译阶段均失败并产生相同的报错信息, 或在执行阶段所有版本生成的 JavaScript 程序都能成功执行且结果一致时, 则认为该变异程序通过了测试. 其余 3 种结果分别对应 3 种类型的潜在缺陷. 当不同版本的编译器不能全部编译成功或全部编译失败, 且报错信息不一致时, 便对报错信息进行手动分析, 若报错信息是由编译器错误地拒绝语法正确的程序导致的, 或编译器输出了错误的语法分析信息, 则认为这是一个潜在的语法分析缺陷; 若报错信息包含错误的类型诊断信息, 则认为这是一个潜在的类型检查缺陷; 若报错信息表明编译器拒绝为一个语义正确的程序生成对应的 JavaScript 程序, 则认为这是一个潜在的代码生成缺陷. 当所有版本的编译器均能成功编译变异程序, 但

生成的 JavaScript 程序无法正常执行或执行结果不同时, 则认为编译器存在潜在的代码生成缺陷.

本文将所有可能表明编译器存在缺陷的变异程序及其对应的报错信息保存下来, 并进行人工分析. 由于人工判别存在主观性风险, 所以对于检测到的潜在缺陷进行区分处理. 对于崩溃型缺陷, 此类缺陷是客观的, 直接根据编译器崩溃时产生的栈跟踪信息进行判断, 不存在判别错误的情况. 对于非崩溃型缺陷 (即语法分析缺陷、类型检查缺陷和代码生成缺陷), 对于这类缺陷, 建立了一个交叉验证机制. 一个潜在缺陷需要由 3 位不同的作者独立分析并达成一致结论后, 才会被确认为是真实缺陷并着手报告. 在分析过程中, 我们会确保报错信息不是由程序中的随机行为 (如随机数) 导致的, 以此过滤伪缺陷. 我们向 TypeScript 社区提交的缺陷报告具有很高的接受率, 这证明了本文的人工分析流程是严谨可靠的. 最终, 缺陷检测阶段的输出将是一系列识别出的缺陷及其对应的变异程序.

3 实验分析

3.1 实验数据

为了构建 TypeScript 的语法程序数据集, 本文收集了 2000 个来自 GitHub 的存储库, 其中包含 498153 个 TypeScript 程序, 构成通用语法程序数据集. 同时, 通过第 2.1 节中设计的提示词, 使用大语言模型 GPT-3.5 生成了 20000 个包含特定语法特征的程序, 构成类型特定语法程序数据集. 之所以选用 GPT-3.5, 主要基于两方面的实际考量: 其一, 在实验启动阶段, GPT-3.5 是代码生成任务中表现突出的主流模型, 且获取与使用均较为便捷; 其二, 从生成效率与 API 调用成本的综合权衡来看, 它能够支撑大规模生成种子程序的需求, 为实验提供切实可行的技术路径. 具体生成流程见第 2.1 节. 上述两个数据集合并后得到一个包含 518153 个程序的初始程序集. 该程序集经过语法和语义检查, 最终获得一个包含 38826 个 TypeScript 程序的语法程序数据集, 作为原始种子程序用于后续变异. 此外, 为提供变异过程中需要的字符串、数字和子树数据集, 本文从语法程序数据集中选取 1000 个程序, 遍历其解析树, 收集了包含 36364 个字符串、21476 个数字和 39789 个子树的集合. 这些集合为变异过程提供了丰富的替代值和结构.

3.2 实验环境设置

本文设计并实现的 TypeScript 编译器自动化测试框架 TscFuzz 共计 9300 行 Java 代码, 基于 Java 20.0.1 版本开发. 在 TypeScript 程序和解析树的转化过程中, 使用了第三方工具 ANTLR 4, 并基于该工具生成的解析树实现了第 2.2 节中的变异算法. 为了进行差分测试, 参考 RustSmith[45]的实现, 我们使用了 3 个版本的 TypeScript 编译器: v5.4.5、v5.5.4 和 v5.6 Beta, 它们分别代表了一个稳定的旧版本、一个最新的正式发行版和一个前沿的测试版. 这种组合策略能够扩大缺陷发现的覆盖面: 1) 对比稳定版和最新版, 可以有效发现回归缺陷 (即旧版本中已修复但在新版本中复现的缺陷); 2) 对比最新版和测试版, 可以提前发现新增缺陷与未上线特性相关的预览缺陷. 同时, 为执行生成的 JavaScript 程序, 本文使用了 Node.js 20.14.0 版本, 并利用 tslint (v6.1.3) 和 ts-node (v10.9.2) 进行语法检查和语义检查. 所有工具均采用了官方默认配置.

所有实验均在一台高性能的 Linux 服务器上进行. 该服务器配备了 144 核 288 线程的英特尔至强铂金处理器 (Intel(R) Xeon(R) Platinum 8374C CPU @ 2.70 GHz) 和 256 GB 的内存空间, 实验操作系统为 Ubuntu 22.04.3 LTS (GNU/Linux 6.2.0-37-generic x86_64).

3.3 实验结果与分析

为了评估基于变异和差分测试的 TypeScript 编译器缺陷检测方法 TscFuzz 的有效性, 本文参考了 JavaScript 引擎 (如 V8、SpiderMonkey) 测试[22,23]和编译器 (如 GCC、LLVM) 测试[46,47]领域的主流方法, 聚焦于在真实世界的软件中发现未知缺陷的能力, 并以此作为衡量工具实用价值的关键指标. 我们研究了以下 4 个问题.

- RQ1: TscFuzz 能否有效发现真实世界中 TypeScript 编译器的缺陷?
- RQ2: TscFuzz 与 JavaScript 引擎测试方法相比, 能否更有效地检测 TypeScript 编译器的缺陷?
- RQ3: TscFuzz 的关键组件是否有利于 TypeScript 编译器的缺陷检测?

• RQ4: TscFuzz 能否生成具有较高复杂度和多样性的测试程序?

RQ1: TscFuzz 能否有效发现真实世界中 TypeScript 编译器的缺陷?

为了评估 TscFuzz 发现真实世界中 TypeScript 编译器缺陷的能力, 本文从 2024 年 6–9 月进行了 3 个月的回归测试实验, 每轮回归测试过程的周期约为 3 天. 测试期间, 主要对 TypeScript 编译器的最新发行版本进行测试. 由于 TypeScript 编译器开发版本更新频繁, 本文在测试过程中会根据版本更新情况及时更新测试版本, 以确保测试的时效性和有效性.

在缺陷发现后, 本文将 TscFuzz 发现的缺陷提交给开发人员进行确认. 对于每个缺陷, 首先根据缺陷信息中的关键字搜索实验期间已发现的缺陷, 并与 TypeScript 编译器的缺陷库中的已知缺陷进行对比. 为确保缺陷的独特性, 本文手动分析并剔除了重复缺陷. 同时, 针对每个缺陷, 手动缩减了触发缺陷的 TypeScript 程序, 以便开发人员快速定位并确认缺陷.

如表 2 所示, 经过 3 个月的回归测试, TscFuzz 共发现 12 个缺陷, 其中 8 个已被开发者确认, 7 个已被修复, 4 个为重复或无效缺陷. 按照缺陷类型, 可以分为语法分析缺陷 (7 个)、类型检查缺陷 (3 个) 和代码生成缺陷 (2 个). 所有缺陷均已确认, 其中, 除代码生成缺陷中有一个尚未修复, 其余缺陷均已修复.

表 2 TscFuzz 在 TypeScript 编译器最新版本上发现的缺陷

类别	提交	确认	修复	重复或无效
语法分析缺陷	7	3	3	4
类型检查缺陷	3	3	3	0
代码生成缺陷	2	2	1	0
合计	12	8	7	4

通过对发现的缺陷进行分析, TscFuzz 发现的语法分析缺陷和类型检查缺陷的数量明显高于代码生成缺陷的数量, 可能的原因主要有以下两个方面: 首先, TypeScript 编译器的语法分析和类型检查功能的复杂度较高, 而代码生成功能相对较为简单. TypeScript 编译器拥有复杂的类型系统, 语法分析和类型检查功能需要处理大量复杂的语法结构和类型推导, 尤其是在处理高级类型时, 编译器的错误处理更容易受到测试程序的影响, 从而引发更多的缺陷. 相比之下, 代码生成功能的主要任务是将解析后的抽象语法树转换为 JavaScript 代码, 相对于语法分析和类型检查功能而言, 其复杂度较低. 因此, 代码生成缺陷的数量较少. 另一方面, 本文设计的两个关键组件——基于大语言模型的特定语法程序数据集构建和类型特定变异算法——使得变异程序集中包含了高复杂度的代码, 这些程序更容易包含复杂的语法结构和类型推导路径, 更容易触发语法分析和类型检查相关的缺陷. 因此语法分析缺陷和类型检查缺陷的数量更多.

接下来, 我们介绍 TscFuzz 发现的缺陷的两个示例.

(1) 语法分析缺陷: 此类缺陷的一个示例是 TypeScript 的 59374 号缺陷 (详见 https://github.com/microsoft/TypeScript/issues/59374), 该缺陷已被开发人员修复. 图 8 显示了该缺陷的代码片段. 该代码片段的第 2 行在语法分析阶段会导致 TypeScript 编译器崩溃. 从图 8 的代码可以看出, 该代码片段混用了标签语法 (null subTitle) 与导出语法 (export 语句). 在模块上下文中, 这种语法是非法的, 违反了 TypeScript 的语法规则. 按照正常的编译器行为, 编译器应该检测到语法错误并输出相应的错误信息, 而不是崩溃. 该缺陷的根本原因是 TypeScript 编译器未能正确处理模块中顶层的标签语句, 在解析和转换模块的过程中, 没有及时识别无效的标签声明或语法错误, 从而未能进行适当的错误处理, 最终导致了编译器崩溃.

```
export const box: string
null subTitle:
export const title: string
```

图 8 TypeScript 编译器语法分析失败导致崩溃的代码

(2) 代码生成缺陷: 此类缺陷的一个示例是 TypeScript 的 59877 号缺陷, 该缺陷已被开发人员修复. 图 2(b) 显示了该缺陷的代码片段. 在代码片段中, TypeScript 代码被错误地翻译为图 2(b) 中的 JavaScript 代码. 从生成的代

码可以看出, 编译器错误地将“const”关键字转换为了“var”关键字, 并生成了非法的 JavaScript 代码“var;”. 该缺陷的根本原因在于, 编译器在处理解构赋值模式 (destructuring patterns) 时, 对于空列表 (如 [,]) 的处理逻辑存在问题. 当声明列表被转换为空时, 编译器未能正确丢弃变量声明语句, 从而生成了非法的 JavaScript 代码.

回答 RQ1: TscFuzz 发现了 12 个真实世界的 TypeScript 编译器缺陷, 其中 8 个已被开发人员确认, 7 个已被修复. 实验结果表明, TscFuzz 可以有效地发现真实世界编译器中的缺陷.

RQ2: TscFuzz 与 JavaScript 引擎测试方法相比, 能否更有效地检测 TypeScript 编译器的缺陷?

本节将 TscFuzz 发现的缺陷数量与两种最先进的 JavaScript 程序生成方法, DIE 和 FuzzJIT, 进行了比较. DIE 和 FuzzJIT 是当前测试 JavaScript 引擎的主流方法. DIE 是一个基于 AFL 的变异模糊器, 在抽象语法树级别对程序进行变异, 采用了一种“结构保留”的变异算法, 在变异过程中随机保留原始种子输入的有效结构和类型信息, 从而生成语法有效且功能丰富的测试程序. FuzzJIT 基于 Fuzzilli 实现, 通过改变程序的控制流和数据流实现对 JavaScript 程序的变异, 专注于揭示 JIT 编译器相关的缺陷.

实验在 TypeScript 5.4 版本上进行的, 这样方便找到大量的静态缺陷进行比较. 由于 TypeScript 编译器的更新频繁, 旧版本的编译器进行测试通常包含较多缺陷, 而使用最新版本的编译器运行测试用例可以帮助我们快速确认缺陷是否已被修复. 如果某个缺陷在最新版本中不存在, 则认为该缺陷已经被修复; 否则, 本文会将其报告给开发人员以供确认.

本文在实验过程中执行每个方法 72 h, 并重复 5 次取平均值. 为了保证公平性, 本实验为所有算法提供了相同的种子数据集. 由于 TscFuzz 对 TypeScript 程序进行变异, 而 DIE 和 FuzzJIT 针对的是 JavaScript 程序, 因此本文通过 TypeScript 编译器将第 2.1 节中构建的 TypeScript 语法程序数据集转换为 JavaScript 程序, 作为 DIE 和 FuzzJIT 的初始种子集. 在 DIE 和 FuzzJIT 对 JavaScript 程序进行变异后, 我们将生成的 JavaScript 程序的后缀更改为“.ts”, 并使用 tslint 和 ts-node 筛选出语法有效的 TypeScript 程序, 然后将这些程序作为测试输入送入第 2.3 节中的缺陷检测模块来统一查找缺陷.

图 9 显示了 TscFuzz 和基线算法发现的缺陷数量. TscFuzz 在测试期间发现的缺陷显著高于 DIE 和 FuzzJIT. 在测试中, TscFuzz、DIE 和 FuzzJIT 分别发现了 5 个、3 个和 2 个缺陷, TscFuzz 发现的缺陷覆盖了另外两个工具发现的缺陷, 表明 TscFuzz 在缺陷发现能力上具有明显优势.

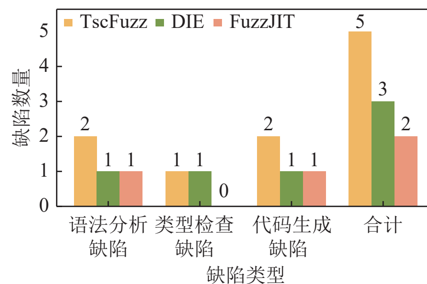


图 9　TscFuzz 和基线方法在 72 h 发现的缺陷数量

为了进一步验证 TscFuzz 的有效性, 我们使用假设检验的方法来评估 TscFuzz、DIE 和 FuzzJIT 这 3 个不同工具在缺陷发现能力上的差异. 假设检验是一种统计学方法, 用于根据样本数据推断总体的特征. 其基本过程包括提出一个关于总体参数的假设 (通常是零假设), 然后通过样本数据进行检验, 从而判断是否拒绝零假设. 本研究中采用了 Wilcoxon 秩和检验, 这是一种常用于比较两组独立样本分布是否存在差异的非参数检验方法. 将 TscFuzz 分别与 DIE、FuzzJIT 进行两两比较, 判断它们发现的缺陷总数是否存在显著差异. 我们定义零假设 (H_0) 为: TscFuzz 与 DIE、FuzzJIT 在缺陷发现能力上的差异不显著, 即三者发现的缺陷总数无明显差异. 在进行 Wilcoxon 秩和检验时, 通过计算检验统计量并得到 P 值, 用以衡量观察结果与零假设一致的程度. P 值表示在零假设为真的前提下, 检验统计量大于或等于实际观察值的概率. 较小的 P 值表明原假设的成立概率较低, 进一步支持拒绝原假

设的理由, 通常, P 值小于 0.01 被认为是非常显著的结果, 表明差异极为显著; 如果 P 值介于 0.01 和 0.05 之间, 则表明样本间差异显著; 而 P 值大于 0.05 则表明结果更倾向于接受原假设. 根据对 TscFuzz、DIE 和 FuzzJIT 的 P 值分析, TscFuzz 相对于 DIE 和 FuzzJIT 的 P 值分别为 0.012 和 0.009, 均远小于 0.05, 显示出 TscFuzz 在缺陷发现能力上具有显著优势.

经过对实验数据的分析, 可以看出两个基线算法在发现不同类型缺陷的能力上存在差异 (如图 9). DIE 和 FuzzJIT 分别发现了 1 个和 0 个类型检查缺陷, 并且各自发现了 1 个语法分析缺陷和 1 个代码生成缺陷.

对于 DIE 和 FuzzJIT, 由于它们的变异策略主要关注 JavaScript 程序的语法特点和执行流程, 而不涉及 TypeScript 的静态类型, 这导致它们较难生成针对 TypeScript 编译器的有效测试用例, 因此发现的缺陷相比于 TscFuzz 更少. 而 TscFuzz 的变异策略包含一组类型特定的变异算法, 能够生成语法和类型信息更加丰富的测试程序, 这使得 TscFuzz 在发现缺陷的能力上超越了 DIE 和 FuzzJIT. 通过生成包含多种复杂类型结构和语法特征的测试程序, TscFuzz 更容易触发 TypeScript 编译器中潜在的缺陷, 尤其是在语法分析缺陷和类型检查缺陷.

另外, 对比两个基线方法可以得出, 在发现的缺陷类型上, DIE 和 FuzzJIT 发现的语法分析缺陷与类型检查缺陷的数量之和均不低于代码生成缺陷的数量, 这和第 4.2.1 节中提到的原因一致, 即 TypeScript 编译器的语法分析和类型检查功能较为复杂, 更容易出现缺陷. 而在 DIE 和 FuzzJIT 之间, DIE 发现类型检查缺陷数量高于 FuzzJIT, 这可能是由于 DIE 的变异算法在生成测试程序时采用了结构保留和类型保留的策略, 这使得生成的测试程序更容易包含丰富的语法和类型信息, 也就更容易触发类型检查缺陷. 相比之下, FuzzJIT 的变异策略更侧重于控制流和数据流的修改, 主要针对 JavaScript 引擎的 JIT 编译器, 这使得其生成的测试程序较少触发 TypeScript 编译器的类型检查缺陷.

为了进一步分析 TscFuzz 与基线方法所发现的缺陷之间的关系, 我们针对图 9 所示的缺陷绘制了对应的韦恩图, 如图 10 所示, 以直观展示 TscFuzz、DIE 和 FuzzJIT 发现的缺陷数量及其分布关系. 图中蓝色圆圈代表 TscFuzz, 绿色圆圈代表 FuzzJIT, 红色圆圈代表 DIE. 韦恩图可以用来表示多个集合之间的关系及其交集情况. 在图 10 所示的韦恩图中可以看出, TscFuzz 发现的缺陷覆盖了其他两种工具发现的缺陷, 说明其缺陷发现能力更为全面. DIE 和 FuzzJIT 发现的缺陷中存在一个重合的缺陷. 实验结果表明, TscFuzz 的覆盖范围最广, 能够发现 DIE 和 FuzzJIT 发现的所有缺陷, 显著体现了其在缺陷发现能力上的优势. 此外, DIE 和 FuzzJIT 发现的缺陷有部分重叠, 独有缺陷较少, 这进一步验证了 TscFuzz 的全面性和有效性.

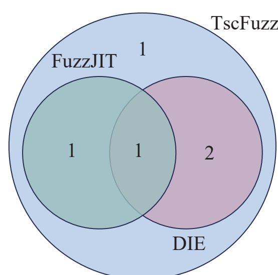


图 10　TscFuzz、DIE 和 FuzzJIT 发现的缺陷数量分布

回答 RQ2: TscFuzz 在 72 h 的测试中共发现了 5 个缺陷, 比基线方法 DIE 和 FuzzJIT 分别多检测了 2 个和 3 个 bug. 这一结果表明, TscFuzz 在 TypeScript 编译器缺陷检测方面具有明显的优势, 相比于 JavaScript 引擎测试工具, TscFuzz 能够更有效地发现 TypeScript 编译器中的潜在缺陷.

RQ3: TscFuzz 的关键组件是否有利于 TypeScript 编译器的缺陷检测?

基于大语言模型的特定语法程序数据集构建和类型特定变异算法是 TscFuzz 的核心组成部分. 通过这两个关键组件, TscFuzz 能够构建一组包含丰富类型信息的程序数据集, 并对程序中的特定类型进行变异, 从而提升缺陷检测能力. 为评估这两个组件的影响, 本文采用的控制变量法涉及 TscFuzz 的 3 个变体, 即 TscFuzz-LLM、TscFuzz-TM

和 TscFuzz-LLM-TM. 其中, TscFuzz-LLM 移除了基于大语言模型生成的特定语法程序数据集, 仅使用从 GitHub 收集的程序构建语法程序数据集; TscFuzz-TM 则移除了类型特定变异算法, 仅采用表 1 中的常规变异算法对语法程序数据集中的程序进行变异; TscFuzz-LLM-TM 同时移除了基于大语言模型生成的特定语法程序数据集和类型特定变异算法, 仅保留通用语法程序数据集和常规变异算法.

本文使用 TscFuzz 及其 3 个变体测试 TypeScript 5.4 版本编译器. 为了保证实验的准确性, 每种方法均重复 5 次实验, 每次实验持续 72 h, 并统计了平均每次实验发现的缺陷数量. 实验结果如表 3 所示, 完整的 TscFuzz 在 72 h 内平均发现了 5 个缺陷, 显著优于其他变体方法; TscFuzz-LLM 和 TscFuzz-TM 分别发现了 3 个和 2 个缺陷, 而 TscFuzz-LLM-TM 的缺陷数量仅为 1 个. 表 3 的第 2 行给出了 3 个变体对应的 P 值, 分别为 0.012、0.009 和 0.009, 均远小于 0.05, 说明了 TscFuzz 在缺陷发现能力上具有显著优势.

表 3　TscFuzz 及其变体发现的缺陷数量

指标	TscFuzz	TscFuzz-LLM	TscFuzz-TM	TscFuzz-LLM-TM
缺陷数量	5	3	2	1
P值	—	0.012	0.009	0.009

实验结果表明, TscFuzz 在缺陷检测能力上显著优于其 3 个变体. 具体而言, TscFuzz 在测试期间共发现了 5 个缺陷, 说明大语言模型生成的特定语法程序数据集和类型特定变异算法这两个关键组件的结合, 能够更好地提高生成测试程序的语法多样性和类型复杂度, 在缺陷检测能力上显著优于使用单一组件的方法.

相比之下, TscFuzz-LLM 发现了 3 个缺陷, 说明基于大语言模型的语法程序数据集构建能够显著扩展测试程序的语法和类型覆盖范围, 移除该组件会降低 TscFuzz 的缺陷检测的有效性. TscFuzz-TM 发现了 2 个缺陷, 表明类型特定变异算法的引入进一步增强了测试程序的复杂性和类型相关性, 使编译器在类型检查阶段更容易暴露潜在缺陷, 而移除该组件会使得生成的测试程序语法和类型的多样性降低, 从而限制 TscFuzz 缺陷检测的全面性. 而 TscFuzz-LLM-TM 仅发现了 1 个缺陷, 说明了单独使用常规变异算法对通用语法程序数据集进行变异时, 生成的程序普遍缺乏复杂的语法结构和类型信息. 这使得生成的测试程序较为简单, 无法有效触及编译器在复杂语法和类型分析中的潜在缺陷, 因此其缺陷发现能力相对较弱.

在 3 个变体之间, TscFuzz-LLM 发现的缺陷数量高于 TscFuzz-TM, 这表明, 相较于大语言模型生成的特定类型程序, 类型特定变异算法能够更有效地提升测试程序的语法和类型复杂度. 而 TscFuzz-LLM-TM 发现的缺陷数量低于另两个变体方法, 进一步说明了基于大语言模型的特定语法程序数据集与类型特定变异算法的组合效果明显优于单独使用任一组件. 这一结果表明, 两个组件的融合相互增强了各自的缺陷检测能力, 产生了协同效应.

回答 RQ3: TscFuzz 和 3 个变体在 72 h 的测试中分别发现了 5 个、3 个、2 个和 1 个缺陷, TscFuzz 发现的缺陷数量显著高于 3 个变体, 说明了本文提出的基于大语言模型的特定语法程序数据集构建和类型特定变异算法这两个组件的有效性.

RQ4: TscFuzz 能否生成具有较高复杂度和多样性的测试程序?

为了探究 TscFuzz 生成的测试程序是否具有较高的复杂度和多样性, 我们从代码复杂度和类型多样性两个方面进行分析.

• 代码复杂度分析: 在实验过程中, 使用第 2.2.4 节设计的代码复杂度度量方式, 对 TscFuzz 变异生成的测试程序进行复杂度评估. 具体来说, 我们从语法程序数据集中随机选取 1 000 个程序, 并使用 TscFuzz 对其进行变异. 计算变异后程序的代码行数、圈复杂度、解析树最大深度、最大嵌套深度、Halstead 复杂度以及类和方法的数量这 6 个指标. 为保证实验结果的准确性和可靠性, 变异程序的复杂度是通过 5 次重复实验取平均值得到的. 表 4 给出了原始程序和变异程序的平均复杂度.

从表 4 数据来看, 各项指标均显示 TscFuzz 变异后的程序复杂度有明显提升. 代码行数从 13.23 增至 36.68, 表明程序规模和逻辑丰富度大幅增加; 圈复杂度从 2.35 提升至 2.53, 意味着程序逻辑分支增多、执行路径更复杂; 解析树最大深度由 16.08 增长至 20.20, 以及最大嵌套深度从 0.38 提升至 0.55, 都说明程序的语法结构和嵌套关系

变得更加复杂; Halstead 复杂度从 3541.53 提升至 3982.97, 体现出程序在操作符和操作数方面的复杂度上升; 类和方法数量从 0.74 增加至 2.51, 进一步说明变异操作丰富了程序的结构与功能. 这充分表明 TscFuzz 生成的测试程序在代码结构和逻辑上更为复杂, 能够覆盖更多潜在的编译器缺陷场景.

表 4 原始程序和变异程序的平均复杂度

程序	代码行数	圈复杂度	解析树最大深度	最大嵌套深度	Halstead复杂度	类和方法数量
原始程序	13.23	2.35	16.08	0.38	3541.53	0.74
变异程序	36.68	2.53	20.20	0.55	3982.97	2.51

• 类型多样性分析: 第 2.1 节总结了 TypeScript 的 11 种特有类型. 为衡量 TscFuzz 生成程序的类型多样性, 通过分析变异前后程序中复杂类型结构所占比重来进行说明.

首先, 从语法程序数据集中随机选取 1000 个程序并使用 TscFuzz 对其进行变异. 为保证实验结果的准确性和可靠性, 重复 5 次实验. 我们采用遍历程序解析树的方式, 计算程序包含 11 种特有类型的语句数以及语句总数, 用两者的比值表示程序的类型多样性. 实验结果表明, 原始程序的类型多样性的比例是 18.36%, 变异程序则达到了 24.87%, 相比于原始程序提高了 35.46%. 充分说明了 TscFuzz 生成的测试程序在类型多样性上具有明显优势, 能够更全面地测试 TypeScript 编译器的类型系统.

其次, 为了验证大语言模型作为种子程序来源的有效性和差异性, 我们将其与 GitHub 开源程序集进行了类型多样性的量化对比. 本文使用第 2.1 节过滤后得到的 6473 个 LLM 生成的有效程序和 32353 个从 GitHub 收集的有效程序. 为了建立一个公平的比较基准, 从 GitHub 数据集中进行随机抽样, 选取了同等数量 (6473 个) 的程序. 为确保统计结果的稳定性并消除单次抽样的偏差, 该随机抽样过程重复执行了 10 轮, 取这 10 轮结果的平均值作为 GitHub 程序集的最终指标. 实验结果显示, GitHub 程序集的类型多样性的比例是 16.92%, LLM 程序集的类型多样性的比例是 20.07%. 这表明, LLM 生成的种子程序具有更高的特定类型的覆盖率, 能为后续变异提供高质量的输入.

回答 RQ4: 相较于原始程序, TscFuzz 生成的程序在 6 个复杂度指标均有所提升, 且程序中复杂类型结构所占比重平均提高了 35.46%. 表明 TscFuzz 能够生成具有较高复杂度和多样性的测试程序.

3.4 讨 论

在本文的研究中, 有两个方面的因素可能会影响实验结果的广泛适用性与准确性. 首先, 在生成测试程序的过程中, 本文使用了 ANTLR 4 提供的 TypeScript 的语法文件, 并以此为基础进行程序变异, 但由于 TypeScript 语言本身的快速迭代和更新, 该语法文件可能并不能完全覆盖所有最新版本的 TypeScript 语言特性. 这意味着, 尽管我们努力通过使用最新版本的 TypeScript 编译器进行测试, 所构建的测试用例可能仍无法全面涵盖所有可能的语法变化和特性. 这种不完全的语法覆盖使得我们无法完全测试 TypeScript 编译器在新特性下的行为, 进而可能影响缺陷检测的全面性. 此外, 无论是通过 GitHub 收集的程序, 还是使用大语言模型 (LLM) 生成的程序, 都具有一定的选择性偏差. 尽管这些程序在一定程度上提供了多样化的测试输入, 但仍然无法全面代表所有语法结构与用法. 因此在一定程度上限制了缺陷检测的全面性.

其次, 本文提出的基于 LLM 构建语法程序数据集这一组件中的提示词设计和类型特定变异算法是针对 TypeScript 语言的特定特性而优化的. 尽管这些方法在 TypeScript 的测试中表现出了较好的缺陷检测能力, 但它们并不一定适用于其他编程语言或编译工具. 由于其他语言的语法结构、类型系统或编译器的设计理念与 TypeScript 存在较大差异, 直接将这些方法移植到其他编程语言上可能会面临较高的难度, 且可能无法产生预期的效果. 因此, 尽管本文的研究为 TypeScript 编译器的缺陷检测提供了有效的解决方案, 但这些方法的普适性和可推广性仍然受到限制, 需要进一步的研究来探讨如何使这些方法适用于更多编程语言或编译器工具.

针对上述威胁, 未来的研究可以通过及时更新语法文件, 以确保测试能够涵盖 TypeScript 语言的最新特性, 并扩大测试用例的来源, 以提高语法覆盖的全面性. 同时, 可以探索如何将当前的方法进行泛化和改进, 使其能够适用于其他编程语言或工具, 尤其是在类型系统和编译过程有显著差异的情况下, 通过调整提示词设计和变异策略,

提高方法的可推广性.

4 相关工作

本文的工作和 JavaScript 引擎测试工作高度相关. 在过去几年中, 针对 JavaScript 引擎的自动化测试方法得到了广泛的研究和应用, 其中的程序生成和变异策略主要包括以下 3 类: 基于语法和抽象语法树 (AST)、基于语义分析以及结合神经网络. 这些方法通过不同的技术手段, 旨在提高 JavaScript 引擎缺陷检测的效率和覆盖范围.

基于语法和 AST 程序变异方法, 主要通过对 JavaScript 程序的抽象语法树进行变异, 以生成新的测试用例. 这些测试用例能够覆盖更多的语法路径和可能的执行场景. Dharma 是其中一个重要的工作, 它通过对 JavaScript 代码的 AST 进行变异, 从而生成具有高语法复杂性的测试用例[11]. LangFuzz 通过变异 AST, 生成具有不同语法结构的程序, 帮助发现 JavaScript 引擎在解析和执行过程中可能出现的缺陷[13]. EASTER 则结合了 AST 变异和程序分析, 提出了一种方法来识别并测试潜在的引擎缺陷[26]. 尽管这些方法在生成测试用例和覆盖不同语法路径方面表现出色, 但它们更关注语法层面的变异, 对于 JavaScript 程序的执行语义和潜在的运行时错误识别则较为有限.

基于语义分析的方法则专注于对程序的执行语义进行深度分析, 帮助检测引擎在运行时可能出现的错误. CodeAlchemist 是一个代表性工具, 通过对 JavaScript 程序的动态行为进行建模, 从语义层面分析程序执行过程中的潜在问题[10]. COMFORT 则在此基础上进一步扩展, 提出了一种基于动态分析的程序变异技术, 能够更准确地发现与执行语义相关的缺陷[20]. 这些方法的优势在于, 它们能够对 JavaScript 引擎的执行过程进行深度挖掘, 并生成语法和语义有效的 JavaScript 程序用于 JavaScript 引擎测试.

神经网络驱动的测试方法则是近年来兴起的一种新兴技术, 通过深度学习模型分析程序的结构和行为, 生成更符合引擎运行机制的测试用例. Montage 利用神经网络语言模型 (NNLM) 学习 AST 片段之间的关系, 并通过替换部分代码生成新的测试用例[23]. CovRL 使用强化学习方法优化测试用例生成, 进一步提升测试的覆盖率和准确性[28]. PMFuzz 则结合蒙特卡罗树搜索和神经网络技术, 生成更加多样化且高效的测试程序[29]. 这些方法通过机器学习模型的支持, 不仅提高了生成测试用例的质量, 还能够智能地探索潜在的错误路径, 从而提高缺陷发现的效率.

这些研究虽然在 JavaScript 引擎的测试中取得了显著进展, 但它们更关注 JavaScript 引擎的运行时特性, 而在处理 TypeScript 编译器及其复杂的类型系统时存在一定的局限性. TypeScript 作为 JavaScript 的超集, 具有更复杂的语法和类型检查机制, 因此需要专门设计针对 TypeScript 编译器的测试方法. 本文提出的 TscFuzz 框架, 通过结合基于大语言模型的程序构建方法和类型特定变异算法, 为 TypeScript 编译器的缺陷检测提供了新的研究视角.

5 总　结

本文设计并实现了一个基于语法和类型变异的 TypeScript 编译器测试框架 TscFuzz, 旨在检测 TypeScript 编译器中的缺陷. TscFuzz 结合基于大语言模型构建的语法程序数据集和类型特定变异算法, 提高了测试程序的语法和类型多样性, 对 TypeScript 类型系统进行了针对性和集中性的测试. 实验结果表明, TscFuzz 在真实世界的 TypeScript 编译器中发现了 12 个缺陷, 其中 8 个已被开发者确认, 7 个已被修复. TscFuzz 在 72 h 内发现了 5 个缺陷, 比基线方法 DIE 和 FuzzJIT 分别多检测了 2 个和 3 个 bug. TscFuzz 的故障检测效果显著优于基线方法. 此外, 实验验证了 TscFuzz 框架中各组件的有效性, 特别是基于大语言模型的特定语法程序数据集构建和类型特定变异算法, 这两个组件的协同作用显著提升了缺陷检测的效率和准确性. 另外, 本文还通过实验说明了 TscFuzz 能够生成复杂度和多样性较高的测试程序.

未来的研究可以针对 TypeScript 语言的新特性进行优化, 以进一步提高 TscFuzz 的缺陷发现能力. 同时, 可以调整和改进本文的变异算法和数据集构建方法, 扩展 TscFuzz 的功能, 提高其适用性, 例如可以进一步探索更先进的提示词工程技术 (如 Few-shot 学习), 通过向大语言模型提供高质量的示例代码, 提升生成程序的初始质量, 从而降低后续验证流程的开销, 提高整个种子程序集的构建效率. 此外, 可以引入变异测试方法作为补充评估维度, 通过植入多个精确触发编译器缺陷的程序变异体, 在受控环境中系统量化评估 TscFuzz 的有效性, 并与基线方法开

展更直接、可复现的对比. 最后, 还可以考虑将 TscFuzz 的应用扩展到 TypeScript 工具链的测试中, 为整个工具链的稳定性提供更全面的缺陷检测支持.

References

[1] Scarsbrook JD, Utting M, Ko RKL. TypeScript's evolution: An analysis of feature adoption over time. In: Proc. of the 20th Int'l Conf. on Mining Software Repositories (MSR). Melbourne: IEEE, 2023. 109–114. [doi: 10.1109/MSR59073.2023.00027]

[2] Bierman G, Abadi M, Torgersen M. Understanding TypeScript. In: Proc. of the 28th European Conf. on Object-oriented Programming (ECOOP 2014). Uppsala: Springer, 2014. 257–281. [doi: 10.1007/978-3-662-44202-9_11]

[3] Aggarwal S. Modern Web-development using ReactJS. Int'l Journal of Recent Research Aspects, 2018, 5(1): 133–137.

[4] Thakkar M. Building React Apps with Server-side Rendering. Berkeley: Apress, 2020. 41–152. [doi: 10.1007/978-1-4842-5869-9]

[5] Freeman A. Understanding Vue.js. In: Freeman A, ed. Pro Vue.js 2. Berkeley: Apress, 2018. 29–36. [doi: 10.1007/978-1-4842-3805-9_2]

[6] Green B, Seshadri S. AngularJS. Sebastopol: O'Reilly Media, Inc., 2013.

[7] Reid B, Treude C, Wagner M. Using the TypeScript compiler to fix erroneous Node.js snippets. In: Proc. of the 23rd IEEE Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM). Bogotá: IEEE, 2023. 220–230. [doi: 10.1109/SCAM59687.2023.00031]

[8] Chen JJ, Patra J, Pradel M, Xiong YF, Zhang HY, Hao D, Zhang L. A survey of compiler testing. ACM Computing Surveys, 2021, 53(1): 4. [doi: 10.1145/3363562]

[9] Xu HR, Wang YJ, Huang ZJ, Xie PD, Fan SH. Compiler fuzzing test case generation with feed-forward neural network. Ruan Jian Xue Bao/Journal of Software, 2022, 33(6): 1996–2011 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6565.htm [doi: 10.13328/j.cnki.jos.006565]

[10] Han HS, Oh DH, Cha SK. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines. In: Proc. of the 26th Annual Network and Distributed System Security Symp. (NDSS). San Diego: The Internet Society, 2019. 1–15. [doi: 10.14722/ndss.2019.23263]

[11] Generation-based, context-free grammar fuzzer. 2020. https://github.com/MozillaSecurity/dharma

[12] Patra J, Pradel M. Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data. Technical Report, TUDTUD-CS-2016-14664, TU Darmstadt, Department of Computer Science, 2016.

[13] Holler C, Herzig K, Zeller A. Fuzzing with code fragments. In: Proc. of the 21st USENIX Security Symp. Bellevue: USENIX Association, 2012. 445–458.

[14] Wang JJ, Chen BH, Wei L, Liu Y. Superion: Grammar-aware greybox fuzzing. In: Proc. of the 41st Int'l Conf. on Software Engineering (ICSE). Montreal: IEEE, 2019. 724–735. [doi: 10.1109/ICSE.2019.00081]

[15] Lima I, Silva J, Miranda B, Pinto G, d'Amorim M. Exposing bugs in JavaScript engines through test transplantation and differential testing. Software Quality Journal, 2021, 29(1): 129–158. [doi: 10.1007/s11219-020-09537-8]

[16] Li YK, Xue YX, Chen HX, Wu XH, Zhang C, Xie XF, Wang HJ, Liu Y. Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection. In: Proc. of the 27th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Tallinn: ACM, 2019. 533–544. [doi: 10.1145/3338906.3338975]

[17] Aschermann C, Frassetto T, Holz T, Jauernig P, Sadeghi AR, Teuchert D. NAUTILUS: Fishing for deep bugs with grammars. In: Proc. of the 26th Annual Network and Distributed System Security Symp. (NDSS). San Diego: The Internet Society, 2019. 1–15. [doi: 10.14722/ndss.2019.23412]

[18] Park J, An S, Youn D, Kim G, Ryu S. JEST: N+1-version differential testing of both JavaScript engines and specification. In: Proc. of the 43rd Int'l Conf. on Software Engineering (ICSE). Madrid: IEEE, 2021. 13–24. [doi: 10.1109/ICSE43902.2021.00015]

[19] Dinh ST, Cho H, Martin K, Oest A, Zeng K, Kapravelos A, Ahn GJ, Bao T, Wang RY, Doupé A, Shoshitaishvili Y. Favocado: Fuzzing the binding code of JavaScript engines using semantically correct test cases. In: Proc. of the 28th Annual Network and Distributed System Security Symp. The Internet Society, 2021. 1–15. [doi: 10.14722/ndss.2021.24224]

[20] Ye GX, Tang ZY, Tan SH, Huang SF, Fang DY, Sun XY, Bian LZ, Wang HB, Wang Z. Automated conformance testing for JavaScript engines via deep compiler fuzzing. In: Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation. ACM, 2021. 435–450. [doi: 10.1145/3453483.3454054]

[21] Tolksdorf S, Lehmann D, Pradel M. Interactive metamorphic testing of debuggers. In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Beijing: ACM, 2019. 273–283. [doi: 10.1145/3293882.3330567]

[22] Park S, Xu W, Yun I, Jang D, Kim T. Fuzzing JavaScript engines with aspect-preserving mutation. In: Proc. of the 2020 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2020. 1629–1642. [doi: 10.1109/SP40000.2020.00067]

[23] Lee S, Han HS, Cha SK, Son S. Montage: A neural network language model-guided JavaScript engine fuzzer. In: Proc. of the 29th USENIX Security Symp. USENIX Association, 2020. 2613–2630.
[24] Wang JJ, Zhang ZY, Liu S, Du XN, Chen JJ. FuzzJIT: Oracle-enhanced fuzzing for JavaScript engine JIT compiler. In: Proc. of the 32nd USENIX Security Symp. Anaheim: USENIX Association, 2023. 1865–1882.
[25] Chen L, Zhou ZD, Li XC, Jiang H. Detecting JavaScript transpiler bugs with grammar-guided mutation. In: Proc. of the 2023 IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Taipa: IEEE, 2023. 558–568. [doi: 10.1109/SANER56733.2023.00058]
[26] Tang S, Liu S, Wang JJ, Zhang XW. An Empirical Study on AST-level mutation-based fuzzing techniques for JavaScript Engines. In: Proc. of the 14th Asia-Pacific Symp. on Internetware. Hangzhou: ACM, 2023. 216–226. [doi: 10.1145/3609437.3609440]
[27] Zhang M, Belhadi A, Arcuri A. JavaScript SBST heuristics to enable effective fuzzing of NodeJS Web APIs. ACM Trans. on Software Engineering and Methodology, 2023, 32(6): 139. [doi: 10.1145/3593801]
[28] Eom J, Jeong S, Kwon T. CovRL: Fuzzing JavaScript engines with coverage-guided reinforcement learning for LLM-based mutation. arXiv:2402.12222, 2024.
[29] Xu HR, Wang YJ, Jiang ZY, Fan SH, Fu SJ, Xie PD. Fuzzing JavaScript engines with a syntax-aware neural program model. Computers & Security, 2024, 144: 103947. [doi: 10.1016/j.cose.2024.103947]
[30] Wang JM, Kang Y, Wu CG, Hu YH, Sun Y, Ren JK, Lai YM, Xie MY, Zhang C, Li T, Wang Z. OptFuzz: Optimization path guided fuzzing for JavaScript JIT compilers. In: Proc. of the 33rd USENIX Security Symp. Philadelphia: USENIX Association, 2024. 865–882.
[31] Wen M, Wang YC, Xia YF, Jin H. Evaluating seed selection for fuzzing JavaScript engines. Empirical Software Engineering, 2023, 28(6): 133. [doi: 10.1007/s10664-023-10340-9]
[32] Tian Y, Qin XJ, Gan ST. Research on fuzzing technology for JavaScript engines. In: Proc. of the 5th Int'l Conf. on Computer Science and Application Engineering. Sanya: ACM, 2021. 32. [doi: 10.1145/3487075.3487107]
[33] He XY, Xie XF, Li YK, Sun JW, Li F, Zou W, Liu Y, Yu L, Zhou JH, Shi WC, Huo W. SoFi: Reflection-augmented fuzzing for JavaScript engines. In: Proc. of the 2021 ACM SIGSAC Conf. on Computer and Communications Security. ACM, 2021. 2229–2242. [doi: 10.1145/3460120.3484823]
[34] Sun LL, Wu CG, Wang Z, Kang Y, Tang BW. KOP-fuzzer: A key-operation-based fuzzer for type confusion bugs in JavaScript engines. In: Proc. of the 46th Annual Computers, Software, and Applications Conf. (COMPSAC). Los Alamitos: IEEE, 2022. 757–766. [doi: 10.1109/COMPSAC54236.2022.00125]
[35] Bernhard L, Scharnowski T, Schloegel M, Blazytko T, Holz T. JIT-Picking: Differential fuzzing of JavaScript engines. In: Proc. of the 2022 ACM SIGSAC Conf. on Computer and Communications Security. Los Angeles: ACM, 2022. 351–364. [doi: 10.1145/3548606.3560624]
[36] Groß S, Koch S, Bernhard L, Holz T, Johns M. Fuzzilli: Fuzzing for JavaScript JIT compiler vulnerabilities. In: Proc. of the 30th Annual Network and Distributed System Security Symp. San Diego: The Internet Society, 2023. 1–17. [doi: 10.14722/ndss.2023.24290]
[37] Lin HY, Zhu JH, Peng JS, Zhu DX. Deity: Finding deep rooted bugs in JavaScript engines. In: Proc. of the 19th Int'l Conf. on Communication Technology (ICCT). Xi'an: IEEE, 2019. 1585–1594. [doi: 10.1109/ICCT46805.2019.8947153]
[38] Wang YL, Gong XQ, Chen H, Li J, Liu BY, Cao S. JSTIFuzz: Type-inference-based JavaScript engine fuzzing. In: Proc. of the 2020 Int'l Conf. on Networking and Network Applications (NaNA). Haikou City: IEEE, 2020. 381–387. [doi: 10.1109/NaNA51271.2020.00071]
[39] Parr T. The Definitive ANTLR 4 Reference. 2nd ed., Dallas: Pragmatic Bookshelf, 2013. 1–326.
[40] Rashid J, Mahmood T, Nisar MW. A study on software metrics and its impact on software quality. arXiv:1905.1292, 2019.
[41] McCabe TJ. A complexity measure. IEEE Trans. on Software Engineering, 1976, SE-2(4): 308–320. [doi: 10.1109/TSE.1976.233837]
[42] Alfadel M, Kobilica A, Hassine J. Evaluation of halstead and cyclomatic complexity metrics in measuring defect density. In: Proc. of the 9th IEEE-GCC Conf. and Exhibition (GCCCE). Manama: IEEE, 2017. 1–9. [doi: 10.1109/IEEEGCC.2017.8447959]
[43] Halstead MH. Elements of Software Science. New York: Elsevier Science Inc., 1977. 1–127.
[44] Helali RGM. A comparison between semantic and syntactic software metrics. Int'l Journal of Advanced Research in Computer and Communication Engineering, 2015, 4(2): 380–384. [doi: 10.17148/IJARCCE.2015.4286]
[45] Sharma M, Yu PS, Donaldson AF. RustSmith: Random differential compiler testing for Rust. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA 2023). Seattle: ACM, 2023. 1483–1486. [doi: 10.1145/3597926.3604919]
[46] Chen JJ, Suo CY. Boosting compiler testing via compiler optimization exploration. ACM Trans. on Software Engineering and Methodology, 2022, 31(4): 72. [doi: 10.1145/3508362]
[47] Li SH, Theodoridis T, Su ZD. Boosting compiler testing by injecting real-world code. Proc. of the ACM on Programming Languages, 2024, 8(PLDI): 156. [doi: 10.1145/3656386]

附中文参考文献

[9] 徐浩然, 王勇军, 黄志坚, 解培岱, 范书珲. 基于前馈神经网络的编译器测试用例生成方法. 软件学报, 2022, 33(6): 1996–2011. http://www.jos.org.cn/1000-9825/6565.htm [doi: 10.13328/j.cnki.jos.006565]

作者简介

任志磊, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为智能软件工程, 程序动态追踪.

高越, 硕士生, 主要研究领域为软件测试, 缺陷定位.

周志德, 博士, CCF 专业会员, 主要研究领域为智能软件工程, 可信编译, 深度学习优化.

敖伟, 硕士生, 主要研究领域为软件测试, 缺陷定位.

江贺, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为智能软件工程, 工业软件测试.