基于静态分析的JavaScript类型失配缺陷查找

魏 苗 吴毅坚 沈立炜 彭 鑫 赵文耘
(复旦大学软件学院 上海 201203) (上海市数据科学重点实验室(复旦大学) 上海 201203)

摘 要 由于JavaScript自身的语言特性,JavaScript程序中可能存在与运行时变量类型不匹配的缺陷,这类缺陷往往难以被察觉,只有在运行时报错后才能发现故障,而人工检查代码时需要开发者花费大量的时间通过调试的方法来定位查找代码缺陷。提出了一种静态分析JavaScript的方法来检查可能的运行时类型不匹配缺陷。该方法首先基于HTML和JSP页面对于JavaScript文件的引用将整个项目中的JavaScript文件进行分组;接着以分组为单位对JavaScript文件进行分析和变量类型推断,再检查每个分组中是否存在多类型属性;然后对这种多类型属性的使用进行检查;最后对检查结果进行报告,并给出修复建议。实现了一个用于自动检测JavaScript中多类型属性缺陷的工具,并通过在真实JavaScript项目中的实验证明了该方法的可行性,与已有的JavaScript分析方法相比,该方法的效果更优,提升了有关缺陷查找的效率与有效性。

关键词 静态分析,JavaScript,缺陷查找
中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.04.048

Finding Type Mismatch Defects of JavaScript Based on Static Analysis

WEI Miao WU Yi-jian SHEN Li-wei PENG Xin ZHAO Wen-yun
(School of Software, Fudan University, Shanghai 201203, China)
(Shanghai Key Laboratory of Data Science, Fudan University, Shanghai 201203, China)

Abstract Because of the nature of the JavaScript language and the increase of amount of JavaScript code with the evolving software, a JavaScript program may have a lot of defects which are related to the runtime variable type. This kind of defect is often difficult to detect, only when runtime errors can find fault. It takes programmers a lot of time to locate and search the code bug by debugging manually. The proposed JavaScript defect inspection method is mainly used to check the possible runtime type unmatched defects. First of all, the JavaScript file was grouped in the project based on HTML, JSP page reference for JavaScript files. Secondly, JavaScript files were analyzed in groups and the variable type was inferred. Then we checked whether there is a multi-type attribute in the group, afterwards the use of the multi-type attribute was checked. Finally, the checking results was reported and the repair advice was gave. A tool for automatic detection of multi-type attribute defect in JavaScript was implemented, through the experiment in the real JavaScript projects, the feasibility of this method was illustrated and the existing JavaScript analysis method was compared to illustrate the effectiveness of this method, improving the JavaScript's defect finding efficiency and effectiveness.

Keywords Static analysis, JavaScript, Defect finding

1 引言

随着Web2.0的到来,JavaScript成为一门越来越流行的语言,被广泛应用于Web应用。Node.js的出现使得JavaScript逐渐被用于编写服务端程序,因此JavaScript不再仅仅是一门基于浏览器的客户端脚本语言。不同于一般的编程语言,JavaScript拥有其特有的性质,首先,其是一种解释性的语言,即不需要编译便可直接运行;其次,JavaScript还具有弱类型、原型、闭包、函数是第一类对象以及能够与CSS(Cascading Style Sheet),HTML(HyperText Markup Language)进行代码的嵌套等特性。

在Java文件中可以通过import标识符得知当前文件依赖的其他文件。与其他的面向对象的语言(如Java,C++)的不同之处在于JS(JavaScript)文件中没有import机制,如果单独考察JS文件,很难知道该JS文件依赖于其他哪些文件。随着软件的不断演化,在一个项目中可能存在一些没有被使用的JS文件,因此在对某个项目的JavaScript程序进行分析时,不仅要考虑多个JS文件,而且应当对有关系的JS文件进行分组分析。如果仅仅只分析单个JS文件或者对某个项目的全部JS文件进行分析,都容易导致分析结果的不准确。

到稿日期:2016-03-04 返修日期:2016-05-21

魏 苗(1991-),女,硕士生,主要研究方向为软件工程,E-mail:13212010021@fudan.edu.cn;吴毅坚 男,博士,副教授,主要研究方向为软件体系结构、软件复用和产品线等;沈立炜 男,博士,副教授,主要研究方向为领域工程、软件产品线与自适应系统;彭 鑫 男,教授,博士生导师,主要研究方向为软件再工程、自适应软件系统;赵文耘 男,教授,博士生导师,主要研究方向为软件工程、电子商务。

JavaScript 是一种弱类型的语言，在声明 JavaScript 变量时，并不会显示指定变量的类型，而是通过 var 标识符声明变量，或者在非严格模式下，全局变量可以在函数内部不使用 var 直接声明。JavaScript 程序在运行时可以根据变量的值确定变量的数据类型，JavaScript 有 5 种原始数据类型：undefined，null，string，boolean 和 number。引用类型包括程序员自己通过 new 标识符创建的对象、Object 和 Array 等。

因 JavaScript 的语言特性和软件规模的扩大等因素，对 JavaScript 缺乏足够了解的程序员很难写出具有高维护性的程序。截止 2015 年底，著名 IT 技术问答网站 Stack Overflow[1] 统计出网站中与 JavaScript 相关的问题超过 100 万个，其中与 JavaScript 库文件 Jquery 相关的问题接近 70 万个。数据显示，目前 JavaScript 是 stack overflow 上问题数目最多的语言，这说明在实际的项目开发中开发者遇到许多与 JavaScript 相关的问题，可以推断出用 JavaScript 编写的 Web 应用可能会包含许多代码缺陷。

本文研究的 JavaScript 缺陷主要是与类型相关的缺陷检查，鉴于目前许多 Web 项目都基于 ECMAScript5，因此本文的研究工作是基于 ECMAScript5 开发的程序。在 Web 项目的开发过程中，由于项目的多人开发、缺少文档以及需求变更等因素，程序员在代码编写过程中并没有发现某个属性在运行时可能具有多种类型，从而对这个属性的使用存在问题，导致程序在运行时可能报错。

JavaScript 缺陷带来的挑战可以归纳为 3 个：

1）Web 项目中有许多的 JavaScript 文件，如何判断哪些 JavaScript 文件是有关联的？

2）JavaScript 是弱类型的语言，如何判断程序中变量的类型？

3）在找到多类型属性缺陷之后，如何向开发人员报告这些缺陷？

针对上述问题，提出了一种静态分析的方式去检测 JavaScript 文件中属性存在的多类型缺陷。首先依据 HTML 和 JSP（Java Server Pages）页面对 Web 项目中的 JavaScript 文件分组；然后依据函数调用关系对 JavaScript 文件中的变量进行类型推断；最后进行缺陷检测和报告。

本文第 2 节通过一个示例代码介绍了背景和挑战；第 3 节介绍了 JavaScript 缺陷检测方法；第 4 节介绍了基于静态分析的 JavaScript 缺陷查找的方法实现，并用实验验证了方法的可行性和效果；第 5 节主要介绍了目前检测 JavaScript 缺陷方面的相关工作；最后总结全文并展望未来。

2 背景和挑战

通过图 1—图 3 的示例来说明 JavaScript 类型在相关的缺陷查找过程中存在的挑战。在图 1—图 3 中存在 3 个 JS 文件：SuperType. js，SubType. js 和 Person. js。图 1 的 SuperType. js 文件中有一个构造函数 SuperType 和原型方法 getSuperValue；图 2 的 SubType. js 文件中有一个构造函数 SubType 和原型方法 getValue，SubType 继承于 SuperType；图 3 的 Person. js 文件中有一个构造函数 Person 和一个原型方法 sayName。

对于图 2 中第 7 行通过 new 表达式新创建的 Person 对象，假如程序员希望知道新创建的 Person 对象具有哪些属性并且这些属性具有什么类型，但是并不能直接知道。分析其原因在于 Person 函数的定义存在于另一个 JavaScript 文件中，并且当前 JS 文件中并没有特别的标识符来标识这个 JavaScript 文件对其他 JavaScript 文件的引用信息，而在本文的方法分析中会对有关联的多个 JavaScript 一起进行分析。

对于图 2 中第 8 行的 this. getSuperValue 方法，假如程序员希望知道这个被调用的函数是在哪里被定义的。若要解决上述问题，则首先需要弄清 this 的指向问题，在 JavaScript 中 this 的指向一般是动态确定的，但是在定义对象原型方法中的 this 一般是指向被定义的对象。其次，需要知道在这个对象及其原型链上是否存在这个方法的定义。在图 2 的文件中并不存在这个方法的定义，但在图 2 中第 5 行说明了其通过原型继承 SuperType，而且 SuperType 的定义是在另外一个文件中，所以要查找的方法实际上定义在图 1 的第 4 行中，本文分析会考虑 JavaScript 基于原型继承这个特性。

多人开发项目时，可能只是关心某个 API 的参数和返回值，忽略这个方法对于形参的影响。图 2 中第 7 行声明了一个 person 对象 p，由于在第 8 行调用了图 1 中第 4 行的 getSuperValue 方法，但是在这个方法中将 sayName 属性的类型修改为 Boolean 类型。因此在图 2 的第 9 行中调用 sayName 方法时会报错。本文关注的是多类型属性相关的缺陷，可以通过推断变量的类型去推断属性的类型。

```
1. function SuperType(){
2. this. property=true;
3. }
4. SuperType. prototype. getSuperValue=function(p){
5. //…
6. var v=p;
7. p. sayName=true;
8. //…
9. }
```

图 1 SuperType. js 中的内容

```
1. function SubType(){
2.   this. subproperty=false;
3. }
4.   //inherit from SuperType
5.   SubType. prototype=new SuperType();
6.   SubType. prototype. getValue=function(){
7.       var p=new Person('tom');
8.       this. getSuperValue(p);
9.       return p. sayName();
10.  }
```

图 2 SubType. js 文件中的内容

```
1.   function Person(name){
2.       this. name=name;
3.   }
4.   Person. prototype. sayName=function (){
5.     return this. name;
6.   }
```

图 3 Person. js 文件中的内容

由于 JavaScript 的语言特性,手动地分析和检测 JavaScript代码中的代码缺陷比较繁琐,而且容易出错,因此需要自动化的工具帮助开发者检测代码中潜藏的代码缺陷,帮助开发者提早发现代码中潜藏的一些问题,提高开发者的工作效率并缩短软件开发周期。

3　基于类型推导的缺陷查找方法

3.1　概述

本文提出的 JavaScript 缺陷检测主要基于以下场景:程序中的某个属性在运行时可能具有多种类型,但是在使用该属性时并没有进行相关的类型检查判断,导致程序运行时出现错误。

由于 JavaScript 中的回调函数、事件机制、方法的动态调用,很难静态分析出哪些属性是有联系的,因此本文认为具有相同属性名的字段是可能有联系的。尽管这样可能导致分析的结果不够准确,但是会尽可能提高相关缺陷的查全率,因此只要控制好最后向程序员给出的推荐建议的数量,最终的检测结果对程序员而言就是非常有意义的。

图 4 示出了 JavaScript 缺陷检查方法的流程:(1)读取待分析 Web 项目中的 HTML 和 JSP 页面,根据 HTML 和 JSP 页面对 JavaScript 文件的引用,对 JS 文件进行分组;通过正则表达式找到页面中所有的〈script〉标签,并依据页面路径信息和 script 标签中的路径信息求得 JS 文件的绝对路径,并使用绝对路径作为文件的标识。(2)对每个分组内的 JS 文件进行分析,通过 esprima. js[2]将每一个 JS 文件解析为一棵抽象语法树(AST),并在树的根节点上添加文件名信息,然后将多棵 AST 合并为一棵 AST。(3)查找变量的引用链,为后面的类型推断做准备,这里考虑了多文件、原型继承、this 指向等特性。(4)根据类型推导规则对变量的类型进行推断,找到每个变量可能的数据类型。(5)根据推断出的数据类型,检测相关的变量是否可能涉及代码缺陷。(6)对检查的结果进行报告(文字和 IDE 导航等形式),并给程序员相应的修复建议。

图 4 中,步骤(1)、步骤(2)是分析前的准备工作,主要目的是简化被分析代码的复杂关联关系,得到有意义的 JS 代码集合,并生成 AST;步骤(6)主要是结果展示。鉴于这些技术相对成熟,本文后续将主要介绍步骤(3)(见 3.2 节)、步骤(4)(见 3.3 节)和步骤(5)(见 3.4 节)。

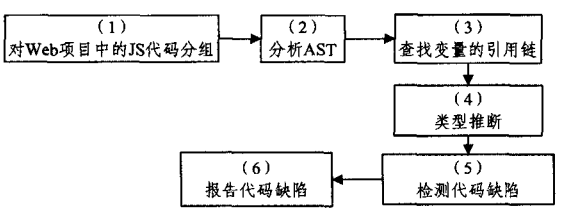

图 4　JavaScript 缺陷检查方法流程

3.2　引用链分析

查找变量引用链的目的在于为后面的类型推断做准备。依据函数调用实参的类型推断被调用函数形参的类型;若某个变量的值等于某个函数的返回值,则可依据函数调用的返回值类型来推断变量的类型。文献[15]提出了一种基于字段的方法来构造函数调用图,但是该分析方法忽略了属性访问中的对象。例如对于属性访问 e. f,该分析方法不会考虑 f 属性所在的对象 e,而是直接去查找 f 所在的位置。本文的分析方法考虑了原型继承、对象属性等性质。

由于 JavaScript 的作用域是基于函数的,因此需要考虑作用域下定义的变量以及这些变量具有哪些属性。通过 escope[3]可以得到每个作用域下有哪些变量,主要考虑通过点操作符和对象直接量两种方式来实现变量属性的添加。

(1)通过点操作符添加。赋值语句的左边是成员表达式,这种方式是直接在变量上添加属性。下例中假设原型属性不存在 Bar 属性,因此在步骤 4 给变量 Foo 添加了一个原型属性 Bar。

```
1. function Foo(x){
2 . this. x=x;
3. }
4. Foo. prototype. Bar=function (){…}
```

(2)对象直接量。赋值语句的右边是对象表达式,下例给变量 mxEvent 添加了两个属性即 addListener 和 removeListener。

```
1. varmxEvent={
2. addListener:function(){…},
3. removeListener:function(){…}
4. }
```

由于变量的直接引用是一种常见的方式,因此这里主要考察函数变量的直接引用,表 1 列出举了常见的函数调用方式。对于第一种类型(见表 1 第一行):若函数调用的表达式只是一个标识符,可以沿着作用域查找,直到找到被调用的函数点。对于第二种类型(见表 1 第二行):由于在 JavaScript 中 this 的指向是运行时动态确定的,因此这里只考虑原型方法中的 this 指向。由于第二种类型的例子中 this 指向的是 A,因此找到变量 A 的定义,然后在变量 A 的属性和原型链中查找 a。对于第三种类型(见表 1 第三行):首先沿着作用域查找对象的定义,然后在对象的属性和原型属性中查找被调用的属性。

表 1　常见的函数调用方式

序号	表达式类型	例子
1	Identifier	A()
2	this 表达式	A. prototype. b=function(){ this. a(); }
3	属性表达式	a. b()

对于 new 表达式,若 new 表达式在一个赋值语句的右侧,则不仅需要找到 new 表达式中被调用函数的定义,而且需要给左侧的变量添加属性。下例中,通过 new 表达式新创建一个对象,需要给 mxGraph 的原型属性添加属性。这里考虑了 JavaScript 的语言特性原型继承,以便方法调用找到其函数调用点。

```
1. mxGraph. prototype=new mxEventSource();
```

3.3　类型推导规则

类型代表了值集合的抽象,每个具体的值都有一个类型。通过对程序进行分析再加上一些推断规则和上下文分析,可以推断某个变量或者表达式可能的类型。本文根据文献[4-5]和自己的开发经验,调查并总结了几种类型推导规则,如表 2、表 3 所列。

表 2 类型推断的基本规则

规则式	说明
$\frac{l}{\Gamma \vdash l: T(l)}$	若是一个字面量，则字面量的类型可以直接由字面量确定
$\frac{\text{typeof } e==l}{\Gamma \vdash e: l}$ [typeof]	若对于一个表达式，使用 typeof 进行判断，则这个表达式的类型可能与等式右边的字面量值相同
$\frac{e++}{\Gamma \vdash e++: n; \Gamma \vdash e: n}$ [++ --]	若一个表达式是自增(++)或者自减(--)，那么这个表达式中的变量类型可能是 number，整个表达式的类型也是 number
$\frac{function\ v()\{\cdots\}}{\Gamma \vdash v: f}$ [function]	对于一个函数定义或者函数表达式，可以推断出这个表达式的类型是 function
$\frac{e1\ \text{op}\ e2}{\text{op} \in \{-, *, /, \%\}\ \Gamma \vdash e1: n; \Gamma \vdash e2: n}$	若两个表达式做减、乘、除法的算术运算，那么这两个表达式的类型都是 number
$\frac{e1\ \text{op}\ e2}{\text{op} \in \{-=, *=, /=\}\ \Gamma \vdash e1: n; \Gamma \vdash e2: n}$	在一个赋值表达式的操作符是--=，* =，/=，那么表达式两边的类型都是 number

表 3 类型推断的推导规则

规则式	说明
$\frac{\Gamma \vdash e1: T\ v=e1}{\Gamma \vdash e2: T}$ [=]	在一个赋值语句中左边变量的类型与右边表达式的类型相同
$\frac{\Gamma \vdash l: T\ e2==l}{\Gamma \vdash e2: T}$ [==]	在一个相等判断中，若有一个表达式是一个字面量，那么推断另外一个表达式的类型与该字面量类型相同
$\frac{e1+e2}{\text{见说明}}$ [+]	若 e1 和 e2 表达式中有任何一个表达式是 string 类型，那么结果集为 string，如果 e1 和 e2 的类型都是 number，则结果集是 number
$\frac{e1\ \text{op}\ l\ \text{op} \in \{<, \leqslant, >, \geqslant\}}{\text{见说明}}$	在一个比较操作中，若可以根据字面量的类型确定其中一个表达式的类型为 string 或者 number，那么另外一个表达式的类型也为 string 或者 number
$\frac{new\ e2(e1)}{\text{见说明}}$ [new-call]	若一个表达式是一个 new 表达式，那么这个 new 表达式的类型是一个类型为 e2 的对象，并且可以根据函数调用关系推断出新产生对象属性的类型

类型推断的规则采用了 Damas 等人的记法[6]，如下所示。

$$\frac{\text{Hypothesis}_1 \cdots \text{Hypothesis}_n}{\text{Conclusion}}$$

Hypothesis 表示假设，Conclusion 表示结论；如果所有的假设是正确的，那么结论也是正确的。

类型推断的符号规则如下：

$A \vdash e: \sigma$

其中，A 表示变量类型推断的函数，或者说是一种映射，即变量到类型的映射；e 表示表达式；σ 表示类型；$\vdash$ 表示可证明表达式 e 具有类型 σ。

规则中需要使用的符号表示如下：b：boolean，s：string，n：number，f：function，o：object，v：variable，l：literal，T(e)：表达式 e 的类型，表达式：{e，e1，e2}。

表 2 列出了类型推导的基本规则，即对 AST 进行遍历，根据常量的类型或者特定的算符直接确定变量的类型。

表 3 列出了类型推导的推理规则，即根据条件进行类型的推理。不能直接根据 AST 得到某个变量的类型，需要借助函数调用等分析手段分析某个变量或者表达式具有的类型或者属性。

3.4 检测代码缺陷

在变量类型推断的基础上检测代码中潜藏的缺陷，检测 JavaScript 代码缺陷的算法如下。

```
input：AST
output：多类型属性的缺陷报告
//合并 AST
Method MergeAST(ASTS)
begin
    新创建一个 root 节点，并将所有的 AST 添加到 root 节点上
end；
Method callSite(AST)
begin
    给 ast 树上的每个节点添加 scope
    分析每个作用域下变量的属性
    查找函数调用和 new 表达式的调用点
end；
Method TypeInfer(AST)
begin
    遍历 AST，根据类型推断规则，推断变量或者是表达式的类型
end；
Method CheckType(AST)
begin
    检查同属性名的多类型属性名
    检查对于多类型属性的使用
Method showResult(properties)
begin
    给出属性所在的文件、行数、类型信息
end；
```

在代码缺陷检查阶段，收集整个项目中同名属性的类型，然后对属性的类型进行检查，查看同名属性的类型是否具有多种类型，然后检查多种类型的属性和多类型的变量在使用之前有无类型相关的条件判断，若没有条件判断，则认为这是一个可能的缺陷。若在一个多类型的属性中存在 function 类型，则检查在属性的方法调用使用之前是否曾判断这个类型为 function。若在一个多类型的属性中存在 number 类型，则检查在算法运算(这里是 * ，/，-)之前是否曾判断过这个类型为 number。

4 工具实现与实验

为了验证本文方法检测 JavaScript 代码缺陷时是否具有可行性及是否有效果，作者实现了一个基于 Nodejs 的检测工具 typeMismatch。该工具将整个 Web 项目作为输入，通过该工具可以自动地对项目中的 JS 文件进行分组，检测每个分组的 JS 文件中可能潜藏的代码缺陷，并将检测到的代码缺陷报告给开发人员。

实验对象为 1 个绘制流程图的 Web 构件项目、1 个企业级科教 Web 项目以及 3 个开源项目(CollegeVis[16]，p4wm[17]，wander-mesh[18])，从项目中选取了相关功能的部分 JS 文件和在开源项目中手动植入一些 bug。表 4 列出了实验对象信息和缺陷查找结果，第 1 列 Source 代表项目来

源,第 2 列 Tl 代表是否包含第三方库文件,第 3 列 FN 代表 JS 文件个数,第 4 列 LOC 代表 JS 代码行数,第 5 列代表本文提出的 typeMismatch 工具检出的可能的 bug 数和有效的 bug 数,第 6 列代表 flow[19] 工具检出的可能的 bug 数、有效的 bug 数以及类型相关的 bug 数目。

表 4　实验对象信息和缺陷查找结果

Source	Tl	FN	LOC	可能 bug|有效 bug (type Mismatch)	可能 bug|有效 bug|类型相关的 bug (flow)
绘制流程图项目	是	5	13411	2|1	11|1|1
科教项目	是	13	13553	11|1	44|22|0
CollegeVis	是	4	1166	14|4	0|0|0
p4wn	否	4	1628	1|0	20|5|0
wander-mesh	是	9	2417	3|2	27|6|1

从表 4 的实验结果可知,当 JS 文件数目较多时,可能的 Bug 数目就越多。一般,文件数目多的项目都是由多人开发的,但是不同开发者的编码习惯不同,在编码过程中,同一个属性名可能被定义为一个函数,也可能将其作为一个 boolean 类型来使用。

通过本文提出的工具检测发现,绘制流程图的 Web 构件项目在运行中的双击按钮的情况下会产生类型错误,需要做类型保护。存在的有效 bug 是一个属性名 isConsumed,其可能具有 function 和 boolean 类型,且分布在不同的 JS 文件中,给出是在哪个 JS 文件的哪一行来推导出这个变量类型,并对这种属性的使用给出报告,例如:"mxEvent. js 第 164 行,由于该属性 isConsumed 可能是一种多种类型的变量 function, boolean,在进行方法调用之前,需要在 if 条件语句中进行判断"。存在的无效 bug 是一个属性名 width,其可能具有 string 和 number 类型,但是在算术运算之前没有相应的类型保护,该错误在目前项目的运行中不会发生,但是若有类型保护则会使得程序更加强壮。企业级科教 Web 项目在运行中有一个 step 属性会产生类型错误,对于其他 10 个无效 bug,程序在运行时不会出现问题,但是程序员可以依据这些信息对项目中的代码进行检测。在开源项目 CollegeVis 中,文件行数不多却检测出较多 bug 的原因在于这个项目中引入的第三方库文件的代码是压缩文件,因此代码行数较少。在该项目中 typeMismatch 能检测出 flow 但不能检测出类型错误的原因在于 flow 不对第三方库文件做类型检测。在开源项目中不仅可以检测出手动植入的 bug,也能检测出一些类型相关的 bug。

目前对于 JavaScript 代码中的类型错误,常用的检测方法是基于数据流和控制流的分析方法,flow 是由 Facebook 提出的一个基于数据流和控制流分析方法的开源工具,主要用于检测 JavaScript 代码中的类型错误。表 4 列出了本文提出的工具 typeMismatch 和现在已有的 JavaScript 类型错误检测工具 flow 之间的 bug 检测对比结果,发现 flow 一般能检测出更多的 bug 数量,但是存在较多无效和有效的 bug,而且本文提出的工具能够检测出 flow 不能检测出的类型错误。这个工具在使用上比较不便,需要开发者在待检查的 JS 文件头部添加注释说明,而且对于第三方库文件以及自定义的全局变量,需要在配置文件中编写接口的定义,本文提出的方法不需要开发者在 JS 文件中增加任何配置,而且会对多个 JS 文件进行分析,自动地识别出多个 JS 文件中定义的全局变量,不需要其他配置文件。flow 对变量的属性分析不够完备,若一个变量为通过 new Object()方式创建的对象,则 flow 会忽略后面通过点操作符添加的属性,在对该属性进行引用时,flow 找不到该属性的定义,本文的分析方法会考虑通过点操作符增加的属性。flow 检查出的 bug 并非全是 bug,只是 flow 给的建议,比如在声明数组变量时 flow 会建议开发者不要通过 new 表达式来创建,而是通过字面量的方式来直接声明。flow 不允许对 null 值的访问,一个变量可能由于函数调用顺序的不同导致这个变量在运行时可能为 null,flow 会在对这个变量进行访问时报错。由于 flow 考虑的 JS 错误类型较多,因此检测出来的可能的 bug 数量和有效的 bug 数量也较多。在科教项目中,使用 flow 工具检查出的 22 个有效 bug 中有 16 个是与全局变量有关的,其他 6 个是与 null 值有关的,且没有检查出 typeMismatch 工具检测的 bug。科教项目存在较多没有通过 var 变量声明的全局变量,容易造成全局变量污染。比如存在一个变量 position,在有的函数内部通过 var 声明而在有的函数内部并没有通过 var 声明就去使用这个变量,position 变量成为了一个全局变量,但是并没有在 JS 文件中进行显示的声明,这给程序的理解和维护带来了不便,很有可能导致程序在运行时出现错误。

本文提出的缺陷检查工具可以在实际开发中帮助开发人员和维护人员。对于开发人员而言,可以对自己编写的代码进行缺陷检查,查看是否存在多类型的,以属性使用错误或者是哪些属性是多类型的为代码重构提供基础;对于代码的维护者而言,可以通过检查哪些属性是多类型的,从而指导维护者正确地使用这些属性。

首先,由于分析变量的引用和类型推断的局限性,在实际检测中并不能百分之百地推导出所有变量的类型,可能会漏报一些类型相关的缺陷;其次,本文采用的相同属性名的字段是可能有联系的,因此存在误报的情况;最后,本文的方法可能存在误报和漏报的情况。但是静态分析方法因为局限性可能会导致漏报,为了尽可能地提高相关缺陷的查全率,误报可以忍受。

5　相关工作

在 JavaScript 程序分析方面,Guarnieri 等人[7] 利用指针分析加上 9 个与安全和可靠性有关的策略来检测 JavaScript 代码中是否存在违反策略的情况;Madsen 等人[8] 利用指针分析及使用分析,主要对库文件的使用进行分析;Wei 等人[9] 采用静态分析和动态分析相结合的方法来处理 JavaScript 的动态性。本文主要是对 JavaScript 中的变量做引用链分析和类型推导。

在有关 JavaScript 类型系统的方面,Jensen 等人[10] 提出的类型分析;基于流敏感的数据流分析;Vardoulakis 等人[11]

提出的类型分析基于先进后出的流分析；Feldthaus 等人[4]基于对象的显示做类型推断，但是他们提出的方法忽略了大多数的函数调用；李世胜等人[5]提出了类型预测算法和基于位置的内联算法来处理 JavaScript 程序中的元数据和对象的类型，但其分析方法只对函数范围内的变量、操作数的类型以及全局数据类型进行流相关的分析。本文提出的类型推断考虑了多个 JS 文件、函数范围内及函数之间的变量做类型推断。

Ocariza 等人[12]对与 JavaScirpt 相关的 bug 报告做了一项经验研究，发现 JavaScript 的很多错误都与 DOM 相关，而且与 DOM 相关的错误比其他 JavaScript 错误需要花费更多的时间成本。他们开发了 Vejovis 工具[13]用于处理与 DOM 相关的错误，假设 DOM 访问方法的参数不正确，提出了一种查找替换算法，为原来的 DOM 访问方法的参数尝试找到了有效的替代，找到后依据代码所在的环境决定给程序员相应的提示信息。

Feldthaus 等人[4]提出了基于字段的流分析构造调用图，然后根据函数调用图分析程序。例如查找代码中潜藏的错误时，有的函数有时通过 new 使用，有时通过普通的调用来使用，对于函数的这种使用通常潜藏着代码问题。

Ocariza 等人[14]提出了静态分析 AUREBESH 方法来检测在使用 JavaScript 的 MVC 框架时标识符和类型不一致的问题。目前该方法应用于前端开发中广泛使用的 MVC 框架 AngularJS 中，其对 22 个使用 AngularJS 的 Web 应用进行分析，发现了 15 个不一致的 bug。

上述方法都解决了一些与 JavaScript 相关的代码缺陷，但是对于 JavaScript 中的属性在代码中可能具有多种类型的情况，却鲜少有人考虑。

结束语 本文提出的方法在程序员开发 JavaScript 代码时能够较为准确地自动获取与多类型属性相关的类型缺陷信息。对于项目中的 JavaScript 文件，其能够依据实际项目中的 HTML 和 JSP 页面对 JavaScript 文件的使用进行分组讨论。在类型推断时，其考虑了原型继承和对象的方法调用类型、部分 this 方法调用。本方法能检测出项目中多类型的属性，并对它们的使用进行检测，最后对检测出的缺陷进行报告。本文提出的 JavaScript 类型缺陷检测方法和已有的 JavaScript 类型缺陷检测方法相比具有以下效果：使用方便，不需要在代码中添加注释和在项目中增加配置文件；增加了类型错误检测的种类。

尽管如此，本文提出的静态分析 JavaScript 缺陷检测方法还需要解决一些问题，例如类型推断时忽略了属性的动态访问，由于静态分析较难实现，需要引入更多的分析技术来进行处理。

本文的后续工作旨在提高类型推断的准确性和解决其他类型的 JavaScript 缺陷；随着 EcmaScript6 的引入，可能目前的一些缺陷不存在，但是也可能引入新的缺陷，后续也将进一步检测新类型的错误。

参 考 文 献

[1] stackoverflow [OL]. http://stackoverflow.com.

[2] esprima[OL]. http://esprima.org.

[3] escope[OL]. https://github.com/estools/escope.

[4] FELDTHAUS A, MØLLER A. Semi-automatic rename refactoring for JavaScript[J]. AcmSigplan Notices, 2013, 48(10): 323-338.

[5] LI S S, CHENG B Q, LI X F, et al. JavaScript Typing System with Prediction[J]. Journal of Computer Research and Development, 2012, 49(2): 421-431. (in Chinese)
李世胜，程步奇，李晓峰，等. 基于预测的 JavaScript 类型系统研究[J]. 计算机研究与发展，2012，49(2)：421-431.

[6] DAMAS L, MILNER R. Principal type-schemes for functional programs[C] // Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, 1982: 207-212.

[7] GUARNIERI S, LIVSHITS B. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code[J]. Washington Sammyg, 2009, 7(4): 151-168.

[8] MADSEN M, LIVSHITS B, FANNING M. Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries[C] // Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 2012: 499-509.

[9] WEI S, RYDER B G. Practical blended taint analysis for JavaScript[C] // Proceedings of the 2013 International Symposium on Software Testing and Analysis. ACM, 2013: 336-346.

[10] JENSEN S H, MØLLER A, THIEMANN P. Type Analysis for JavaScript[M] // Static Analysis. Springer Berlin Heidelberg, 2009: 238-255.

[11] VARDOULAKIS D. CFA2: Pushdown Flow Analysis for Higher-Order Languages[D]. Boston: Northeastern University, 2012

[12] OCARIZA F, BAJAJ K, PATTABIRAMAN K, et al. An Empirical Study of Client-Side JavaScript Bugs[C] // 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. IEEE Computer Society, 2013: 55-64.

[13] OCARIZA F S, PATTABIRMAN K, MESBAH A. Vejovis: Suggesting Fixes for JavaScript Faults[C] // International Conference on Software Engineering. 2014: 837-847.

[14] OCARIZA F S, PATTABIRMAN K, MESBAH A. Detecting Inconsistencies in JavaScript MVC Applications[C] // 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE). IEEE, 2015: 325-335.

[15] FELDTHAUS A, SCH &#, FER M, et al. Efficient construction of approximate call graphs for JavaScript IDE services[C] // International Conference on Software Engineering. IEEE Press, 2013: 752-761.

[16] collegesvis[OL]. https://github.com/nerdyworm/collegesvis.

[17] p4wn[OL]. https://github.com/douglasbagnall/p4wn.

[18] wander-mesh[OL]. https://github.com/notlion/wander-mesh.

[19] flow[OL]. http://flowtype.org.