基于规则的高风险动态类型代码检测研究

陈芝菲, 郝洋, 陈林, 肖亮

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.221100244

基于规则的高风险动态类型代码检测研究

陈芝菲[1] 郝 洋[1] 陈 林[2] 肖 亮[1]
1 南京理工大学计算机科学与工程学院 南京 210094
2 南京大学计算机软件新技术国家重点实验室 南京 210023

摘 要 近年来,Python的应用呈爆炸式增长。虽然Python的动态类型特性为开发人员提供了强大的编程抽象能力,但同样也导致代码库中聚集了与类型相关的缺陷。为了减少软件代码中的类型缺陷,文中分析并检测了6种可能导致类型缺陷的高风险动态类型代码。首先,形式化地描述了每种类型的高风险动态类型代码的规则;然后,提出了一种基于规则的高风险动态类型代码检测技术;最后,对25个Python开源软件项目(总规模超过945kLOC)展开了实验评估。结果表明,高风险动态类型代码在开源软件项目中广泛存在,尤其是单个Python函数中可能存在多处变量类型不一致的代码,而基于规则的检测技术在Python软件项目中具有较高的准确率和较好的性能表现。针对高风险动态类型代码的检测技术及实验结论,将为动态类型特性的良性发展以及软件项目的质量保障提供有力的参考和支持。

关键词:Python;动态类型;类型缺陷;开源软件;高风险动态类型代码;检测技术
中图法分类号 TP311

Rule-based Technique for Detecting Risky Dynamic Typing Code

CHEN Zhifei[1],HAO Yang[1],CHEN Lin[2] and XIAO Liang[1]
1 School of Computer Science and Engineering,Nanjing University of Science and Technology,Nanjing 210094,China
2 State Key Laboratory for Novel Software Technology,Nanjing University,Nanjing 210023,China

Abstract Python has experienced an explosive growth in application in recent years,especially in scientific computing and machine learning areas. While Python's dynamically-typed nature provides developers with powerful programming abstractions,that same dynamic type system allows for type-related defects to accumulate in code bases. To address these issues,this paper investigates six types of risky dynamic typing code which could incur type-related defects. This paper formally describes the rule of each type of risky dynamic typing code and then proposes a rule-based technique for detecting them. We conduct a study on 25 real-world Python open-source software projects (with the total size of more than 945kLOC). The results show that risky dynamic typing code is widespread in open-source software projects and a single Python method could gather multiple instances of inconsistent variable types,and the rule-based detection technique achieves high detection accuracy and good performance. The technique for detecting risky dynamic typing code and findings from this work will provide a strong reference and support for healthy evolvement of dynamic typing feature and quality assurance of software projects.

Keywords Python,Dynamic typing,Type defects,Open-source software,Risky dynamic typing code,Detection technique

1 引言

近年来,Python已成为最流行的编程语言之一。根据IEEE Spectrum对顶级编程语言的排名[1)],Python已连续多年成为各领域应用程序使用的主要语言,包括科学计算、Web、机器学习等领域。然而,作为一个典型动态类型语言,Python中的变量无需类型定义,其属性结构也可以动态变更。近期研究发现,动态类型系统降低了软件开发效率[1-2]、代码可用性[3]和代码质量[4-8],尤其是Python程序中存在一类常见的运行时错误,即TypeError异常。这是由于Python中变量类型是动态变更的,当对错误类型的值执行操作时会引发运行时错误。例如,在缺陷报告numpy/numpy/#17466[1)]中,执行"$a+=0.1$"语句时抛出TypeError异常,因为数组a中的元素类型为dtype('int64'),而Python的add

1) https://spectrum.ieee.org/top-programming-languages-2022

到稿日期:2022-11-29 返修日期:2023-03-09
基金项目:国家重点研发计划(2022YFF0712100);江苏省自然科学基金(BK20220937);江苏省研究生科研与实践创新计划项目(KYCX22_0465)
This work was supported by the National Key Research and Development Program of China(2022YFF0712100),Natural Science Foundation of Jiangsu Province,China(BK20220937) and Postgraduate Research & Practice Innovation Program of Jiangsu Province(KYCX22_0465).
通信作者:陈芝菲(chenzhifei@njust.edu.cn)

操作的结果无法从 dtype('float64')强制类型转换为 dtype('int64')。事实上,"+="运算符重用相同的内存,因此结果类型必须保持相同。此外,由于类型对象的属性也支持动态变更,当对特定类型的对象执行不支持的属性操作时也会引发运行时错误,抛出 AttributeError 异常。例如,numpy/numpy/#16687[2]报告在调用一个 numpy.ndarray 类型对象的 abs()函数时抛出了 AttributeError 异常,因为该类型对象不存在 abs 这个属性。对此,本文收集了 GitHub 平台[3]上 100 个收藏数最多的 Python 项目,抽取了这些项目缺陷报告中记录的所有发生的异常类型,发现其中有 65 个项目中 TypeError 位列最常见异常类型前三位,同时有 66 个项目中 AttributeError 位列最常见异常类型前三位。

无论是 TypeError 异常还是 AttributeError 异常,均是因动态类型机制下程序员对一个对象的类型或结构的错误理解而造成的,因此识别动态类型代码中的潜在风险至关重要。面对动态类型的这些副作用,Python 开发人员开始关注类型测试,但类型测试并不是一件容易的事情。我们发现诸多缺陷报告都是关于类型测试的问题,例如指出开发人员需要花费大量精力来处理类型测试[4]。

为了预防和减少这类类型缺陷,本文研究了 Python 动态类型相关的 6 类高风险动态类型代码,包括赋值类型不一致、参数类型不一致、变量类型不一致、动态删除元素、动态删除属性和动态访问属性,它们均在可读性或可理解性上违反了软件可维护性原则。目前已有一些研究工作提出了类似的动态类型相关的反模式,包括 Python 语言[9]和 JavaScript 语言[10-11],本文在这些研究工作的基础上进一步对 Python 程序中常见的高风险动态类型代码形式进行了调研总结并进行了形式化表示。首先形式化描述了 6 类高风险动态类型代码规则,然后提出了一种基于规则的技术来检测 Python 软件中的高风险动态类型代码,为程序员提供重构机会,从而避免动态类型滥用造成的软件运行时异常。最后,通过实验评估了本文提出的高风险动态类型代码检测技术的精度、召回率和效率,并分析了 25 个 Python 开源软件中高风险动态类型代码的分布规律。

实验结果表明,基于规则的高风险动态类型代码检测技术的精度和召回率分别高达 97%和 98%,其中误报和漏报主要来自变量类型不一致的检测。在检测效率方面,平均检测一个软件项目需要花费 2.5min,主要受限于类型分析任务的复杂性。基于该检测技术,实验发现所选的 25 个软件项目均存在高风险动态类型代码(平均每个项目有 153 处实例),其中最普遍存在的一类是变量类型不一致,而且一个 Python 函数会包含多处高风险动态类型代码。与之相反,参数类型不一致和动态属性删除这两类高风险动态类型代码较少出现。

本文的主要贡献如下:

(1)形式化描述了 Python 程序中的 6 类高风险动态类型代码规则。

(2)提出了一种基于规则的 Python 高风险动态类型代码检测技术。

(3)评估了基于规则的高风险动态类型代码检测技术在 25 个 Python 开源软件项目上的有效性和效率。

(4)探索了 Python 开源软件中高风险动态类型代码的分布规律。

本文第 2 节介绍了动态类型和类型缺陷的相关工作;第 3 节介绍了 6 类高风险动态类型代码并描述了其形式化规则;第 4 节展示了基于规则的检测技术;第 5 节和第 6 节分别描述了实验设计和实验结果;第 7 节讨论了本文研究工作的不足;最后总结全文并展望未来。

2 相关研究

动态类型是程序设计语言最重要的动态特性之一。目前已有许多学者研究动态编程语言的动态特性,涉及的语言包括 Smalltalk[12],JavaScript[13],AspectJ[14]和 Python[15-18]。这些研究工作主要通过追踪具有动态特性的代码的执行,来观察动态特性代码的使用场景,分析动态特性于程序质量的利弊。研究结果表明,使用动态特性是程序员完成编程任务的重要组成部分,但是软件维护阶段中暴露了动态特性的负面影响,尤其是在 JavaScript[19]和 Python[20-21]这两种语言中。Park 等[19]发现极具动态特性的 JavaScript 代码会使 Web 应用程序的静态分析变得困难。Wang 等[21]指出,具有动态特性的代码通常容易出错,这一特性也导致 Python 系统不易维护。Chen 等[20]还发现滥用动态特性会在 Python 系统中引入错误。作为动态特性的另一个负面影响,本文展示了 Python 动态类型系统的滥用现象。这些研究有助于编程语言设计者和软件开发者更多维度地认识到相关语言特性的利弊。

鉴于动态类型语言程序中普遍存在类型缺陷,Khan 等[22]通过实证研究验证了类型检查在 Python 项目开发流程中的重要性。一些研究者对此提出了相应的检测方法。例如,Xu 等[7]为 Python 程序开发了一个预测分析引擎,其可以自动生成用于检测类型漏洞的断言,包括子类型断言和属性断言。但目前这类基于静态程序分析的类型漏洞检测技术仍然受限于被分析系统的规模。

除了类型缺陷的检测,对动态类型语言中导致类型缺陷的代码反模式或不良编码实践的检测也受到了研究人员的重视,因为这些检测结果能够提供重要的重构机会。对于 JavaScript,Gong 等[23]针对类型缺陷定义了若干条代码质量规则,并通过动态方法检测出了违反代码质量规则的运行时事件。Pradel 等[10]将 JavaScript 中的强制类型转换分为可能无害和潜在有害,并总结了强制类型转换的不良用法和丑陋用法。与本文工作类似,Pradel 等[11]提醒开发人员 JavaScript 程序中存在类型不一致的变量、属性和函数,但他们采用的是动态分析方法。

除了 JavaScript,Python 中与类型相关的反模式也颇受关注。Mypy[1]是 Python 的一个静态类型检查器,它将程序中的类型注释信息和类型检查相结合,能够检测出违反类型注释的函数调用,但用户需要 Python 3.5 或更高版本才能运行

[1] https://github.com/numpy/numpy/issues/17466

[2] https://github.com/numpy/numpy/issues/16687

[3] https://github.com

[4] https://github.com/ansible/ansible/pull/73319

Mypy，且 Mypy 不会对没有类型注释的动态类型函数进行类型检查。事实上，根据 Rak-amnouykit 等[24]的研究结果，仅有一小部分 Python 项目使用类型注释功能，因此 Mypy 无法报告大多数常规的无类型注释代码中的潜在风险。相比之下，本文旨在设计实现一个 Python 2 和 Python 3 版本均可支持的检测工具，并且可以检测更多种类、更为普遍的高风险编码行为。另一个与本文类似的工作中，Chen 等[9]同样定义和检测了 Python 程序中存在潜在风险的几类动态类型代码，但他们并未对这些高风险代码进行形式化描述，也没有给出具体的检测规则，无法评估其检测工具的有效性和效率。本文期望在他们的研究工作的基础上，深入探索各类高风险动态类型代码的所有模式，总结出形式化检测规则，提出更有效的检测方法，从而高效准确地检测出 Python 软件中具有潜在风险的动态类型代码。

3 软件开发实践中的高风险动态类型编码模式

基于对现有动态类型代码风险研究工作的调研以及对开源 Python 软件中实际类型缺陷的溯源分析，本文总结了 Python 软件开发中的 6 类高风险动态类型编码模式，分别介绍了各类高风险动态类型代码的定义、依据及形式化规则。本文研究的高风险动态类型代码分为两大类：一类是关于类型不一致的不良编码模式，另一类是关于结构不固定的不良编码模式。

3.1 类型不一致

Python 具有的动态类型特性使得变量不需要静态声明，因而变量类型也不再固定不变。在程序运行过程中，任何变量类型的改变都可能导致运行错误，这与变量类型的不固定密切相关。在软件开发过程中，即使变量类型的变更违背了开发人员的意愿，在代码编译时也不会发出警告，因此程序会由于类型错误而导致运行结果不正确。综上，类型不一致是 Python 程序中的一种反模式。Python 中类型不一致的概念的定义如下。

定义 1（类型不一致） 若两个类型 τ_1 和 τ_2 满足下列条件之一，则认为它们类型一致：1）τ_1 和 τ_2 类型相同；2）其中一个类型是另一个类型的子类型；3）τ_1 和 τ_2 结构上等价（即具有相同的属性）。否则，τ_1 和 τ_2 类型不一致。其形式化表示如下：

$$consistent(\tau_1,\tau_2)=\begin{cases}\text{true}, & \text{if } \tau_1=\tau_2 \vee \tau_2\in subtypes(\tau_1)\vee\\ & \tau_1\in subtypes(\tau_2)\vee\\ & structure(\tau_1)=structure(\tau_2)\\ \text{false}, & \text{otherwise}\end{cases}$$

其中，$subtypes(\tau)$ 表示类型 τ 的所有子类集合，$structure(\tau)$ 表示类型 τ 的所有属性结构集合。

在此基础上，可以扩展出一组类型内部的类型不一致和两组类型间的类型不一致的定义。

定义 2（组内类型不一致） 若一组类型集合 T 中存在任意两个元素 τ_1 和 τ_2 满足 τ_1 和 τ_2 类型不一致，则类型集合 T 组内存在类型不一致。其形式化表示如下：

$$INCONT(T)=\begin{cases}\text{true}, & \text{if } \exists\,\tau_1,\tau_2\in T, !consistent(\tau_1,\tau_2)\\ \text{false}, & \text{otherwise}\end{cases}$$

定义 3（组间类型不一致） 若类型集合 T_1 中任意一个元素 τ_1 和类型集合 T_2 中任意一个元素 τ_2 满足 τ_1 和 τ_2 类型不一致，则类型集合 T_1 和类型集合 T_2 两组的类型不一致。其形式化表示如下：

$$INCONT(T_1,T_2)=\begin{cases}\text{true}, & \text{if } \exists\,\tau_1\in T_1,\tau_2\in T_2,\\ & !consistent(\tau_1,\tau_2)\\ \text{false}, & \text{otherwise}\end{cases}$$

基于上述基本定义，下面介绍 Python 程序中的 3 种类型不一致的编码风格。

类型 1（赋值类型不一致） 使用类型不一致的对象重新定义一个变量。

一种情况是，如果开发人员对一个变量进行重新定义，而新的类型与旧的类型不一致，则可能导致开发人员混淆该变量的功能和用法。另一种情况是，如果开发人员对变量类型发生的改变不知情，则会发生误用。ipython 项目[2)]的一个实例如下：

```
module_name={T_API_PYSIDE:'PySide',
             QT_API_PYQT:'PyQt4',
             QT_API_PYQTv1:'PyQt4',
             QT_API_PYQT5:'PyQt5',
             QT_API_PYQT_DEFAULT:'PyQt4'}
  ...
module_name=module_name[api]
```

在最后一条语句中，*module_name* 的类型从字典类型更改为字符串类型，存在类型不一致。该问题是由变量重用造成的。

令 $S=\{s_1,s_2,\cdots,s_m\}$ 为 Python 程序中的所有语句集合，该程序中赋值类型不一致实例集合的形式化规则如下：

$$\begin{aligned}D1=\{(s_i\in S,s_j\in S)\mid\ & s_i\in AssignmentStatement\wedge\\ & s_j\in AssignmentStatement\wedge\\ & scope(s_i)\\ =\ & scope(s_j)\wedge\\ & Def_{s_i}\\ =\ & Def_{s_j}\wedge\\ & INCONT(types(Def_{s_i},s_i),types(Def_{s_j},s_j))\}\end{aligned}$$

其中，$AssignmentStatement$ 表示赋值语句集合，$scope(s)$ 表示语句 s 的作用域，Def_s 表示语句 s 中定义的变量，$types(v,s)$ 表示语句 s 中变量 v 的类型。该规则检查同一个作用域内相同变量的不同赋值语句所接受的类型集合间是否存在组间类型不一致。

类型 2（参数类型不一致） 同一个函数的参数在不同的调用中传入的参数类型不一致。

由于缺少函数参数的类型声明，当开发人员在不同的函数调用中将不一致的类型分配给相同的参数时，就会发生这种反模式。即使函数中的参数类型不可接受，Python 解释器也不会发出任何警告。因此，建议在函数调用中只使用一种类型的参数，以避免意外的类型错误。tornado 项目[1)]的一个实例如下：

1) http://mypy-lang.org/

2) https://github.com/ipython/ipython

```
logging.warning((red.response.http_error.desc,
    vars(red.response.http_error),url))
...
logging.warning("Starting fetch with curl client")
```

在 logging.warning()函数的两次调用中，实参的类型不一致，其中第一次函数调用传入了一个元组，第二次函数调用传入了一个字符串。在这种情况下，logging.warning()函数只能对这个实参执行有限的操作，即打印实参值。任何特定的操作都会引发错误，例如字符的串拼接或元素的更新。

程序 S 中参数类型不一致实例集合的形式化规则如下：

$$D2=\{(s_i\in S,s_j\in S)\mid s_i\in FunctionCallStatement\wedge s_j\in FunctionCallStatement\wedge Fun_{s_i}=Fun_{s_j}\wedge \exists n\in N, INCONT(types(Arg^n_{s_i},s_i),types(Arg^n_{s_j},s_j))\}$$

其中，$FunctionCallStatement$ 表示函数调用语句集合，Fun_s 表示函数调用语句 s 中调用的函数，Arg^n_s 表示语句 s 中函数调用的第 n 个实参。该规则检查同一函数的不同调用中相同位置实参的类型集合间是否存在组间类型不一致。

类型 3(变量类型不一致)　语句中引用的变量具有不一致的类型。

变量在经过不同的执行路径或者在给定不同的输入值时，可能被赋予不同的变量类型，它增加了执行状态的不确定性，从而导致程序不稳定。ansible 项目[2]的一个实例如下：

```
def get_inventory(enterprise,config):
    if cache_available(config):
        inv=get_cache('inventory',config)
    else:
        ...
        inv=generate_inv_from_api(enterprise,config)
        save_cache(inv,config)
```

在这个函数中，参数 *config* 通过解析配置文件获得，因此它的值是不固定的。根据 *config* 的不同值，变量 *inv* 的值可能是在调用 *get_cache*()函数后返回的一个字符串，也可能是在调用 *generate_inv_from_api*()函数后返回的一个字典。因此，最后一条语句中引用的变量 *inv* 会传递这种不一致的类型到 *save_cache*()函数内部，进而影响代码执行结果。

程序 S 中变量类型不一致实例集合的形式化规则如下：

$$D3=\{s_i\in S\mid \exists v\in Ref_{s_i}, INCONT(types(v,s_i))\}$$

其中，Ref_s 表示语句 s 中使用的变量集合。该规则检查语句中所使用的某个变量的所有类型集合是否存在组内类型不一致。

值得注意的是，以上 3 种类型代码之间不是完全独立的。例如，在某些函数中，变量类型不一致与参数类型不一致有关，因为不同类型的参数值会导致同一函数中某些变量的类型不一致。但是在大多数情况下，在不同的场景中会出现 3 种不同的类型不一致问题。

3.2 结构不固定

由于 Python 程序中创建的对象结构是在运行时决定的，因此在使用动态类型时的另一种高风险实践是动态更改对象的属性结构，特别是容器对象和类/实例对象，对象的结构不固定会使引用它的所有变量受到影响。此类实践与对象的类型高度相关，因为只有可变类型支持动态更改结构，且更改一个类的结构也会影响所有实例对象的结构。另外 3 种关于结构不固定的高风险动态类型编码模式如下。

类型 4(动态删除元素)　通过动态确定的索引值从容器中删除对应元素。

在 Python 类型系统中，容器由 4 种类型组成：列表、元组、字典和集合。其中，列表、字典和集合在 Python 中是可变类型，它们的对象的结构可以动态更改，包括添加元素和删除元素。这些动态行为可能在程序运行时的任何时刻发生，意外地和未确定地动态删除元素都将导致元素访问失败。scikit-learn 项目[3]中的一个实例如下：

```
for k,v inlist(six.iteritems(namecount)):
    if v==1:
        del namecount[k]
```

这些语句使用 *del* 函数从字典类型变量 *namecount* 中删除了一些键值对，且被删除的元素是由一个变量 k 索引的，而不是一个常数。如果元素在删除之后被访问，则会抛出异常。此外，从列表对象中删除元素时，会使得开发人员在访问剩余元素时混淆新旧索引。

程序 S 中动态删除元素实例集合的形式化规则如下：

$$D4=\{s_i\in S\mid s_i\in ElementDeletionStatement\wedge types(Con_{s_i},s_i)\subseteq\{list,dict,set\}\wedge !isConstant(Idx_{s_i})\}$$

其中，$ElementDeletionStatement$ 是元素删除语句集合；Con_s 表示语句 s 中被删除元素的容器对象；Idx_s 表示语句 s 删除的元素索引或者键；$isConstant(e)$ 为一个逻辑值，用于判断实体 e 是否为不变值。该规则检查元素删除语句中索引值是否是一个固定值且容器对象的类型是列表、字典或集合。

类型 5(动态删除属性)　通过动态确定的属性名从对象中删除一个属性。

在 Python 中，所有对象都是第一类对象，无论什么类型都可以动态创建、销毁、作为参数传递给函数并作为值返回，因此对象的生命周期可以跨越不同的函数作用域而存在。然而，任何属性都可以从任何对象中动态删除，因此对象的结构是不可控的。事实上即使是未使用的属性也并不建议动态删除。django 项目中[4]的一个实例如下：

```
delattr(obj.__class__,self.name)
```

由于 Python2 和 Python3 之间的版本差异，*obj* 类的属性列表存在细微的差异。为了支持程序在不同的 Python 平台上运行，该实例从 *obj* 中删除了未使用的属性(即以 *self.name* 的值作为属性名的属性)以隐藏版本间的差异。但是如果动态删除操作之后该属性仍然被访问，则会抛出 Attribu-

[3] https://github.com/tornadoweb/tornado

[4] https://github.com/ansible/ansible

[3] https://github.com/scikit-learn/scikit-learn

[4] https://github.com/django/django

teError。值得注意的是，从类对象中删除属性会使该属性对其所有实例不再可用。

程序 S 中动态删除属性实例集合的形式化规则如下：

$$D5=\{s_i\in S\mid s_i\in AttributeDeletionStatement\wedge types(Attr_{s_i},s_i)=\{string\}\wedge !isConstant(Attr_{s_i})\}$$

其中，$AttributeDeletionStatement$ 是属性删除语句集合，$Attr_s$ 表示语句 s 访问的属性。该规则检查属性删除语句中的属性名字是否是一个字符串变量。

类型 6(动态访问属性)　通过动态确定的名称访问一个属性。

由于对象的属性列表在运行时是可变的，因此以不确定的行为访问属性是有风险的。如果访问失败，则会引发 AttributeError 异常。ansible 项目中的一个实例如下：

```
dep_value=getattr(dep,attr)
```

该语句从 dep 中获得一个属性值，而属性名 $attr$ 的值未确定。对此，建议在获取属性值之前检查其是否存在或主动捕获异常。

程序 S 中动态访问属性实例集合的形式化规则如下：

$$D6=\{s_i\in S\mid s_i\in AttributeAccessStatement\wedge types(Attr_{s_i},s_i)=\{string\}\wedge !isConstant(Attr_{s_i})\}$$

其中，$AttributeAccessStatement$ 是属性访问语句集合。该规则检查属性访问语句中的属性名字是否是一个字符串变量。

本文研究上述 6 种动态类型相关的不良编码实践。根据初步研究[9]，这些编程实践不一定会直接导致程序中的错误，但会增加出现错误的风险。这并不意味着不允许它们出现在程序中，而是应对其进行限制，以免发生程序错误。值得注意的是，一些其他语言也支持部分动态类型特性，也会存在上述某几类高风险动态类型代码[10-11]，因此本研究有助于众多开发人员和研究人员开展工作。

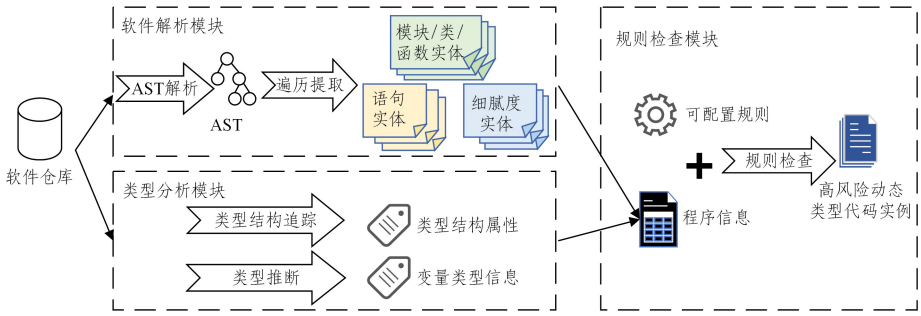

图 1　基于规则的高风险动态类型代码检测技术的框架

Fig. 1　Architecture of rule-based detection technique

4　基于规则的高风险动态类型代码检测技术

为了检测第 3 节描述的高风险动态类型代码，本文提出了一种基于规则的静态检测技术。图 1 给出了该检测技术的总体框架，主要包含 3 个功能模块：软件解析模块、类型分析模块和规则检查模块。下面分别介绍这 3 个功能模块的实现过程。

4.1　软件解析模块

为了抽取必要的上下文信息，该模块将 Python 软件的源代码解析为抽象语法树(Syntax Tree，AST)的形式。AST 是编译器前端阶段通常使用的内部数据结构，它对程序的结构、声明、变量使用、函数调用等进行编码，是程序的无损表示。更重要的是，AST 还使用类型信息来进行修饰，并包含声明(类型、函数、变量)及其使用之间的连接。

本模块使用 Python 中的内置函数 $parse()$ 和 $compile()$ 生成所有 Python 文件的 AST 集合。基于 Python 代码的 AST 集合，本模块在不同的粒度级别上完成以下任务。

(1)模块、类和函数级别：根据生成的 AST，软件解析模块收集每个 Python 模块节点、类节点和函数节点的所有子树，据此得到每个作用域(即模块作用域、类作用域或函数作用域)中的语句列表，最后逆向确定每个语句所属的作用域。

(2)语句级别：在这个级别上，每个语句(如赋值语句或条件语句)的类型都由其 AST 子树的结构确定。本功能模块从解析的 AST 结构中提取 5 种类型的语句实体，包括赋值语句、函数调用语句、元素删除语句、属性删除语句和属性访问语句。需要注意的是，一个语句可以同时属于多个类型，例如，将某个函数的返回值赋给一个变量的语句被拆分为赋值语句和函数调用语句。

(3)细粒度级别：对于每个语句节点，软件解析模块通过遍历它的子树来分析操作符和操作数。例如，AST 叶子节点包含每个语句中使用的变量和常量，通过这种方式可以识别语句的关键元素。此外，本模块将每个赋值语句 s 中定义的变量表示为Def_s，将每个语句 s 中使用的变量表示为Ref_s，将每个函数调用语句 s 中的函数和第 i 个参数分别表示为Fun_s和Arg_s^i，将每个元素删除语句 s 中的容器和元素索引表示为Con_s和Idx_s，将每个属性删除语句 s 或属性访问语句 s 中所访问的属性表示为$Attr_s$。

4.2　类型分析模块

该模块主要收集目标程序中的类型信息，它有两个主要任务：类型推断和类型结构跟踪。

首先完成类型推断任务。Python 程序中类型系统包括基本类型(如整型和字符串)、容器类型(如列表和元组)、框架类型(如类和函数)和其他用户定义类型(如用户定义的类)。在没有类型声明的情况下，由于输入的不同，在某个程序点上的变量可能有多个类型，而检测高风险动态类型代码的关键是要能够在目标程序的任一位置上推断每个变量的类型。

本模块优先使用 Pythonar2[1] 来推断目标程序中所有变量的类型，Pythonar2 是最流行的 Python 类型推断工具之一[20]。与完全静态的类型推断工具 Pythonar2 不同，PyAnnotate[2] 和 MonkeyType[3] 是两个动态工具，它们可以根据程序运行时监测到的参数和返回值的类型自动生成类型注释。但是当需要监测大量代码的执行时，此过程将非常耗时，而且它们只生成参数和返回值的类型信息。除此之外，如果语句未被执行，那么这些动态工具将无法为其生成类型信息，而且程序运行时无法涵盖语句所有可能的计算结果。因此，这类动态工具不适用于高风险动态类型代码的检测，故本模块选择使用 Pythonar2 来推断每个变量的可能类型。我们将语句 s 中变量 v 的所有可能类型表示为 $types(v,s)$，如果它只有一个类型则记为 $\{t\}$；如果它有多个类型则记为 $\{t1|t2|\cdots\}$。

接下来完成类型结构跟踪任务。在 Python 中，类型本质上是一个定义好的类，它具有一组属性。同时，两个类型（如超类型和子类型）之间的关系本质上是两个定义类（如基类和派生类）之间的关系。类属性（即字段和方法）的设计表明了类型的结构。为了获取每种类型的结构特征，本模块通过遍历程序来跟踪每个类及其属性的定义。算法 1 给出了在类型分析模块中类型结构跟踪的功能实现。

首先，跟踪所有类的定义语句并获取每个类的基类。对于程序中的每个类型 τ，它的子类型表示为 $subtypes(\tau)$，由其直接派生类和每个派生类的所有子类型组成。这个过程如算法 1 中的第 1－10 行所示。

随后，以 *obj.attr* 形式收集程序中的所有属性访问操作来获取类型结构。其中 *obj* 是一个实例对象，*attr* 是属性名。根据类型推断的结果，如果 *obj* 的类型为 τ，则将 *attr* 加入到 τ 的属性集中。由于继承机制，τ 的每个子类型也具有此属性。这个过程如算法 1 中的第 11－17 行所示。

最后，对于程序中的每个类型 τ，返回 τ 的直接和间接派生类的集合 $subtypes(\tau)$ 作为其子类型，返回 τ 的属性的集合 $structures(\tau)$ 作为其结构。

算法 1 类型结构跟踪算法

输入：Python 程序代码

输出：每一个类型的所有子类型集合和属性集合

```
1. T={}
2. for each class definition statement s in the program do
3.       τ=the class name defined in s
4.       T=T∪τ
5.       subtypes(τ)={}
6.       baseclasses(τ)={base classes declared in s}
7. sort T to ensure that each derived class ranks before its base classes
8. for each type τ in T do
9.    for each base class τb in baseclasses(τ) do
10.          subtypes(τb)=subtypes(τb)∪τ ∪subtypes(τ)
11. for each type τ in T do
12.          structure(τ)={}
13. for each attribute access operation in the form of obj. attr in the
    program do
14.       τ=the type of obj according to type inference
15.       structure(τ)=structure(τ)∪attr
16.       for each subtype τs in subtypes(τ) do
17.             structure(τs)=structure(τs)∪attr
18. return subtypes, structure
```

4.3 规则检查模块

在分析目标软件中的 AST 实体和类型信息后，该模块根据规则检测 6 种类型的高风险动态类型代码。注意，该模块的规则是可配置的，利于后续扩展更多高风险动态类型代码规则。

根据类型推断和结构跟踪的结果，使用本文第 3 节描述的形式化方式可以识别类型不一致现象。具体来说，判断一组类型内是否类型不一致（即 $INCONT(T)$）和判断两组类型间是否类型不一致（即 $INCONT(T_1,T_2)$）的实现过程如下。

（1）标准化：对于每个类型集合，如果其中的元素有多个类型（如 $\{list|tuple|string\}$），则将其拆分为单个类型的元素，然后从集合中删除重复的元素。例如，原始类型集合 $\{string, \{list|function|string\}, int\}$ 将被转换为 $\{string, list, function, int\}$。

（2）计算 $INCONT(T)$：如果集合 T 中的任意两个元素的所属种类是完全不同的（例如字符串和函数分别是基本类型和框架类型），则返回 $INCONT(T)$ 的结果为 true。此外，如果任意两个元素不相同或结构上不等价，或者其中一个元素不是另一个元素的结构子类型，则结果也将返回 true。除上述情况以外，$INCONT(T)$ 返回 false。

（3）计算 $INCONT(T_1,T_2)$：与检查 $INCONT(T)$ 类似，如果 T_1 中的任一元素的类型和 T_2 中的任一元素的类型不一致，则 $INCONT(T_1,T_2)$ 的返回结果为 true；否则，返回 false。

根据不同粒度级别的 AST 实体信息，结合类型分析模块的输出结果和类型不一致的计算结果，使用本文第 3 节描述的规则可以检测出 6 种类型的高风险动态类型代码。具体来说，通过遍历软件解析模块提取的所有语句实体，并检查各语句是否满足高风险动态类型代码检测规则，最后从软件项目中识别出所有高风险动态类型代码实例报告给用户。

5 实验设计

为了评估本文检测方法的优势和不足，并且探索高风险动态类型代码在 Python 软件中的分布特点，本文针对开源 Python 软件项目展开实验评估与分析。实验运行环境为 64 位 Windows 系统，1.61GHz 的 Intel(R) Core(TM) i7 CPU，16GB 物理内存，编译环境覆盖 Python 2 和 Python3。

实验对象是来自 GitHub 平台上的 25 个 Python 项目，这些项目可以在 GitHub Archive 中[4] 检索获取，平台承载了项目所有公共活动的数据库，整个数据库可以通过公开的 API[5] 访问。表 1 列出了所有实验对象项目的信息。

1) https://github.com/yinwang0/pysonar2

2) https://github.com/dropbox/pyannotate

3) https://github.com/Instagram/MonkeyType

4) https://www.gharchive.org/

5) https://ghdocs-prod.azurewebsites.net/en

表 1 实验对象项目

Table 1 Subject projects

Project	# Stars	Creation Data	# Commits	Analyzed Release	# Files	# Method	LOC
requests	47 100	2011-02	6 106	v2.22.0	35	608	9 738
fabric	13 300	2009-05	1 511	2.5.0	46	537	7 238
tornado	20 500	2009-09	4 404	v5.0.2	122	3 434	45 253
ansible	52 600	2012-03	52 471	v2.2.0.0-1	504	3 587	85 593
ipython	15 300	2010-05	25 929	5.1.0	351	3 113	66 737
beets	10 800	2010-08	10 468	v1.4.9	166	4 087	58 626
flask	58 400	2010-04	4 584	1.1.1	73	1 343	16 965
transformers	60 100	2018-10	9 389	v0.6.1	36	463	12 143
scrapy	43 200	2010-02	9 120	1.7.0	292	3 128	38 046
locust	18 500	2011-02	3 465	0.11.1	43	427	5 550
magenta	17 500	2016-05	1 401	2.1.0	330	2 619	71 951
ray	19 700	2016-10	11 885	ray-0.6.0	361	3 441	63 116
poetry	19 000	2018-02	2 081	1.0.0b1	331	1 863	36 091
jumpserver	18 200	2014-07	7 076	1.5.2	474	1 664	29 106
bokeh	16 100	2012-03	19 520	1.1.0	1 021	5 028	117 704
pyspider	15 400	2014-02	1 181	v0.3.10	103	1 032	14 324
streamlit	18 400	2019-08	4 633	0.44.0	222	896	20 837
dash	16 100	2015-04	5 787	v1.3.1	69	563	12 348
luigi	15 500	2012-09	4 029	2.8.9	253	4 837	61 530
cookiecutter	16 500	2013-07	2 763	1.7.0	82	379	7 649
labelImg	16 900	2015-09	427	v1.8.3	24	253	3 603
kivy	14 500	2010-11	12 649	1.11.1	426	4 070	96 271
InstaPy	14 000	2016-09	3 332	0.6.4	33	312	17 395
jax	16 900	2018-10	10 900	jaxlib-v0.1.32	110	3 228	41 879
prophet	14 200	2016-11	706	v0.5	12	130	5 336

注：数据取自 2022 年 8 月。

对于每个项目，表 1 列出了项目名称（第 1 列）、用户收藏数（第 2 列）、创建日期（第 3 列）、自创建以来的提交次数（第 4 列）、本实验分析版本（第 5 列）、该版本中 Python 文件数目（第 6 列）、函数数目（第 7 列）和代码行数（第 8 列）。这些项目的所属领域、创建时间和开发团队规模都有所不同。其中，项目收藏数为 1.08 万到 6.01 万不等，代码规模为 0.3 万行到 11.7 万行不等。这些研究对象的筛选方法如下：首先收集 GitHub 上收藏数量排名前 100 的 Python 项目，然后丢弃规模相对较小的项目（如短于 1 000 行代码）或开发时间较短的项目（如短于 3 年），最后丢弃无法成功编译的项目（主要是因为 Python 各版本间的语法差异和语义变化），最终经过筛选之后选取 25 个可用于研究的开源软件项目。

基于上述实验设置和实验对象，本文针对以下 3 个研究问题展开实验研究。

5.1 RQ1：基于规则的高风险动态类型代码检测技术的有效性如何？

该研究问题主要关注本文提出的基于规则的检测技术能否在实际 Python 项目中有效地检测出高风险动态类型代码实例。

对此，实验以 Python 函数为样本，衡量样本内检测出的高风险动态类型代码实例的正确性。实验参考 Krejcie 和 Morgan 的做法[25]，在随机抽取样本进行评估时，设定抽样置信度为 95%，置信区间为 5%。具体而言，本实验研究对象中共有 51 042 个 Python 函数，需要 382 个样本才能达到期望的置信度和置信区间，为了方便起见把样本量四舍五入到 400 个。由于很多高风险动态类型代码实例集中在一小部分 Python 函数中，随机选择 400 个 Python 函数很可能导致其中绝大部分样本不存在任何高风险动态类型代码实例，因此本实验样本的选取分为两个部分。其中一部分样本是从检测出高风险动态类型代码实例的 Python 函数中随机选择 200 个函数，其中每类高风险动态类型代码大约为 33 个样本。另一部分样本是从未检测出高风险动态类型代码实例的 Python 函数中随机选择 200 个函数。对于每个抽样到的 Python 函数，结合动态分析方法和静态分析方法，根据 6 类高风险动态类型代码的定义来确定函数中所有语句是否存在高风险动态类型代码。其中动态分析方法主要依靠软件中的测试用例来触发函数执行，动态监测函数中的语句是否存在高风险动态类型代码实例。而静态分析方法主要通过检查 API 文档来了解函数的功能和用法，然后静态审查代码找出高风险动态类型代码实例。最后，将从抽样的 400 个函数中人工找到的高风险动态类型代码实例作为 RQ1 的基准数据集。基于这个基准数据集，使用 3 个指标来评估本文提出的基于规则的检测技术的有效性，即 *precision*（准确率）、*recall*（召回率）和 *F-measure*（F 值），其计算式如下：

$$precision=\frac{|Instances_{Groundtruth}\cap Instances_{Tool}|}{Instances_{Tool}}$$

$$recall=\frac{|Instances_{Groundtruth}\cap Instances_{Tool}|}{Instances_{Tool}}$$

$$F\text{-}measure=2\cdot\frac{precision\cdot recall}{precision+recall}$$

其中，$Instances_{Groundtruth}$ 是由基准数据集中人工标注的实例组成的集合，$Instances_{Tool}$ 是由本文检测框架报告组成的实例集合。

5.2 RQ2：基于规则的高风险动态类型代码检测技术的效率如何？

本研究问题关注基于规则的检测技术能否在实际 Python 项目中高效地检测出高风险动态类型代码实例。

由于实验选取的 25 个软件项目代码规模为 0.3 万行到 11.7 万行不等，因此对在这些软件项目上的检测效率进行统计分析，足以体现基于规则的检测技术在不同规模的软件项目中的效率。实验记录在每个软件项目中基于规则的检测技术在软件解析模块、类型分析模块和规则检查模块所花费的

时间，进而分析性能瓶颈。

5.3 RQ3：各类高风险动态类型代码在 Python 软件中是如何分布的？

RQ3 关注哪些种类的高风险动态类型代码在 Python 软件项目中更为普遍，以及它们具体是如何分布于软件代码内部的。

实验包含了各类高风险动态类型代码的普遍性衡量以及分布规律研究两个部分。对于普遍性衡量，基于在 25 个 Python 软件项目中的高风险动态类型代码检测结果，统计每种类型的高风险动态类型代码在项目中的出现次数，然后通过横向比较不同类型的实例数目来比较它们的普遍性。对于分布规律的研究，则是通过统计各 Python 函数中高风险动态类型代码实例的分布，来衡量高风险动态类型代码在软件代码中的聚集性程度。

6 实验结果

基于表 1 中的 25 个开源软件项目，本节报告了 3 个研究问题的研究结果。

6.1 RQ1：基于规则的高风险动态类型代码检测的有效性

表 2 列出了 400 个抽样函数中高风险动态类型代码的检测结果。表 2 中第二行表示通过人工审查在样本中找到的高风险动态类型代码实例数目（即基准数据集），第三行表示本文提出的检测框架在样本中检测到的高风险动态类型代码实例数目，这两行数据中括号中的数字表示包含高风险动态类型代码的函数个数。其余行分别表示基于规则的检测结果产生的假阳性（FP）、假阴性（FN）、准确率（*precision*）、召回率（*recall*）和 F 值（即 *F-measure*）。

表 2 本文检测方法在 400 个抽样 Python 函数上的有效性

Table2 Effectiveness of the proposed detection method in 400 sampled Python functions

Metric	赋值类型不一致	参数类型不一致	变量类型不一致	动态删除元素	动态删除属性	动态访问属性	所有类型
$\#Instances_{Groundtruth}$	47(35)	46(33)	89(36)	56(37)	48(36)	71(45)	357(196)
$\#Instances_{Tool}$	49(37)	49(36)	88(36)	56(37)	48(36)	71(45)	361(200)
FP	2	3	4	0	0	0	9
FN	0	0	5	0	0	0	5
precision	0.95	0.93	0.95	1.0	1.0	1.0	0.97
recall	1.0	1.0	0.94	1.0	1.0	1.0	0.98
F-measure	0.97	0.96	0.94	1.0	1.0	1.0	0.98

基于规则的检测技术在实验中暴露出 9 个假阳性，包括 2 个赋值类型不一致实例、3 个参数类型不一致实例和 4 个变量类型不一致实例。由于个别变量和参数的类型在特定上下文环境中难以推断，类型推断结果产生了偏差，因此导致了赋值类型不一致和参数类型不一致这两类结果的假阳性。此外，尽管通过类型推断可以判断出某个变量存在不一致类型，但在实际应用场景中这些不一致类型不一定会在运行时出现，这导致本文的检测技术在变量类型不一致的检测中存在 4 个假阳性。

此外，基于规则的检测技术在实验中还出现了 5 个假阴性，均为变量类型不一致的实例。通过结果溯源，发现这是由于检测过程中在 AST 解析时相关变量被遗漏所导致的。通过增强 AST 解析能力、覆盖更全面的 Python 语法，可以解决此类不足。在动态删除元素、动态删除属性和动态访问属性的检测结果中，并未发现假阳性或假阴性。这是因为这几类高风险动态类型代码的检测不依赖于类型推断，而是依赖于 AST 结构的解析，而 AST 结构解析结果通常是相对准确的。

6.2 RQ2：基于规则的高风险动态类型代码检测的效率

表 3 列出了不同规模的软件项目中本文检测框架在各步骤上的时间统计，可以十分直观地看出一个普遍的规律：代码行越长的软件，检测高风险动态类型代码所用的时间就越长。

从表 3 中最后一列数据可以看出，每个软件项目的平均检测时间约为 155s，即 2.5min 左右，这样的检测效率相比人工审查要快得多。当然，随着代码库规模的增大和检测规则的扩展，检测时间也会随之增长。通过比较软件解析、类型分析和规则检查这 3 个模块的执行时间可以得到，高风险动态类型代码的检测时间主要受限于类型分析模块，而软件解析模块和规则检查模块仅需执行数秒或数十秒。由此可以推断，提升高风险动态类型代码检测效率的首要任务是优化类型分析算法。

表 3 本文检测技术在各模块的执行时间统计

Table 3 Execution time statistics of the proposed detection technology in each module

Project	LOC	软件解析/s	类型分析/s	规则检查/s	总时间/s
requests	9738	10.42	52.28	5.21	67.91
fabric	7238	8.23	40.28	3.78	52.29
tornado	45253	44.55	124.32	2.39	171.26
ansible	85593	47.85	230.21	5.39	283.45
ipython	66737	30.23	190.91	6.48	227.62
beets	58626	23.56	170.12	3.41	197.09
flask	16965	12.09	89.56	2.97	104.62
transformers	12143	10.47	88.69	3.56	102.72
scrapy	38046	20.73	120.87	3.35	144.95
locust	5550	7.91	38.97	2.99	49.87
magenta	71951	31.89	209.73	5.98	247.60
ray	63116	29.09	199.81	4.25	233.15
poetry	36091	20.16	127.92	4.59	152.67
jumpserver	29106	18.37	105.71	3.29	127.37
bokeh	117704	40.78	356.89	7.11	404.78
pyspider	14324	7.91	91.82	2.99	102.72
streamlit	20837	10.59	98.55	5.23	114.37
dash	12348	9.71	78.98	4.36	93.05
luigi	61530	29.63	193.82	3.44	226.89
cookiecutter	7649	3.95	38.59	3.23	45.77
labelImg	3603	3.19	39.69	2.98	45.86
kivy	96271	38.72	319.31	5.43	363.46
InstaPy	17395	11.22	105.99	3.92	121.13
jax	41879	19.73	132.22	4.28	156.23
prophet	5336	5.87	37.29	3.29	46.45
平均	37801	19.87	131.30	4.16	155.33

6.3 RQ3：高风险动态类型代码在 Python 软件中的分布

表 4 列出了 25 个 Python 软件中各类高风险动态类型代码的数目，括号中的数字表示发现具有高风险动态类型代码的函数个数。结果发现，所有这些开源软件项目的 Python

代码中均至少存在一种高风险动态类型代码，平均每个项目中存在153个高风险动态类型代码实例，涉及到81个Python函数。其中有6个项目（占24%）中6类高风险动态类型代码全部存在，剩余19个项目中有9个项目存在5种类型的高风险动态类型代码。这个结果揭示了高风险动态类型代码在Python编程中普遍存在。

表4 高风险动态类型代码在软件项目中出现的数目

Table 4 Occurrences of risky dynamic typing code in subject projects

Project	赋值类型不一致	参数类型不一致	变量类型不一致	动态删除元素	动态删除属性	动态访问属性	所有类型
requests	4(4)	2(1)	18(11)	11(11)	0(0)	17(12)	52(36)
fabric	1(1)	1(1)	8(2)	1(1)	0(0)	5(4)	16(9)
tornado	10(3)	9(7)	36(24)	30(28)	3(3)	45(41)	133(103)
ansible	39(28)	5(4)	128(50)	77(54)	1(1)	157(110)	407(234)
ipython	3(3)	4(4)	216(87)	50(43)	5(5)	99(68)	377(195)
beets	14(11)	2(2)	98(45)	42(34)	8(8)	36(29)	200(124)
flask	3(3)	1(1)	6(2)	2(2)	0(0)	24(20)	36(28)
transformers	12(6)	0(0)	49(11)	0(0)	0(0)	18(7)	79(24)
scrapy	15(14)	4(3)	83(33)	17(17)	6(6)	73(61)	198(130)
locust	2(1)	0(0)	6(5)	8(4)	0(0)	3(3)	19(11)
magenta	40(14)	1(1)	165(67)	35(26)	0(0)	12(11)	253(114)
ray	6(5)	0(0)	99(27)	28(25)	1(1)	16(15)	150(72)
poetry	5(3)	4(4)	137(51)	27(23)	0(0)	12(11)	185(88)
jumpserver	9(9)	0(0)	23(13)	2(2)	0(0)	43(36)	77(58)
bokeh	3(3)	2(2)	165(69)	54(37)	10(5)	108(79)	342(186)
pyspider	7(7)	0(0)	92(37)	36(22)	12(8)	29(26)	176(94)
streamlit	1(1)	1(1)	20(8)	17(14)	0(0)	36(21)	75(43)
dash	4(3)	0(0)	32(9)	7(6)	0(0)	38(27)	81(39)
luigi	0(0)	1(1)	235(88)	10(9)	0(0)	63(50)	309(146)
cookiecutter	0(0)	0(0)	9(4)	1(1)	0(0)	0(0)	10(5)
labelImg	0(0)	0(0)	0(0)	2(1)	0(0)	0(0)	2(1)
kivy	14(14)	2(2)	0(0)	70(54)	14(9)	77(59)	177(131)
InstaPy	2(2)	8(1)	82(14)	1(1)	0(0)	0(0)	93(18)
jax	2(2)	0(0)	281(74)	0(0)	0(0)	58(40)	341(116)
prophet	0(0)	1(1)	31(12)	0(0)	0(0)	6(3)	38(16)
平均	8(5)	2(1)	81(31)	21(17)	2(2)	39(29)	153(81)

在选取的这些软件中，高风险动态类型代码实例的数量为2个（labelImg项目）到407个（ansible项目）不等。25个项目中有13个项目（52%）存在100个以上的高风险动态类型代码实例，大多数高风险动态类型代码的种类是变量类型不一致。作为最常见的类型，变量类型不一致几乎出现在所有的软件项目中，平均每个项目有81个实例，涉及30个Python函数。通过审查部分变量类型不一致的实例，发现这些实例通常是紧密相关的，因为一个变量的类型不一致会沿着执行路径传递给其他变量。这意味着重构变量类型不一致这类代码问题的关键是要找到将类型不一致传播到其他变量的根源变量。此外，通过进一步审查其中的不一致类型集合，发现类型不一致往往和字符串相关。通常情况下，这些变量的值是从用户输入值中解析出来的，其类型由输入源（如XML文件、数据库或命令行）决定。此外还有一些其他常见的类型不一致的组合，例如，由算术运算结果导致的int与float的类型不一致组合；由于0/1和True/False的混用导致的int与bool的类型不一致组合。

并不是在所有的软件项目中变量类型不一致都是最常见的。例如，在ansible项目中最常见的高风险动态类型代码是动态访问属性，存在157个实例。类似地，kivy存在70个动态删除元素的实例和77个动态访问属性的实例，占所有高风险动态类型代码实例总数的83%。

相比之下，参数类型不一致和动态删除属性在Python软件代码中很少出现，平均每个项目中只有2个实例。类型注释是Python3版本中的一项新功能，它允许开发人员在声明函数时注释所需的参数类型，这减少了API调用的错误使用，并进一步减少了参数类型不一致的出现。所研究的25个项目中，只有9个项目存在动态删除属性的实例。我们发现对于一些过时无用的属性，即使开发人员不再使用，他们也会选择将这些属性保留下来。

对于高风险动态类型代码的分布规律，通过实验得到一个重要的发现：一个Python函数内可能聚集多个高风险动态类型代码。例如，jax项目中检测到了341个高风险动态类型代码实例，覆盖了116个Python函数。为了便于理解，本文统计了所有存在高风险动态类型代码的函数中的实例数目。图2展示了包含高风险动态类型代码的函数中实例数目的分布情况。

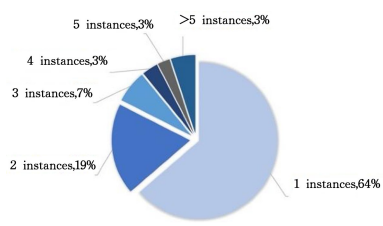

图2 含有高风险动态类型代码的函数中实例数目的分布情况

Fig. 2 Distribution of the number of instance in a function containing risky dynamic typing code

结果显示，在含有高风险动态类型代码的函数中，有36%的函数包含多个实例，其中有5%的函数包含5个以上的实例。我们审查了存在20个以上高风险动态类型代码实例的函数（即magenta项目中的*cmd_to_vector*函数、luigi项目中的*run_job*函数和InstaPy项目中的*unfollow*函数），这些函数中检测出的所有实例类型都是变量类型不一致，实例的聚集性源于不一致类型在相互依赖的变量间的传递。

7 有效性分析

本节从内部有效性和外部有效性这两个方面进行讨论。

本文方法的内部有效性所受的影响主要来自效率实验的随机性，即每次运行检测工具都有可能获得不同的执行时间结果。本文对于每个研究对象上的检测效率实验任务均运行3次，取结果的平均值，这样可以在一定程度上降低这种随机性的影响。

外部有效性所受的影响主要在于所研究的高风险动态类型代码类型的选取和实验对象的选取两方面。本文研究的6种高风险动态类型代码均在之前的研究中被证实与类型缺陷显著相关，具有一定的代表性。此外，本文实验所选取的25个软件项目涵盖了从中型到大型的软件规模，数据集具有一定的代表性。在后续研究中，将扩大实证研究的规模，在更多视角检测更多类型的高风险动态类型代码。

结束语 动态类型特性的滥用对软件质量构成了严重威胁。本文对Python软件中的6类高风险动态类型代码展开研究，对其总结出形式化规则，并提出了一种基于规则的检测技术。此外，本文在25个不同规模的开源软件项目中展开评估实验。实验结果表明：1）在抽样的400个Python函数中，本文基于规则的检测技术达到97%的准确率和98%的召回率；2）检测一个软件项目平均需要花费2.5min，其中类型分析任务占据了大部分检测时间；3）平均每个软件项目存在153个高风险动态类型代码实例，且一个函数内可能聚集多个实例，其中最常见的种类是变量类型不一致。

总体而言，本文研究发现，基于规则的高风险动态类型代码检测技术在实际软件中的检测效果较好，且实验暴露出高风险动态类型代码在Python软件中普遍存在。未来计划研究不同类型的高风险动态类型代码之间的内部联系，并尝试开发一个有效的IDE插件工具用于修复高风险动态类型代码产生的缺陷。

参 考 文 献

[1] KLEINSCHMAGER S, ROBBES R, STEFIK A, et al. Do static type systemsimprove the maintainability of software systems? An empirical study[C]// Proceedings of the 20th IEEE International Conference on Program Comprehension. 2012: 153-162.

[2] ZHANG X F, ZHU C. Empirical study of code smell impact on software evolution[J]. Journal of Software, 2019, 30(5): 1422-1437.

[3] MAYER C, HANENBERG S, ROBBES R, et al. An empirical study of the influence of static type systems on the usability of undocumented software[J]. ACMSIGPLAN Notices, 2012, 47(10): 683-702.

[4] GAO Z, BIRD C, BARR E T. To type or not to type: Quantifying detectable bugs in JavaScript[C]// Proceedings of the 39th International Conference on Software Engineering. IEEE, 2017: 758-769.

[5] MEYEROVICH L A, RABKIN A S. Empirical analysis of programming language adoption[C]// Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. 2013: 1-18.

[6] RAY B, POSNETT D, FILKOV V, et al. A large scale study of programming languages and code quality in github[C]// Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York, NY, USA: Association for Computing Machinery, 2014: 155-165.

[7] XU Z G, LIU P, ZHANG X Y, et al. Python predictive analysis for bug detection[C]// Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York, NY, USA: Association for Computing Machinery, 2016: 121-132.

[8] TIAN Y C, LI K J, WANG T M, et al. Survey on code smells [J]. Journal of Software, 2023, 34(1): 150-170.

[9] CHEN Z F, LI Y H, CHEN B H, et al. An empirical study on dynamic typing related practices in Python systems[C]// Proceedings of the 28th International Conference on Program Comprehension. New York, NY, USA: Association for Computing Machinery, 2020: 83-93.

[10] PRADEL M, SEN K. The good, the bad, and the ugly: An empirical study of implicit type conversions in JavaScript[C]// Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP). Dagstuhl, Germany: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015: 519-541.

[11] PRADEL M, SCHUH P, SEN K. TypeDevil: Dynamic type inconsistency analysis for JavaScript[C]// Proceedings of the 37th IEEE International Conference on Software Engineering. 2015: 314-324.

[12] CALLAÚ O, ROBBES R, TANTER É, et al. How (and why) developers use the dynamic features of programming languages: The case of Smalltalk[J]. Empirical Software Engineering, 2013, 18(6): 1156-1194.

[13] RICHARDS G, LEBRESNE S, BURG B, et al. An analysis of the dynamic behavior of JavaScript programs[C]// Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA, Association for Computing Machinery, 2010: 1-12.

[14] DUFOUR B, GOARD C, HENDREN L, et al. Measuring the dynamic behaviour of AspectJ programs[C]// Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. New York, NY, USA, Association for Computing Machinery, 2004: 150-169.

[15] HOLKNER A，HARLAND J. Evaluating the dynamic behaviour of Python applications[C]// Proceedings of the Thirty-Second Australasian Conference on Computer Science-Volume 91. AUS，Australian ComputerSociety，Inc. ，2009：19-28.

[16] ÅKERBLOM B，STENDAHL J，TUMLIN M，et al. Tracing dynamic features inPython programs[C]// Proceedings of the 11th Working Conference on Mining Software Repositories. New York，NY，USA，Association for Computing Machinery，2014：292-295.

[17] PENG Y，ZHANG Y，HU M Z. An Empirical study for common language features used in Python projects[C]// Proceedings of IEEE International Conference on Software Analysis，Evolution and Reengineering(SANER). 2021：24-35.

[18] JIANG C M，HUA B J，FAN Q L，et al. Empirical security study of native code in Python virtual machines[J]. Computer Science，2022，49(6A)：474-479.

[19] PARK J，LIM I，RYU S. Battles with false positives in static analysis of JavaScript web applications in the wild[C]// Proceedings of the 38th International Conference on Software Engineering Companion(ICSE-C). 2016：61-70.

[20] CHEN Z F，MA W W Y，LIN W，et al. A study on the changes of dynamic feature code when fixing bugs，Towards the benefits and costs of Python dynamic features[J]. Science China Information Sciences，2018，61(1)：012107.

[21] WANG B B，CHEN L，MA W W Y，et al. An empirical study on the impact of Python dynamic features on change-proneness [C]// Proceedings of 27th International Conference on Software Engineering and Knowledge Engineering. 2015：134-139.

[22] KHAN F，CHEN B Q，VARRO D，et al. An empirical study of type-related defects in Python projects[J]. IEEE Transactions on Software Engineering，2022，48(8)：3145-3158.

[23] GONG L，PRADEL M，SRIDHARAN M，et al. DLint，Dynamically checking bad coding practices in JavaScript[C]// Proceedings of the 2015 International Symposium on Software Testing and Analysis. New York，NY，USA，Association for Computing Machinery，2015：94-105.

[24] RAK-AMNOUYKIT I，MCCREVAN D，MILANOVA A，et al. Python 3 types in the wild，A tale of two type systems[C]// Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages. New York，NY，USA，Association for Computing Machinery，2020：57-70.

[25] KREJCIE R V，MORGAN D W. Determining sample size for research activities[J]. Educational and Psychological Measurement，1970，30(3)：607-610.

CHEN Zhifei，born in 1990，Ph.D，associate professor. Her main research interests include program analysis，software testing，and software maintenance.

（责任编辑：何杨）