软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(6):2608−2630 [doi: 10.13328/j.cnki.jos.007100]
©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

DDoop: 基于差分式 Datalog 求解的增量指针分析框架*

沈天琪[1,2], 王熙灶[1,2], 宾向荣[1,2], 卜 磊[1,3]

[1](计算机软件新技术全国重点实验室 (南京大学), 江苏 南京 210023)
[2](南京大学 计算机科学与技术系, 江苏 南京 210023)
[3](南京大学 软件学院, 江苏 南京 210023)
通信作者: 卜磊, E-mail: bulei@nju.edu.cn

摘 要: 指针分析是对软件进行编译优化、错误检测的核心基础技术之一. 现有经典指针分析框架, 如 Doop, 会将待分析程序和分析算法转化成 Datalog 评估问题并进行求解, 如程序规模较大, 单次求解分析时间开销较大. 在程序频繁变更发布的情况下, 相关程序分析的开销更是难以负担. 近年来, 增量分析作为一种在代码频繁变更场景下有效复用已有分析结果提升分析效率的技术受到了越来越多的关注. 然而, 目前的增量指针分析技术通常针对特定算法设计, 支持的指针分析选项有限, 其可用性也受到较大限制. 针对上述问题, 设计并实现一种基于差分式 Datalog 求解的增量指针分析框架 DDoop (Differential Doop). DDoop 实现增量输入事实生成技术与增量分析规则自动化重写技术, 将多版本程序增量分析问题表达为差分 Datalog 评估问题, 从而可以充分利用成熟的差分式 Datalog 求解引擎, 如 DDlog, 来实现端到端的增量指针分析, 并最大化兼容复用 Doop 中已有的指针分析实现, 提供透明的增量化支持. 在广泛应用的真实世界程序上对 DDoop 进行实验评估, 实验结果显示 DDoop 相较于非增量的 Doop 框架具有显著的性能优势, 同时高度兼容 Doop 中已有的各种指针分析规则.

关键词: 指针分析; 增量分析; Datalog 引擎; 增量计算; 差分式 Datalog
中图法分类号: TP314

中文引用格式: 沈天琪, 王熙灶, 宾向荣, 卜磊. DDoop: 基于差分式Datalog求解的增量指针分析框架. 软件学报, 2024, 35(6): 2608–2630. http://www.jos.org.cn/1000-9825/7100.htm
英文引用格式: Shen TQ, Wang XZ, Bin XR, Bu L. DDoop: Incremental Pointer Analysis Framework Based on Differential Datalog Evaluation. Ruan Jian Xue Bao/Journal of Software, 2024, 35(6): 2608–2630 (in Chinese). http://www.jos.org.cn/1000-9825/7100.htm

DDoop: Incremental Pointer Analysis Framework Based on Differential Datalog Evaluation

SHEN Tian-Qi[1,2], WANG Xi-Zao[1,2], BIN Xiang-Rong[1,2], BU Lei[1,3]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)
[2](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)
[3](Software Institute, Nanjing University, Nanjing 210023, China)

Abstract: Pointer analysis is a core and fundamental technology for software compiler optimization and bug detection. Existing classic pointer analysis frameworks such as Doop will transform the programs to be analyzed and analysis algorithms into Datalog evaluation problems like too large program size and solve them. As a result, the analysis time overhead of a single solution can be high, and the program analysis overhead can hardly be afforded especially in situations where programs are frequently changed and released. In recent years, as a technology that effectively reemploys existing analysis results and improves analysis efficiency under frequent code changes, incremental analysis has caught increasing attention. However, since current incremental pointer analysis techniques are often designed for

* 基金项目: 国家自然科学基金 (62232008, 62172200); 江苏省前沿引领技术基础研究专项 (BK20202001); 中央高校基本科研业务费专项资金 (020214380101)
本文由“编译技术与编译器设计”专题特约编辑冯晓兵研究员、郝丹教授、高耀清博士、左志强副教授推荐.
收稿时间: 2023-09-11; 修改时间: 2023-10-30; 采用时间: 2023-12-14; jos 在线出版时间: 2024-01-05
CNKI 网络首发时间: 2024-03-29

specific algorithms, the supported pointer analysis options are limited and their usability is significantly restricted. To this end, this study designs and implements Differential Doop (DDoop), an incremental pointer analysis framework based on Differential Datalog evaluation. DDoop implements incremental input fact generation and automatic rewriting for incremental analysis rules, expressing incremental analysis problems of multi-version programs as Differential Datalog evaluation problems. Finally, a mature Differential Datalog solution engine like DDlog can be fully utilized to achieve end-to-end incremental pointer analysis, maximizing compatibility and reuse of existing pointer analysis implementations in Doop and providing transparent support for incrementalization. Additionally, experimental evaluation of DDoop is conducted on widely adopted real-world programs. The results show that compared to the non-incremental Doop framework, DDoop has a significant performance advantage while highly compatible with a variety of pointer analysis rules existing in Doop.
Key words: pointer analysis; incremental analysis; Datalog engine; incremental computation; Differential Datalog

随着信息技术的发展, 我们已经处于一个软件定义一切的时代. 大到航空、电力、铁路系统, 小到智能家居、移动支付, 软件在生活中已经几乎是无处不在. 而随着软件的普及, 软件故障可能会严重影响人们的财产或生命安全, 甚至带来灾难性的后果, 因此人们对软件质量的要求越来越高, 而对软件质量的保障就显得愈发重要. 静态程序分析是一类重要的软件质量保障技术, 其能够对程序代码进行自动化扫描, 而无需实际运行程序.

指针分析是公认最基础的静态程序分析技术之一, 其用于在编译期静态计算程序中的指针变量在运行时可能指向的内存位置 (或对象). 指针分析的结果通常可表示为待分析程序中每个指针的指向集 (points-to set). 这些信息对于推理面向对象程序中的别名关系和过程间控制流至关重要, 被广泛地应用于错误检测[1]、安全分析[2]、程序验证[3]、编译优化[4]等一系列技术中. 这些技术可以视为指针分析的客户端分析, 也即它们需要将指针分析产生的结果作为输入, 因而指针分析的效率和精度直接影响了后续的客户端分析. 也正因如此, 虽然已经有超过 40 年的研究历史, 指针分析至今仍是静态程序分析学术研究中的重点领域[5].

然而, 指针分析目前的实用性仍有所不足. 一方面, 尽管研究人员已经在指针分析的算法和实现上投入了大量人力, 但是即使是目前最先进的全量指针分析算法或框架, 其可扩展性仍然有所欠缺[6]: 将高精度的指针分析应用到数百万行的大型程序上可能需要花费超过数小时[7]. 对于大多数生产应用来说, 这样的时间消耗通常是不可接受的. 此问题严重削弱了高精度指针分析在现实世界大规模程序中的应用能力.

另一方面, 在如今提倡的 DevOps 实践[8]中, 代码的变更和发布与以往相比更加频繁. CI/CD 是 DevOps 实践中的重要流程, 静态代码扫描是 CI/CD 中的一个关键环节, 指针分析又是静态代码扫描的基础. 频繁的变更意味着更频繁地对代码进行扫描, 这对指针分析的可扩展性提出了更高要求. 另外, 频繁的变更也意味着连续两个版本的代码差异往往很小: 通常仅限于程序的某个模块. 换言之, 对于整个程序而言, 在一次变更后, 程序的变化主要是局部的, 相应地, 指针分析结果的变化通常也仅占很小比例.

在处理代码变更频繁且单次变更变化相对局部的场景时, 传统的全程序程序分析方法效率较低: 如果我们每次都对整个程序进行完整的全程序指针分析, 会发现在绝大多数情况下, 可推断的指针指向关系实际上在新旧两次分析中是完全不变的, 尤其是对于程序中未发生代码变更或不受代码变更影响的部分[9]. 这意味着我们在这些部分进行了完全重复的冗余计算, 这无疑是对计算资源的巨大浪费, 同时也会导致开发者等待 CI/CD 流程的时间过长, 严重影响开发者的工作效率.

有鉴于此, 已经有研究人员提出了增量指针分析技术[9,10]. 增量指针分析是一种能够有效利用先前分析结果的方法, 它通过保留分析中的中间结果, 仅针对两次输入程序中不同的部分计算指针指向集的变化, 从而达到减少重复计算, 节省运行时间的效果. 近年来 Liu 等人提出的 IPA[9]和 SHARP[10]分别是上下文不敏感指针分析和上下文敏感指针分析的最新增量化工作. 它们通过在 Andersen 风格的指针分析算法中识别指向集传播的特殊性质, 从而减少增量分析过程中的冗余计算. 然而, 这两个工作对增量指针分析的支持仅限于少数几种指针分析, 并且都是基于命令式语言 (Java) 开发的, 这在一定程度上限制了它们的可选择性和易用性, 特别是在需要进行面向特定场景定制的开发时.

另一方面, Doop[11]是一个基于 Datalog 的声明式指针分析框架, 其被认为是在过去 10 年内最主流的指针分析框架, 被用于实现和比较新提出的指针分析算法[12,13]. Doop 会先将待分析程序转换为输入事实, 然后将这些输入事实和相应的分析规则传递给 Datalog 引擎进行计算, 最后将 Datalog 引擎的输出作为分析结果. Doop 提供了一组优雅的规

© 中国科学院软件研究所 http://www.jos.org.cn

则来处理不同的分析设置, 如上下文敏感性、反射处理等, 目前在 Doop 中已经实现了约 50 种不同的指针分析算法. 然而, 目前的 Doop 框架中仅提供了全程序指针分析算法, 并不支持增量指针分析算法. 此外, 从分析算法的角度来提供增量支持的方式需要对现有的指针分析算法及实现进行大规模重构, 增加了分析算法的开发和维护成本.

本文设计并实现了一种新的增量指针分析框架 DDoop (Differential Doop), 旨在为 CI/CD 流程提供高效的增量指针分析, 并充分兼容和复用 Doop 中提供的大量指针分析实现. 我们的核心洞察是将增量指针分析委托给支持增量评估机制的 Datalog 引擎, 而无需修改现有的指针分析算法和实现. 在 Datalog 中, 增量评估是一种优化技术, 用于在现有计算结果的基础上, 仅对新添加或修改的数据进行重新计算, 以减少重复计算和提高查询效率. 我们观察到, 代码的变更在 Doop 中实际上可以映射到输入事实的变更, 其可作为增量 Datalog 引擎的输入, 经增量评估后即可得到增量输出结果 (即增量分析结果), 而无需对分析规则进行任何修改.

我们的 DDoop 框架采用了基于差分数据流 (differential dataflow, DDF) 的 DDlog 引擎[14], 这是一种能够支持高效的增量评估的 Datalog 引擎. Doop 框架目前所使用的 Datalog 引擎 Soufflé [15]并不支持增量评估机制. Zhao 等人[16]尝试为 Soufflé 提供增量评估的支持, 但由于 Soufflé 设计与实现机制的限制, 他们的工作只能为 Soufflé 中的部分语法结构提供增量特性, 这种弱增量能力不足以支持 Doop 框架中的复杂指针分析规则. Ritsogianni[17]曾试图在 Doop 框架中整合 DDlog 引擎以提供增量分析的能力, 然而其方法设计与实现过于粗糙, 仅能支持 Doop 中最简单的分析. 此外, 根据其实验评估, 它只是在每一轮分析的时候将 Soufflé 引擎换成 DDlog 引擎, 实际运行的还是完整评估, 并未有效的利用 DDlog 引擎的增量能力.

因此, 在我们的 DDoop 增量指针分析框架中, 主要需要解决以下两个挑战: 1) 如何高效地将代码变更转换为输入事实的变更; 2) 如何将 Doop 中基于 Soufflé 的指针分析实现移植到支持增量评估机制的 DDlog 引擎. 为了解决这些问题, 本文将 DDoop 增量指针分析框架设计为前后端分离的架构: 前端实现增量输入事实生成, 以获取对应代码变更的输入事实变更; 后端通过自动化规则重写器, 将 Doop 中现有的 Soufflé 版本规则重写转换为 DDlog 版本规则. 通过这种方式, 我们的框架能够透明地兼容复用 Doop 框架中现有的指针分析规则, 而无需在算法实现层面进行修改即可实现增量效果. 具体而言, DDoop 框架做到了如下两点: 1) 兼容 Doop 框架本身已经拥有的丰富的、各种精度的指针分析规则; 2) 利用增量特性, 尽量减少重复计算, 以在连续的分析场景下比非增量框架的批量模式更快地得到运行结果. 实验评估结果显示, DDoop 增量框架在我们的实验基准集上相比原始 Doop 框架可实现平均约 5×, 最高约 36×的加速.

总而言之, 本文作出了以下主要贡献.

(1) 设计并实现了一种基于差分式 Datalog 的增量指针分析框架 DDoop, DDoop 框架主要面向代码变更频繁的 CI/CD 场景设计. 值得注意的是, 尽管我们并未引入新的增量指针分析算法, 但我们成功地在 DDoop 框架中实现了对 Doop 中现有的大量指针分析算法的透明增量化支持.

(2) 详细阐述了 DDoop 框架的前端和后端设计, 包括如何有效地识别和处理代码变更, 以及如何将这些变更映射到指针集的变化, 从而减少不必要的重复计算.

(3) 通过实验评估验证了 DDoop 框架的分析效率和兼容性. 实验评估结果表明 DDoop 框架对 Doop 框架中现有指针分析规则实现了兼容性和可复用性的最大化, 且在处理各种大小和复杂度的代码变更时, 都能实现显著的分析加速.

本文第 1 节介绍本工作的相关背景知识. 第 2 节介绍所提出的增量指针分析框架 DDoop 的整体架构设计与实现. 第 3 节通过实验展示 DDoop 框架的性能和兼容性. 第 4 节讨论 DDoop 框架设计与实现中的一些局限性. 第 5 节回顾相关工作. 第 6 节对本工作进行总结, 并提出未来工作展望.

1 背景知识

1.1 指针分析

在目前的大多数主流编程语言中 (如 C、C++、Java 等), 都存在指针或引用类型, 指针或引用类型变量的值

为运行时程序地址空间中的内存地址, 即它“指向”了内存中的某个位置 (或对象). 在 Java 这样的面向对象语言中, 指针可以是局部变量或实例字段, 而内存位置通常对应堆对象. 从指针到对象的这种指向关系被称为指针的指向信息, 指向信息是编译优化和静态程序分析的基础. 指针分析是一种用来静态计算程序中指针 (或引用) 在运行时的指向信息的上近似的静态程序分析技术. 经过 40 余年的发展, 指针分析已经是一个积累了大量文献[5,18]的研究领域, 但鉴于其基础地位和重要性, 指针分析仍然是静态程序分析学术研究的重点.

在本节我们主要介绍针对 Java 语言的指针分析. 形式化地说, 我们可以将 Java 指针分析视为一个映射集为 $pt:(V\cup H\times F)\rightarrow P(H)$, 其中 V 代表待分析程序中的局部变量集合, F 代表待分析程序中的实例字段集合, H 代表待分析程序中的抽象堆对象集合. 在 Java 程序中, 由于存在循环和递归, 因此在执行过程中可以无限地创建堆对象. 为了可判定性和可扩展性, 指针分析必须使用堆抽象, 将无限大小的堆划分为有限数量的抽象堆对象. 指针分析 pt 将一个局部变量 $x\in V$ 或一个实例字段 $(o_i,f)\in H\times F$ 映射到一个抽象堆对象集合 $s\in P(H)$, 这个抽象堆对象集合即指针分析结果, 被称为指针的指向集.

上述形式化定义的 Java 指针分析是上下文不敏感指针分析. 上下文不敏感指针分析的定义和实现相对简单, 并且针对大规模程序的可扩展性更好. 然而上下文不敏感指针分析未对程序中的方法调用上下文进行建模, 在分析过程中会合并不同过程间动态执行路径上的行为, 导致较大的分析精度损失. 实践证明, 上下文敏感性对于提高 Java 程序指针分析精度很有用. 上下文敏感指针分析对不同调用上下文下的相同方法和堆对象分别进行分析, 从而区分不同过程间动态执行路径上的行为. 理论上, 调用上下文可以概括为与调用点的控制流相关的程序状态的某种抽象. 因此, 可以通过使用的不同上下文元素来区分上下文, 得到上下文敏感性的不同变体. 对于 Java 等面向对象语言, 通常使用 3 种上下文敏感性, 即调用点敏感性、对象敏感性以及类型敏感性, 它们的上下文元素分别是调用点、抽象堆对象 (分配点) 和类型.

上下文敏感指针分析同样可以形式化定义为一个从指针到指向集的映射. 其与上下文不敏感指针分析的定义的不同之处在与其中的指针和抽象堆对象都是带上下文限定的. 具体来说, 上下文敏感指针分析是一个映射 $cpt:(C\times V\cup HC\times H\times F)\rightarrow P(HC\times H)$, 其中, C 代表方法上下文, 而 HC 代表堆上下文. 上下文敏感指针分析 cpt 会将一个带上下文限定的局部变量 $(c,x)\in C\times V$ 或实例字段 $(hc,o_i,f)\in HC\times H\times F$ 映射到一个带上下文限定的抽象堆对象集合 $cs\in P(HC\times H)$, 即上下文敏感指向集.

除上下文敏感性外, 指针分析中还有其他几个影响精度的维度: 流敏感性、字段敏感性、堆模型等. 一般来说, 某个维度不敏感必然牺牲算法的一部分精度, 而某个维度敏感必然增加算法复杂度, 降低算法实现的效率. 指针分析当前研究工作面临的主要问题是, 在保证算法精度的同时平衡时间/空间上的消耗, 设计并实现可扩展的、高精度的指针分析算法.

1.2 Datalog

Datalog 是一种基于逻辑编程的声明式查询语言, 其在语法上是 Prolog 的一个子集. Datalog 被广泛用于处理结构化数据, 并在数据库管理、人工智能、声明式网络和程序分析等领域中发挥着重要作用. 这些领域使用 Datalog 语言作为查询图和关系结构以及实现迭代和递归的声明式抽象, Datalog 提供了一个声明式接口, 允许程序员专注于任务 (what to do) 而非低级细节 (how to do).

Datalog 语言的语法非常简洁. 一个 Datalog 程序由一组 Datalog 规则组成, 规则包含规则头和规则体两部分. 规则格式如公式 (1) 所示.

$$A\ \text{:-}\ B_1,\ B_2,\ldots,B_n \tag{1}$$

其中, A 和 B_i 表示 Datalog 中的谓词, A 是这条规则的规则头, 而 $B_1, B_2,\ldots, B_n$ 的合取构成规则体. Datalog 的逻辑基础源自一阶谓词逻辑, 每条规则都是一个蕴含式, 例如公式 (1) 对应的蕴含式为 $B_1\wedge B_2\wedge\ldots\wedge B_n\rightarrow A$. 一个谓词可以带有参数, 如 $P(e_1,e_2,\ldots,e_k)$ 表示一个元谓词. 谓词的实例元组被称为事实.

$$\begin{aligned}&reachable(X,Y)\ \text{:-}\ edge(X,Y)\\&reachable(X,Y)\ \text{:-}\ edge(X,Z),reachable(Z,Y)\end{aligned} \tag{2}$$

© 中国科学院软件研究所 http://www.jos.org.cn

公式 (2) 中展示了一个实例 Datalog 程序, 其中包含 2 条用于计算图中的可达性信息的规则. 谓词 *edge*(*X*, *Y*) 表示图中从节点 *X* 到节点 *Y* 存在一条边, 谓词 *reachable*(*X*, *Y*) 表示图中从节点 *X* 到节点 *Y* 存在一条可达路径. 第 1 条规则表示若 *X* 和 *Y* 间存在一条边, 则 *Y* 从 *X* 可达; 第 2 条规则表示若从 *Z* 到 *X* 可达且 *Z* 和 *Y* 间存在一条边, 则 *Y* 从 *X* 可达.

给定一个 Datalog 程序以及一组输入事实, 经 Datalog 引擎评估后得到一组输出事实. Datalog 引擎通常使用自底向上的评估模型, 它重复应用规则, 直到不再有新的事实产生或达到终止条件. 为了在计算的时候尽量避免重复之前迭代中已经完成的结果, 提出了一种名为半朴素法 (semi-naive) 的更优的自底向上评估方法. 此外, 在实际应用中, Datalog 常常需要处理大规模的数据集和复杂的查询, 涉及到输入事实的变化. 为了提高 Datalog 的性能, 引入了增量评估技术. 基于差分数据流的增量评估是一种优化技术, 它通过跟踪输入数据的变化, 并更新相应的输出结果, 从而避免重复计算和减少计算开销. 这种增量评估技术在处理大规模数据和持续查询时非常有效, 显著提高了 Datalog 的查询性能和效率.

如今, Datalog 已经发展成为一类语言规范: 由于 Datalog 在不同应用场景中的发展和拓展, 出现了一些不同的 Datalog 方言. 不同的 Datalog 方言加入了为了增强语言表现力或方便程序员开发而设计的拓展语言特性. 例如 bddbddb[19]、Soufflé[15]、DDlog[14]等. 尽管这些方言间并不能直接互通, 它们都可以被归类为“Datalog 语言”. Datalog 方言的实现被称为 Datalog 引擎.

最后介绍 Datalog 语言在程序分析领域中的应用. 最初, 静态分析研究人员因为使用传统的命令式语言实现具体的程序分析算法而感到困扰, 因为这需要开发人员具备强大的工程能力. 大量的开发时间并未用于实现算法, 而是用于处理各种细节性的工程问题, 例如极其复杂的数据结构设计、内存管理、程序并行化过程中的信号量同步、死锁问题的解决、异常处理、边界条件检查甚至性能调优等.

声明式的 Datalog 语言在这方面有其突出的优势: 由于声明式语言的抽象特性, 这类工程问题可以委托给 Datalog 引擎处理, 让开发人员可以专注于算法逻辑, 无需过多关心底层实现细节. Datalog 语言在指针分析领域的应用可以追溯到 2006 年的 bddbddb[19]. Doop 是当前最先进的 Java 指针分析框架, 其同样采用了基于 Datalog 的声明式分析范式. Doop 将指针分析算法表述为 Datalog 规则, 并实现了一个从 Java 程序中抽取语义信息并转换为 Datalog 引擎可接受的输入事实格式的前端. 通过这种方式, Doop 框架成功地将指针分析从指针赋值图上的一个图计算问题转换为了 Datalog 引擎上的一个数据查询问题.

2 DDoop 框架

本节我们主要介绍针对 Java 语言的增量指针分析框架 DDoop 的架构设计, 并介绍实现中的相关细节. 在现有非增量的 Doop 指针分析框架的基础上, 我们在 DDoop 框架中提出了增量输入事实生成技术和 DDlog 增量分析程序自动生成技术, 以兼容复用 Doop 框架中已有的大量指针分析实现, 并提供高效的透明增量化支持. 图 1 展示了 DDoop 增量指针分析框架的整体架构.

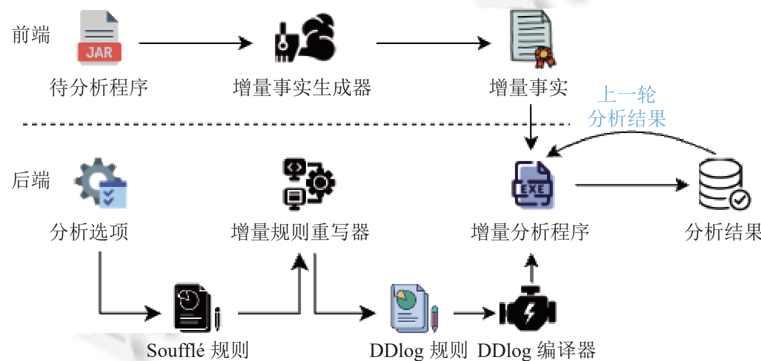

图 1 框架整体架构

© 中国科学院软件研究所 http://www.jos.org.cn

从整体架构上, DDoop 框架主要可分为前端和后端两个部分, 这两个部分都拥有增量特性.

(1) 增量生成输入事实的前端. 其自动化处理待分析程序, 从中抽取变更信息并转化为后端增量评估所需的增量输入.

(2) 自动化生成 DDlog 增量分析程序的后端. 根据输入的分析选项, 后端会将原始 Doop 框架的 Soufflé 规则自动化重写为支持增量评估的 DDlog 规则, 并生成相应的增量分析程序.

首先, 我们对图 1 所示的 DDoop 框架的整体运行流程概述如下: 根据分析输入, 我们可以确定分析使用的 Soufflé 指针分析规则, DDoop 后端会将这份分析规则自动化重写为 DDlog 版本的增量分析程序. 随后, DDoop 前端会收到待分析程序对应的一系列代码提交, 其将会按照顺序, 将代码变更转化为 DDlog 增量分析程序可接受的增量输入事实形式. 增量分析程序接收到代码变更的增量输入事实后, 增量评估产生分析结果的增量输出.

接下来, 我们将详细介绍 DDoop 框架前端和后端这两个部分.

2.1 前端: 增量输入事实构建

DDoop 框架和 Doop 框架都将指针分析问题归约为 Datalog 程序评估问题. Datalog 引擎接受一组事实和规则作为输入. 在 Doop 框架中, 规则对应指针分析算法, 而输入事实则对应待分析程序的一组语义性质. 为此, 我们需要一个前端来对待分析程序进行预处理, 从中抽取与指针分析相关的程序语义性质作为 Datalog 引擎的输入事实. 前端的存在是为了实现分析框架的端到端过程: 将不能直接被 Datalog 引擎处理的 jar 包转换为其可接受的输入事实形式.

Doop 框架中已经提供了一个前端, 能够将一个完整的 Java 程序 (jar 包) 转换为 Soufflé 引擎的输入事实. 其前端的输入生成器抽取语义信息时不涉及类之间的相互依赖信息. 它使用 Soot 框架加载 jar 包中所有的类字节码文件, 然后对已加载的每个类并行化的抽取其中的语义信息. 在具体处理时, 每个类中的方法在逻辑上都是独立的. 但是 Doop 中的前端在我们的增量场景下存在一些问题.

● 第 1 个问题是, Soufflé 引擎和 DDlog 引擎的输入在形式上存在细微差异: DDlog 引擎是为增量评估设计的, 输入时需要标记每条输入事实是插入还是删除 (特别地, 某一条事实在增量过程中的更新会被处理为一条删除旧值和一条插入新值的组合); 而 Soufflé 引擎将所有输入事实均视为插入.

● 第 2 个问题是, Doop 框架的前端并未考虑到增量设计: 对于待分析程序的每一个版本, 其都会从头开始处理整个待分析程序. 但在连续输入的增量场景中, 绝大多数的 Java 类文件都是不变的, 对应的语义信息是完全相同的. 这就导致了重复计算问题.

解决第 1 个问题相对容易, 这只是格式上的区别. 我们只需要: 1) 标记每条输入事实是插入还是删除; 2) 格式转换. 然而, 对第 2 个问题, 我们需要一个支持增量事实生成的全新前端. 这也是我们在此处的最主要挑战.

2.1.1 框架前端架构

在我们的增量指针分析框架中, 我们希望避免重复计算: 即对于那些在代码变更中未发生变化的类和方法, 我们可以复用此前生成的输入事实. 为此, 对于给定两个版本的 jar 包, 我们需要一种方案来识别出在代码变更中未发生变化的类文件, 并在处理时跳过它们. 为此, 我们需要解决以下 3 个主要问题.

(1) 划分合适的基本“增量单元”, 其粒度必须适当. 在实现增量分析时, 选择适当的“增量单元”是至关重要的. 该单元应能在保证分析精度的同时, 最大程度地降低实现和计算的复杂度. 我们需要确定, 是将方法、类还是编译模块视为“增量单元”. 增量单元越大, 每一次出现变更时需要重做的范围就会越大, 但是增量单元越小, 用来判断增量的范围就相对更加复杂.

(2) 找出能够标识增量单元未发生变化的元素, 或者说一种“摘要”. 为了实现快速的增量变更检测, 我们需要一种能够快速且准确地反映“增量单元”变化的“摘要”机制. 这种机制可以采用“增量单元”哈希值, 或者是结合“增量单元”的某些性质, 如代码行数, 方法数量等. 选择哪种“摘要”机制取决于它们能否提供足够的信息来判断代码的变化, 同时又能在短时间内完成计算.

(3) 维护我们在增量过程中的必要信息. 在增量分析过程中, 有效地管理和存储分析信息是一个挑战. 我们需

© 中国科学院软件研究所 http://www.jos.org.cn

要跟踪哪些“增量单元”已经被处理, 哪些还未被处理. 此外, 我们可能需要保存一些中间结果, 以便在后续过程中使用. 因此, 我们需要一个高效、可靠且易于操作的信息管理的方式, 以支持复杂的查询和更新操作.

针对上述问题, 我们设计了一个高效且精确的、支持增量输入事实生成的前端. DDoop 的前端架构如图 2 所示. 框架前端中分别给出了如下解决方案: 基于实证研究的增量单元粒度选择与确定, 基于程序性质与哈希的增量单元摘要设计, 基于内存数据库的增量信息高效存储与管理. 在第 2.1.2–2.1.4 节, 我们将对各部分进行详细介绍, 而在第 2.1.5 节, 我们将对一个实现中的优化进行介绍.

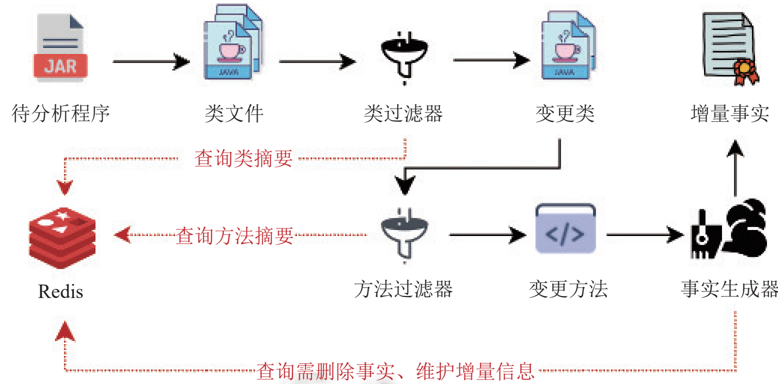

图 2 框架前端架构

2.1.2 增量单元粒度选择

确定合适的增量单元粒度是解决增量问题的关键之一. 在 Java 增量分析中, 增量粒度可以有多种选择, 例如编译单元级、类级、方法级、语句级甚至更加精细的表达式级. 在 Doop 框架中, 我们注意到, 它天生就在类级并行, 因此可以直接排除粒度更粗的编译单元级增量粒度. 此外, 尽管 Doop 框架的并行处理粒度在类级别, 但其处理逻辑中方法与方法之间仍然是独立的, 所以从实现的角度来说, 比方法级别更加精细的增量单元可能会引入相当规模的强耦合关系, 不能做到增量单元间的高效并行.

因此, 在 Doop 框架设置下, 我们主要关注类级和方法级两种增量单元. 为了确定哪一种增量单元更加适合我们的场景, 我们在一组基准项目上开展了关于增量单元粒度选择的实证研究, 通过简单的快速实验, 对比不同粒度的增量单元对于增量输入事实生成的效率的影响.

我们在 DaCapo 基准 (如 xalan、Jedis、PMD 等) 和一些真实世界大型项目 (如 ErrorProne、ZooKeeper 等) 上开展了实证研究. 我们发现, 对运行时 Soot 对象进行简单的哈希在多次运行中并不能保证幂等性, 因此我们选择将其中的 SootMethod 对象转化为中间表示 (intermediate representation, IR). 我们认为, 如果 IR 表示不同, 则此方法发生了变化 (关于增量单元摘要的设计细节, 见第 2.1.3 节). 在我们的两级摘要设计过程中, 我们首先会过滤出那些内容存在变化的类, 之后才会对这些类中的每一个方法计算摘要以进一步缩小增量计算范围. 那么, 这一策略要相较于基于类层级的增量方案节省时间, 就要求在变化过程中平均每一个类变化的方法数量相对有限. 不妨假设我们发生变化的类中平均一共有 n 个方法, 其中平均有 x 个方法发生变化, 记单个类事实生成时间为 a, 生成摘要信息为 b, 要让我们的二级摘要方案时间消耗更少, 即需要满足如下约束: $a\cdot n\geqslant b\cdot n+a\cdot x$. 实证研究结果显示, 对每个方法做从 SootMethod 对象到摘要的转化操作平均需要 0.09 ms, 而为每个方法生成相应的事实平均需要 0.25 ms. 通过前述不等式约束可以得出结论: 类中少于 64% 的方法变化时我们的二级摘要可以节约时间.

根据实证研究的实验结果以及上述结论, 我们发现方法级别的增量单元能够更有效地处理这个问题, 因此我们选择以方法作为 DDoop 前端的基本增量单元粒度.

2.1.3 增量单元摘要设计

为了标识一个增量单元在代码变更中是否发生变化, 我们需要为增量单元设计一种摘要机制以表征这种变化. 我们设计了一种高效的两级摘要方案, 旨在平衡性能和增量粒度. 具体而言, 我们首先使用类的字节码文件对

应的字符数组的哈希值加长度作为一个类的标识. 一旦这个标识发生变化, 就说明类中存在改变, 需要进入下一级摘要. 在第 2 级摘要中, 我们会对这个类在方法级别进行匹配. 我们首先将 SootMethod 对象转化为方法的 IR 表示, 并计算哈希作为摘要. 在这样的摘要设计下, 如果一个方法的源代码发生了任何有意义的变化, 这种变化都会反映到字节码层面的摘要上.

我们以图 3 中的一次代码变更为例, 对我们的两级摘要方案的设计进行说明. 以上的代码中, 旧版本包含了 A1, B, C1 这 3 个类, 新版本中则包含了 A1, B, C2 这 3 个类, 同时 A1 类中的方法在此次变更中也产生了变化. 为了方便叙述, 我们在此处忽略它们依赖的库以及其中没有显式定义的任何其他方法. 在第 1 级摘要中, 我们会使用 A1.class, B.class, C2.class 这 3 个字节码文件对应的字节码数组的摘要 (hash+长度) 对类进行标识, 于是就可以识别出不变的类 B, 变化的类 A1, 删除的类 C1, 新增的类 C2. 针对 A1, 它会进入第 2 级摘要, 我们会对其中的方法 f(), f2(), method() 生成对应的 IR 摘要, 并且和之前旧 A1 类对应摘要进行对比. 我们就可以确定 A1 中新增了 f(), 删除了 f1(), f2() 发生变化以及 method() 保持不变. 对于类的字段信息、修饰符信息, 当类发生变化时, 我们会默认这些信息发生了改变, 进行增量处理.

```
1      public class B {}
2    - private class C1 {}
3    + private class C2 {}
4      private class A1 {
5        B b;
6    -   void f1(){}
7    -   void f2(){}
8    +   void  f(){}
9    +   void f2(){this.b =this.b2;}
10       void method() {}
11     }
```

图 3　Java 代码变更示例

2.1.4　增量信息存储与管理

在增量输入事实生成的过程中, 需要记录一些必要的增量信息. 其中最主要的内容包括前文提及的增量单元的信息以及对应生成的 facts 组. 我们的框架首先基于上一轮增量生成中维护的当前程序“活跃”增量单元对应的增量信息确定本轮增量生成中哪些增量单元需要重新进行处理. 对于那些本轮不再出现的增量单元, 我们的框架将会将它们对应的 facts 从缓存中取出标记为删除; 对于本轮新出现或者发生变更的增量单元, 则将维护它们的相关信息, 并将新的 facts 写入缓存. 通过这样的机制, 缓存系统独立记录了增量前端当前的状态, 提供了下一轮增量分析所需的必要信息.

为此, 我们需要设计一种缓存方案来维护这些增量信息. 我们考虑了几种常见的缓存方案: 基于文件系统的缓存、基于磁盘数据库的缓存、基于内存数据库的缓存以及前端程序在内存内直接管理缓存的方案.

从实现方面来看, 数据库方案相对简单, 开发者只需定义数据结构, 而无须关心具体如何安排这些数据, 如何做增删改查等这些细节问题. 同时内存数据库在此场景下的性能远高于磁盘数据库. 综合考虑效率和开发效率, 我们最终决定选择使用成熟的内存数据库 Redis 作为我们存储和管理增量信息的方式.

从软件工程方面来看, 在现代软件架构中, 应用通常会在分布式节点上部署, Redis 正适合这样的场景. 在这样的架构中, 节点间的信息交换需要通过网络栈进行, 频繁的 I/O 操作可能会成为系统中最耗时的部分. 我们通过按需读写的策略优化了 I/O 粒度和读写频率: 我们的系统会首先基于一份相对较小的摘要信息确定哪些类真正发生了变化. 对于那些未发生变化的增量单元对应的 facts, 我们完全不会在程序运行过程中读写它们. 通过这种优化, 我们的设计能够在高度分布式的环境中同样有效地降低 I/O 开销, 提高系统性能.

© 中国科学院软件研究所 http://www.jos.org.cn

2.1.5 增量前端剪枝优化

通过增量单元的设计, 我们已经大幅减少了需要重新抽取输入事实的类和方法数量. 然而在实现过程中, 由于框架使用的 Soot 框架的类加载机制, 为了获取一个类的所有依赖, 需要加载大量的类. 不仅包括输入的 jar 包中的所有类, 还包括这些类可能引用的系统类库中的类. 这个过程是迭代的, 直到无法继续找到传递依赖的类为止. 这样的类加载策略是为了尝试对输入的待分析程序进行一个尽可能精确的建模. 在 DDoop 框架的早期实现中, 我们采用了这种类加载策略, 以保证与原始 Doop 框架有等价的指针分析精度. 然而, 这种实现相对低效.

在我们关注的变更频繁的增量场景下, 一次代码提交引入的变化通常是局部的, 不会导致整个程序引用的基础类库发生巨大的变化. 因此, 全面类加载中也包含重复计算. 因此, 我们提出了对类加载过程的剪枝优化策略, 通过减少需要加载的类数量来提升前端增量输入事实生成的速度.

图 4 展示了我们所采用的剪枝策略. 在增量输入事实生成的类加载过程中, 我们不再加载间接引用的类库中的类. 也即在加载类时, 我们只会包括应用类和被应用类直接引用的类库中的类. 然而, 我们的剪枝策略可能会对精度产生影响. 我们将在第 3.4 节和第 4.2 节分别从实验和理论层面就剪枝策略对精度的影响进行讨论.

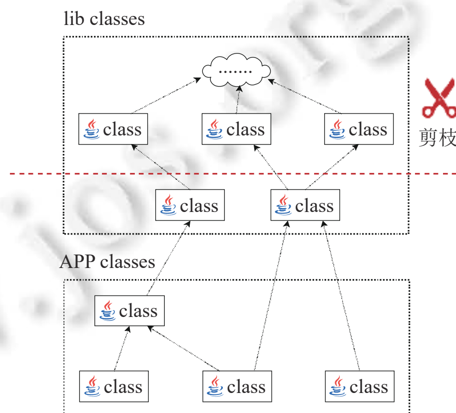

图 4　类加载过程剪枝策略示意图

2.2 后端: 增量分析程序生成

Datalog 是 Prolog 的一个子集, 它是一种更为限制和精简的逻辑编程语言. 现代 Datalog 引擎都遵循了 Datalog 语言的理论规范, 但为了增强表现能力都对语言进行了拓展. 然而不同引擎之间的拓展并不完全相同, 并且目前也不存在通用的标准语法. 因此, 不同的 Datalog 引擎之间一般并不能直接互通.

Soufflé 和 DDlog 是当前流行的两种 Datalog 引擎. Soufflé 是 Doop 框架目前所采用的 Datalog 引擎, 而 DDlog 支持高效的增量评估. 如果将 Doop 中面向 Soufflé 引擎实现的分析规则直接提交给 DDlog 编译器, 因为语法细节层面的细微变化, 是无法通过编译的.

这意味着, 切换具体的 Datalog 引擎需要对所有已经有的 Datalog 规则进行语义等价的重写, 这无疑是一项艰巨的任务. Doop 框架在发展过程中就有过一次切换 Datalog 引擎的经历, 它从对学术使用有相当限制的 LogicBlox 引擎切换到了当时新发布的、专为程序分析开发的 Soufflé 引擎. 开发者为此耗费了大量精力 (大约 10 人月)[20]完全重写了指针分析的规则库, 将 LogicBlox 引擎所用的 LogiQL 语言切换到了 Soufflé 语言.

在我们框架的设计目标中, 我们需要兼容 Doop 框架已有的丰富的 Soufflé 指针分析规则. 这就意味着我们需要对 Doop 框架中已有的 Soufflé 指针分析规则进行完全的重写. 然而, 这其中存在两项主要的挑战, 使得像 Doop 开发者曾经做过的那样进行手动转换在软件工程领域变得不可接受.

(1) Doop 框架利用 Soufflé 语言的模块化特性动态地根据具体的分析选项情况从规则库中“装配”生成新的规则, 而 DDlog 语言中没有显式的模块化语法. 这就意味着, 对于 Doop 中的数十种精度的分析及其对应的大量的分

析选项, 我们可能需要准备数千个对应不同情况的 DDlog 规则文件. 即使这种手动转换可以实现, 但是它会带来巨大的工作量, 并导致规则库代码的严重膨胀. 更糟糕的是, 这种设计将会给后期对这些规则的维护带来巨大的困难.

(2) 随着 Doop 框架的流行, 有相当一部分场景性的程序分析算法和新的指针分析算法实际上就是在 Doop 框架所提供的指针分析规则基础之上开发的. 从兼容性出发, 我们需要尽力支持这种类型的潜在代码. 然而, 如果我们使用的是手动转换, 我们将无法做到这一点. 因 Datalog 引擎切换导致的兼容性破坏的一个例子是 Zipper[12], Zipper 是由 Li 等人提出的一种新的选择性上下文敏感指针分析算法, 在基于 LogicBlox 引擎的旧版本 Doop 中实现, 目前由于当前版本的 Doop 手动切换到了 Soufflé 引擎, 导致 Zipper 在当前版本的 Doop 中不可用[21].

面向这两项挑战, 我们设计了一个自动化的 Datalog 规则重写器. 这个自动化规则重写器是我们框架后端的核心, 其设计目标是能够处理 Doop 框架内任何分析选项的组合产生的 Soufflé 指针分析规则. 通过规则重写器, 我们可以全自动化地将这些分析规则重写为 DDlog 引擎的格式, 而无需进行繁琐而容易出错的手工转换. 同时, 这个自动化重写器也使我们可以对基于 Doop 框架开发的其他分析进行兼容.

2.2.1 框架后端架构

我们框架的后端架构如图 5 所示. 其流程基于 Doop 框架的规则生成部分. 首先, 我们将分析选项输入 Doop 框架, 以获得对应的 Soufflé 规则文件. 然后, 这个规则文件会被输入到我们的自动化规则重写器中, 转换为语义等价的 DDlog 规则. 最后, DDlog 工具链会将这些规则编译成支持增量评估的 DDlog 程序, 以响应前端生成的增量输入事实.

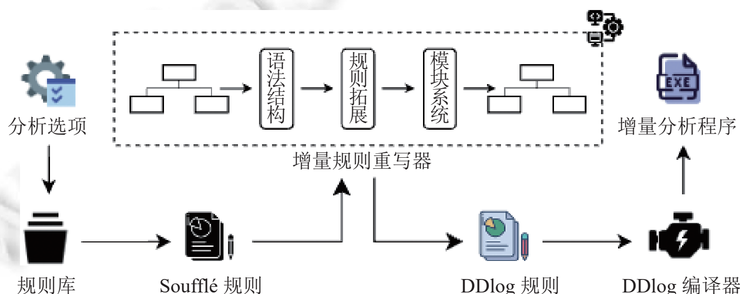

图 5 框架后端架构

如前文所述, Soufflé 和 DDlog 是两种不同的 Datalog 方言, 它们在语法和拓展上都存在一系列的差异. 这些差异主要可以分为 3 类.

(1) 第 1 类是语法结构差异. 在 Soufflé 和 DDlog 中很多语言结构的语义是相同的, 但是在具体的语法定义上存在细微差异, 例如类型系统的命名、变量命名规范等. 另外, 在 Soufflé 还存在一些仅用于辅助编译的注解等.

(2) 第 2 类是规则拓展差异. Soufflé 对 Datalog 规则的拓展比 DDlog 更为激进, 例如 Soufflé 规则中引入了析取元素, 并允许输入关系的可变性.

(3) 第 3 类是模块系统差异. Soufflé 拥有组件结构, 这是一个相对完善的模块系统, 而 DDlog 中则不存在对应的结构. 正因为有了这个组件系统, Doop 框架可以根据分析选项通过条件编译的方式组装出我们需要的规则.

因此, 我们的重写过程需要在这两种方言之间建立一个映射, 以克服两者之间的语法和拓展差异. 接下来, 我们将分别针对这 3 类差异进行详细的介绍, 并提出相应的解决方案.

2.2.2 语法结构重写

根据语法结构的语义性质, 我们可以将语法结构的处理方式进一步细分为两种策略: “重写”和“忽略”.

“重写”策略主要针对那些在规则层面有实际意义的语法结构, 如类型系统的差异、文法格式的区别, 以及变量命名规范的不同等. 处理这类差异的基本策略是对抽象语法树 (AST) 的子树进行模式转换. 例如, 参考图 6 中 Soufflé 参考代码的第 1 行和 DDlog 参考代码中的第 5 行中的类型定义, 可以发现这两种语言在类型定义上的差

© 中国科学院软件研究所 http://www.jos.org.cn

异: 首先是类型定义语法上的轻微差异, 例如 Soufflé 中的“.type”与 DDlog 中的“typedef”; 其次则是类型系统上的差异, 例如 Soufflé 中的“symbol”表示所有的字符串类型而 DDlog 中对应的类型名为“Tsymbol” (“IString”的别名). 这些差异都可以通过在 AST 的结构上进行对应的转换来简单地解决. 同样的处理方案也适用于变量命名风格、内置函数命名、输入规则是否不变等方面的差异.

```
1   .type  node = symbol
2   .decl  node_ord(a:node,b:number)
3   .output  node_ord
4   .comp Graph{
5   .decl  edge(a:node,b:node)
6   .input  edge
7   }
8   .init  graph = Graph
9
10  node_ord(?a, ?a_ord) : -
11  (?a_ord = ord(?a), graph.edge(?a, _));
12  (?a_ord = ord(?a), graph.edge(_, ?a)).
13  .plan  1:(2,1), 2:(1,2)
```

(a) Soufflé 参考代码

```
1   import fp
2   import intern
3   import souffle_lib
4   import souffle_types
5   typedef  node = Tsymbol
6   relation  graph_edge(a:node, b:node)
7   input relation  graph_edge_shadow(a:node, b:node)
8   graph_edge(a, b) : -  graph_edge_shadow(a, b).
9   output relation  node_ord(a:node, b:Tnumber)
10  node_ord(a, a_ord) : -
11  var  a_ord = ord(a), graph_edge(_, a).
12  node_ord(a, a_ord) : -
13  var  a_ord = ord(a), graph_edge(a, _).
```

(b) DDlog 参考代码

图 6 Soufflé 和 DDlog 语义等价的 Datalog 代码示例

“忽略”策略主要用于处理作为编译辅助内容的语法结构, 这些元素在 DDlog 中并没有对应的表示. 例如, 图 6 中 Soufflé 参考代码第 13 行的. plan 语句就是一种指导规则子句重排策略的语法结构. 在“忽略”策略下, 这类与 Soufflé 特定的语句将被我们的重写器直接忽略.

2.2.3 规则拓展重写

相比 DDlog 引擎, Soufflé 引擎在规则的拓展层面拥有更加强大的支持. 例如, 它可以支持 Datalog 语言不支持的析取操作; 它也可以支持否定规则子句的无条件前置; 此外, 它还拥有一个优秀的规则子句重排器, 不仅可以推断出对应规则可能的最优半朴素评估顺序, 还可以自动消除评估中的不确定规则.

对于这些与语义密切相关的差异, 不能通过单纯的 AST 结构重写来解决. 不过幸运的是, Datalog 规则的表现力保证了这些 Soufflé 引擎的拓展在 DDlog 中存在一种语义等价的语法形式.

第一, 针对析取. 虽然 DDlog 中没有直接表示析取的语法结构, 但是通过并列两条规则的方式, DDlog 可以表示这个语义. 以下是一个 Soufflé 中简单的例子.

```
node_ord(?a, ?a_ord):-
(?a_ord = ord(a), graph.edge(?a, _));
(?a_ord = ord(a), graph.edge(_, ?a)).
```

这个规则表示 node_ord 关系中的内容需要增加来自于两个推导的结果的并集, 中间通过表示析取的“;”连接. 遵循 Datalog 规则, 我们也可以将它重写为不包含“;”的等价形式如下.

```
node_ord(?a, ?a_ord) :- ?a_ord = ord(?a), graph.edge(?a, _).
node_ord(?a, ?a_ord) :- ?a_ord = ord(?a), graph.edge(_, ?a).
```

在图 5 的例子中, 这条规则出现在 Soufflé 参考代码中第 10–12 行, 和转换后 DDlog 参考代码中第 10–13 行的两条规则相比, 我们还可以看到规则子句的重排, 有关于这部分的重写, 我们稍后就会谈到它.

第二, 针对函数类或带否定规则子句的重排. 在处理此类规则子句时, Soufflé 和 DDlog 对子句顺序的限制有

所不同, DDlog 中要求规则子句中对变量的使用不能出现在能确定它们的规则子句之前, 而 Soufflé 对此限制更宽松, 允许此类规则的存在. 考虑如下以 Soufflé 语法写成的示例规则.

node_ord(?a, ?a_ord) :- ?a_ord = ord(?a), graph.edge(_, ?a).

这条示例规则表示 node_ord 关系需要增加来自于 graph.edge 关系中的元素?a 在被 ord 函数处理后的返回值. 其中 ord 函数可以对某一个确定性的字符串输入?a 返回一个运行中的唯一对应值. 对于 DDlog 的编译器, 在处理规则的第 1 条子句?a_ord = ord(?a) 时, 它没能找到?a 的定义, 因而编译失败.

为了解决这个问题, 我们采用了规则重写方案. 通过分析规则子句中的 def-use 关系, 我们可以产生规则子句的一个重排版本. 新的规则子句确保了每个变量在被使用之前都已经被确定. 重排后的规则如下.

node_ord(?a, ?a_ord) :- graph.edge(_, ?a), ?a_ord = ord(?a).

2.2.4 模块系统重写

组件系统是 Soufflé 在 Datalog 上为增强模块化而带来的一个语法糖, 而 DDlog 目前尚未支持模块化. 但在编译过程中, Soufflé 编译器会在编译期首先进行解糖: 将模块化的规则进行"展开". 因此, 我们的重写器针对模块化的重写策略与之类似, 具体描述于算法 1 中.

算法 1. Soufflé 到 DDlog 的规则重写算法.

输入: A Soufflé program Ps;

输出: A DDlog program P_D.

1. **for** each node v in $AST(Ps)$ **do**
2.   if v is a component declaration **then**
3. o_c = new Component // 包含组件名称、类型参数列表、继承关系及组件体
4.   **for** each node u in $AST(o_c)$ **do**
5.   **if** u is an .override directive **then**
6.   Mark relations that o_c overrides // 被标记的关系不会输出到 P_D 中
7.   **end if**
8.   **end for**
9.   **end if**
10. **end for**
11. **for** each node v in $AST(Ps)$ **do**
12.   **if** v is a component instantiation of o_c **then**
13. o_c.instantiate() // 调用 Component 对象的 initantiate 方法
      // 将组件的类型参数实例化为具体类型
      // 实例化所有父组件并合并到当前组件实例中
14.   **end if**
15. **end for**
16. **for** each node v in $AST(Ps)$ **do**
17.   **if** v is a declaration in a component body **then**
18. v_D ＝rewrite(v) // 将声明以 DDlog 语法重写
19.   add v_D to P_D // 将重写后的声明添加到目标 DDlog 程序中
20.   **end if**
21. **end for**
22. **return** P_D

针对算法 1 简要介绍如下.

1) 程序首先遍历 Soufflé 程序的抽象语法树, 并在遇到组件声明时创建一个 Component 对象.

2) 在第 2 次扫描的时候, 当程序遇到组件实例化时, 它会调用 Component 对象的 instantiate 方法. 这个方法负责将组件的类型参数替换为实例化时提供的具体类型参数. 同时, 它还会处理组件的继承关系, 实例化所有父组件, 并将它们的内容合并到当前组件实例中.

3) 在实例化过程中, 程序会处理.override 指令, 用于标记当前组件覆盖了哪些关系.

4) 最后, 程序将遍历组件体中的声明 (如关系、规则等), 并将它们重写为 DDlog 语法.

以 Soufflé 参考代码中的不包含继承关系的简单组件为例, 我们在扫描到.init graph = Graph 命令的时候, 我们就会开始进行实例化, 由于这里没有.override 指令, 所以对 edge 的定义会被处理, 并且被实例化为 graph_edge, 之后就是按照此前讨论的语法结构重写规则进行重写 (第 2.2.2 节).

3 实验评估

为了展示我们的 DDoop 增量指针分析框架在代码变更频繁的场景下的性能和对 Doop 框架现有指针分析规则的兼容性, 我们在一组代码变更上对 DDoop 框架进行了实证评估, 并与原始 Doop 框架及当前尝试为 Doop 框架添加增量支持的现有工作进行了实验对比. 我们的实验评估主要试图回答以下 3 个研究问题.

• RQ1: 我们的增量 DDoop 框架与原始的非增量 Doop 框架相比性能表现如何?

• RQ2: 我们的增量 DDoop 框架对 Doop 中现有指针分析规则的兼容性如何?

• RQ3: 我们在 DDoop 前端中进行的剪枝操作对分析精度的影响如何?

3.1 实验设置

实验平台. 我们的所有实验在一台配备 Intel(R) Xeon(R) Gold 6240 @ 2.60 GHz 处理器 (75 核), 285 GB 内存, 1 TB 磁盘的 Ubuntu 20.04.6 LTS 服务器上运行. 在实验评估中我们设置最大可用线程数为 8.

基线工具. 我们在实验评估中对比的基线工具分别是原始 Doop 框架以及两种带基础增量支持的 Doop 框架. 基线工具的详细信息如下.

• Doop: 基于 Soufflé (2.0.3) 的原始 Doop 框架 (4.24.10).

• Doop-SE: 基于 Soufflé-elastic (540cf8d) 的带基础增量支持的 Doop 框架 (4.24.10). 在 Soufflé-elastic 工作中并未提供与 Doop 框架的整合. 这里的 Doop-SE 是我们框架前端的一个修改版 (修改了增量指令格式以适配 Soufflé-elastic) 与 Soufflé-elastic 增量引擎的一个简单组合.

• Doop-DDlog[17]: 初步整合了 DDlog 引擎 (0.4.0) 的 Doop 框架 (4.24.10).

我们的 DDoop 增量指针分析框架是基于 Doop (4.24.10) 和 DDlog (1.2.3) 进行实现的.

指针分析精度. 在实验中, 我们选择了来自 Doop 中的多种不同精度的指针分析规则, 旨在证明我们的框架对于现有的 Doop 中丰富的指针分析规则的高度兼容性. 具体而言, 我们选择了如下几种精度: 上下文不敏感、k 对象敏感 (k=1 或 2)、k 调用点敏感 (k=1)、k 类型敏感 (k=1 或 2)、选择性对象敏感.

实验基准集. 我们的增量指针分析框架针对的是在变更频繁且单次变更的变化相对局部的场景. 按照这个原则, 我们选定的实验基准都是在实际中广泛使用的 Java 项目, 并且在其 GitHub 存储库中代码变更频繁, 如表 1 所示. 我们的实验都集中在它们的 master 分支上, 对从表 1 中记录的起始提交开始, 按照提交历史中的顺序对连续 20 次代码提交进行增量分析. 当然, 在实际的增量分析场景下我们可能会针对一次 PR 甚至一次发布进行增量分析, 但是我们同样可以将 PR 或者发布视为一次较大的代码提交.

我们对 5 个选定的 Java 项目做简要的介绍.

1) Jedis 是 Redis 的 Java 客户端, 专为性能和易用性而设计.

2) ErrorProne 是 Java 的静态分析工具, 可在编译时捕获常见的编程错误.

3) ZooKeeper 是一个分布式协调服务, 用于管理和维护分布式系统中的配置、命名等信息.

© 中国科学院软件研究所 http://www.jos.org.cn

4) PMD 是一个源代码分析器, 用于检测和识别 Java 代码中的潜在问题和不规范编码风格.

5) CheckStyle 是一个用于检查 Java 源代码是否符合代码标准或验证规则集的工具.

表 1　实验数据集

Project	init	#commits	freq	LOC	ΔLOC (avg/max)	Δmet (avg/max)	Δfile (avg/max)
Jedis	5359c37	2 248	3 per day	102k	266/2 692	45/387	6/27
ErrorProne	3dd2abc	6 056	1 per day	317k	104/924	11/77	3/21
ZooKeeper	5b6823a	2 504	1 per week	184k	122/735	11/74	5/31
PMD	792fe44	26 317	10 per week	224k	151/1 720	2/21	4/33
CheckStyle	7007bef	12 794	10 per day	318k	270/2 027	7/73	7/23

3.2　RQ1: 性能

为了评估我们的 DDoop 增量指针分析框架在代码变更频繁的场景下的性能, 我们将 DDoop 增量框架与原始 Doop 框架进行对比. 具体而言, 对于 20 次连续代码提交编译生成的 20 个版本 jar 包, 我们的 DDoop 增量框架和原始 Doop 框架将会依次分析每一个版本, 并且统计对每一个版本的分析所需要的时间. 对于我们的增量分析框架, 对第 1 个版本 jar 包的分析是全量分析, 后续的 19 个版本则是增量分析; 而对于原始 Doop 框架, 它对于 20 个版本 jar 包的处理方式完全相同, 每一次都是从头开始全量分析.

图 7 展示了 DDoop 增量框架在开启了前端剪枝优化的情况下, 与原始 Doop 框架在 7 种指针分析精度设置、5 个基准项目上的分析耗时结果, 其中纵坐标采用了对数轴. 图 7 分别给出了 Doop 框架在 20 个版本 jar 包上的平均分析时间, DDoop 增量框架在第 1 个版本 jar 包上的全量分析时间以及在后续 19 个版本 jar 包上的增量分析平均时间. 从图 7 我们可以发现, 虽然 DDoop 增量框架在第 1 个版本 jar 包上的分析耗时要比原始 Doop 框架更多, 但是在后续的增量分析过程中, DDoop 增量框架的耗时相比于原始 Doop 框架显著减少. DDoop 的第 1 轮全量分析较慢主要包含以下两方面的原因: 首先, DDoop 增量框架前端需要记录增量信息, 这一过程会花费额外的时间. 这是因为我们的框架需要追踪和存储代码的变化, 以便在后续的分析中只关注这些变化部分. 这种增量信息的收集和管理会增加一定开销. 其次, DDoop 增量框架后端所使用的 DDlog 引擎本身比非增量的 Soufflé 引擎要慢. 这是因为 DDlog 需要额外的计算资源维护增量信息.

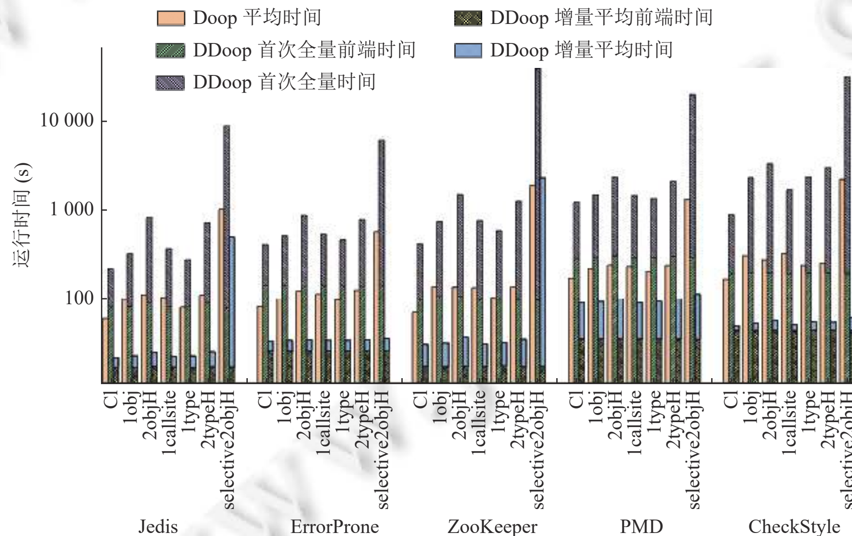

图 7　DDoop 与 Doop 的实际运行时间对比 (开启剪枝选项)

图 8 展示了 DDoop 增量框架在开启了前端剪枝优化的情况下, 相比原始 Doop 框架的加速比, 参考值表示加速比为 1. 从图 8 可以看出, 在增量分析场景下, DDoop 增量框架相比原始 Doop 框架的平均加速比约为 5. 在

© 中国科学院软件研究所 http://www.jos.org.cn

selective2objH 的精度设置下, DDoop 增量框架相比原始 Doop 框架的加速比最为明显. 在我们的实验评估中, DDoop 增量框架在 selective2objH 的精度设置下可实现最高约 36×的分析加速 (CheckStyle).

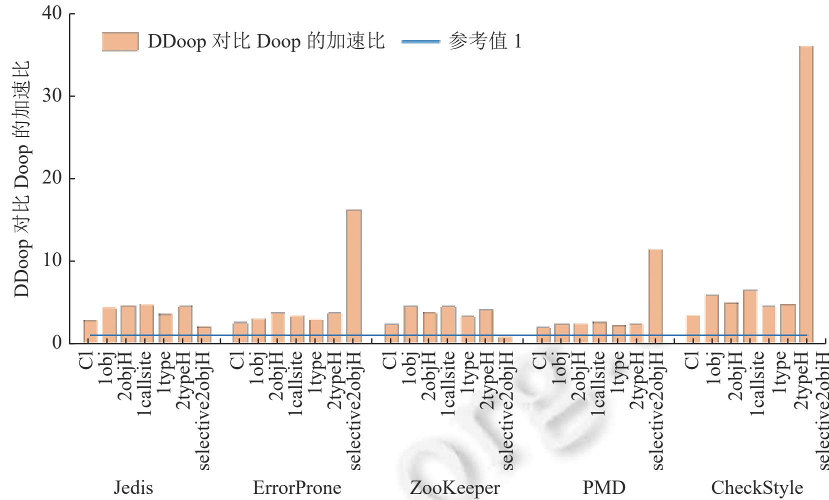

图 8　DDoop 对 Doop 的运行时间加速比

为了展示剪枝优化对分析效率的影响, 我们分别在图 7 (开启剪枝选项) 和图 9 (关闭剪枝选项) 中展示了 DDoop 增量框架在运行时的前端、后端更加细节的耗时信息. 我们可以发现, 在关闭剪枝优化选项时, 增量过程中前端会更加耗时, 关闭剪枝优化选项的前端耗时平均比开启剪枝优化选项要多 41.3%. 这让 DDoop 框架在增量时对比原版 Doop 框架的优势有所缩小. 此时, DDoop 增量框架相比原始 Doop 框架的平均加速比约为 4, 在 selective2objH 的精度设置下可实现最高约 27×的分析加速.

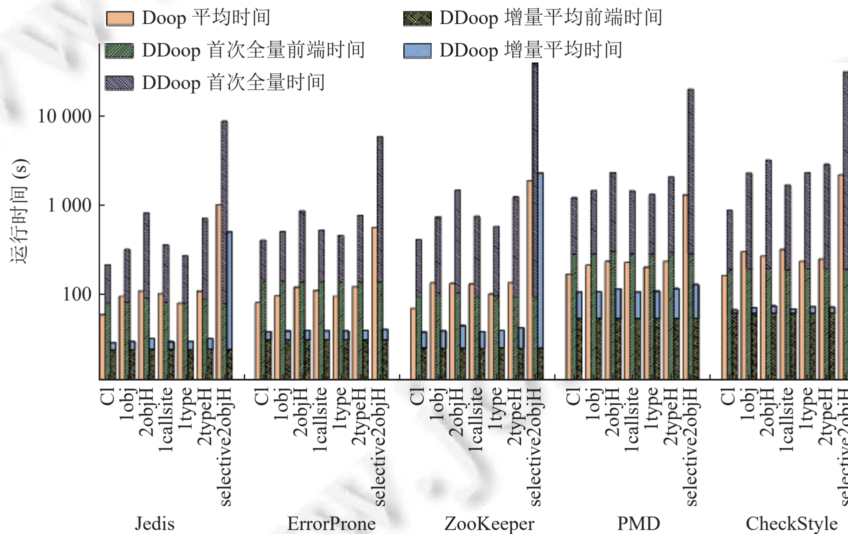

图 9　DDoop 与 Doop 的实际运行时间对比 (关闭剪枝选项)

需要注意的是, 增量分析并不能保证在任何场景下、对任何项目的变更都能提供明显的性能优势, 特别是对于可能影响到项目的较大范围的变更, 对其进行增量分析的耗时甚至可能会比重新全量分析更长. 在图 7 和图 9 中, 我们可以观察到这样一个稍显异常的例子: 在 ZooKeeper 代码库的 selective2objH 精度设置下, DDoop 增量框架在增量过程中的平均消耗时间略差于 Doop 框架, 并且主要的耗时集中在后端评估部分, 我们对应给出了这种

© 中国科学院软件研究所 http://www.jos.org.cn

情况下后端每次评估的耗时, 如图 10 所示. 从图 10 可以看出, 第 11 次提交对应的代码变更显著拉高了增量分析耗时的平均值. 分析此次提交对应的代码变更可以发现, 在这一次提交中, 代码的变化引发了上下文方面的大范围变化, 在相对更加复杂的上下文敏感性选项下, 这个问题会更加容易变得不可拓展. 但是从我们的实验评估中的所有 665 次评估数据 (7 种精度设置下 5 个项目各有 19 次增量分析) 来看, 这种异常情况出现的概率 (2/665) 并不高. 整体上看, DDoop 的增量化支持依旧可以在绝大多数场景下获得性能优势.

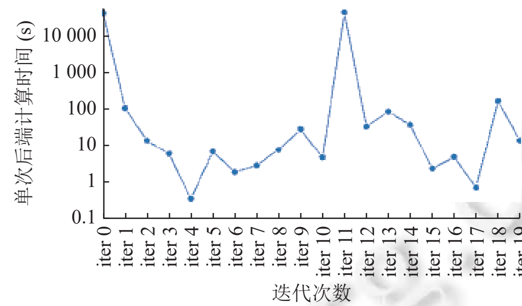

图 10 selective2objH 精度设置下 DDoop 在 ZooKeeper 上的后端耗时

实验结果表明, 我们的 DDoop 增量分析框架在代码频繁变更的场景下, 通过对代码变更进行增量分析, 在分析耗时方面能够实现比现有先进的 Doop 框架的全量分析的显著加速. 并且在变更越频繁, 变更次数更多的场景下, 我们的增量分析框架的累计耗时加速优势将更为显著.

3.3 RQ2: 兼容性

与 Doop 中现有指针分析规则兼容是 DDoop 增量框架设计中的一大核心目标. 为了评估 DDoop 增量框架对 Doop 框架中现有指针分析规则的兼容性, 在这部分的实验中, 我们将 DDoop 增量框架和两个现有的尝试为 Doop 框架提供基础增量支持的工作进行对比. 具体而言, 我们将 DDoop 框架与一个基于 Soufflé-elastic 实现基础增量支持的 Doop-SE 框架以及另一个初步整合了 DDlog 引擎的 Doop-DDlog 框架进行兼容性测试. 基于实验设置中选定的 7 种不同精度的指针分析规则以及 Doop 中提供的一种简化的指针分析规则 micro, 我们分别使用 DDoop 和 Doop-SE, 在我们的实验基准集中的 5 个真实世界大型 Java 项目上对 20 次连续的代码提交对应的 jar 包进行指针分析. 由于 Doop-DDlog 当前并无开放可获取的工具, 我们无法对其进行实际的实验评估, 因此我们这里采用了其论文[17]中报告的结果和结论.

表 2 展示了 DDoop、Doop-SE 和 Doop-DDlog 对 Doop 现有指针分析规则的兼容性情况. 从表 2 可以看出, 我们的 DDoop 能够兼容所有 8 种精度的指针分析规则, 而 Doop-SE 和 Doop-DDlog 都只能处理简化的 micro 分析规则, 在其他复杂分析规则上的兼容性测试均无法正确运行. 对于 Doop-SE, 就目前 Soufflé-elastic 的实现而言, 在面对 Doop 中的绝大多数指针分析规则时, 它并不能将规则文件正确的编译或是解释执行, 对报错信息和实现细节进行分析可以发现, 这个问题的直接原因是出在其中的规则重排器组件实现中. 此外, 对于 Doop 规则中的聚合函数 (例如 sum()), 它实际上也并没有提出有效的增量解决方案, 它只提供了 Soufflé 中有限的语法结构的增量能力. 对于 Doop-DDlog, 它试图在 Doop 框架中整合 DDlog 引擎以提供增量分析的能力, 然而和我们更为完善的规则重写器相比, 它只实现了从 Soufflé 到 DDlog 的简单转换, 并且明确指出了其只支持了 Doop 中最简单的“self-contained”分析和简化的 micro 分析. 此外, 根据其实验评估, 它只是在每一轮分析的时候将 Soufflé 引擎换成 DDlog 引擎, 实际运行的还是完整评估, 并未有效地利用 DDlog 引擎的增量能力.

表 2 实验数据集

Analysis framework	CI	1obj	2objH	1callsite	1type	2typeH	selective2objH	micro
DDoop	√	√	√	√	√	√	√	√
Doop-SE	×	×	×	×	×	×	×	√
Doop-DDlog	×	×	×	×	×	×	×	√

兼容性测试的实验评估结果表明, 我们的 DDoop 能够完全兼容所有精度的指针分析规则, 而 Doop-SE 和 Doop-DDlog 都只能处理简化的 micro 分析规则, 在其他复杂分析规则上均无法做到正确兼容. 此外, 由于我们的规则重写器的通用性和完备性, 除了实验评估中用到的指针分析规则, Doop 中其他精度的指针分析规则我们也能够兼容, 并且对于未来在 Doop 基础上开发的程序分析规则也能够继续提供兼容性.

3.4 RQ3: 精度

为了评估我们的增量指针分析框架前端采取的剪枝策略对最终分析结果的影响, 在这部分的实验中, 我们会分别启用和关闭前端增量生成过程中的剪枝选项进行对比. 具体而言, 对于每一个不同的待分析程序和分析精度的组合, 我们会分别运行两次依次分析每一个 jar 包的增量分析, 其中一次会在前端生成事实时开启剪枝, 另一次则不开启剪枝. 在最后, 我们将统计两者最后分析结果的差异.

表 3 展示了前端剪枝选项在 5 个基准项目、7 种指针分析规则上对指针分析结果精度的影响. 其中使用了 3 种指标#vpt、#fpt 和#call-edge 来度量指针分析结果的精度, 分别对应 Doop 框架中定义的 3 种关系: VarPointsTo、InstanceFieldPointsTo 和 CallGraphEdge. 需要注意的是, 在 Doop 框架定义中, 这些关系中引入了上下文信息, 为了符合对指针分析算法精度评估的一般惯例, 我们使用一个简单的数据过滤程序过滤了其中仅上下文不同的数据, 得到了表 3 中现在的结果. 可以看出, 在我们实验基准集上, 在增量输入事实生成阶段引入剪枝策略在除了 Jedis 项目以外的其他仓库上都没有影响以上 3 个精度指标. 而在 Jedis 项目中, 剪枝后的版本在所有精度设置下的精度指标都略微差于剪枝前, 整体差距约在 2% 左右.

表 3 剪枝选项对指针分析精度的影响

Project	Analysis	Precision metrics (unpruned)			Precision metrics (pruned)		
		#vpt	#fpt	#call-edge	#vpt	#fpt	#call-edge
Jedis	CI	1624685	148313	55137	1661612	150356	55496
	1obj	899178	54102	50983	921445	54721	51309
	2objH	416994	34655	47565	421017	34832	47825
	1callsite	1112252	90253	52216	1136813	91134	52550
	1type	1045098	73262	52087	1069259	74132	52449
	2typeH	417744	34655	47568	421767	34832	47828
	selective2objH	542573	42007	48477	552705	42336	48791
ErrorProne	CI	1349996	121779	50993	1349996	121779	50993
	1obj	783850	48511	47363	783850	48511	47363
	2objH	372583	31409	43993	372583	31409	43993
	1callsite	927188	73538	48284	927188	73538	48284
	1type	904739	64988	48448	904739	64988	48448
	2typeH	373315	31409	43996	373315	31409	43996
	selective2objH	492411	37874	44873	492411	37874	44873
ZooKeeper	CI	2713829	217008	66106	2713829	217008	66106
	1obj	1505314	72831	61765	1505314	72831	61765
	2objH	563567	41641	55098	563567	41641	55098
	1callsite	1731519	121498	62693	1731519	121498	62693
	1type	1618749	95138	62130	1618749	95138	62130
	2typeH	575060	41770	55204	575060	41770	55204
	selective2objH	762910	50018	56091	762910	50018	56091
PMD	CI	2789814	204493	63638	2789814	204493	63638
	1obj	1589995	72821	59057	1589995	72821	59057
	2objH	607586	44393	52482	607586	44393	52482
	1callsite	1755897	106580	59658	1755897	106580	59658
	1type	1702373	94118	59731	1702373	94118	59731
	2typeH	623805	44860	52537	623805	44860	52537
	selective2objH	924034	59498	54239	924034	59498	54239

表 3　剪枝选项对指针分析精度的影响 (续)

Project	Analysis	Precision metrics (unpruned)			Precision metrics (pruned)		
		#vpt	#fpt	#call-edge	#vpt	#fpt	#call-edge
CheckStyle	CI	4093997	316226	84567	4093997	316226	84567
	1obj	2339198	115218	77897	2339198	115218	77897
	2objH	713208	55539	68782	713208	55539	68782
	1callsite	2721068	170652	79733	2721068	170652	79733
	1type	2531744	135910	78952	2531744	135910	78952
	2typeH	717783	55624	68887	717783	55624	68887
	selective2objH	1066552	75882	71261	1066552	75882	71261

实验评估结果表明, 虽然在增量输入事实生成阶段引入剪枝策略可能会最终的指针分析结果带来轻微的精度损失, 但影响极小, 我们设计的剪枝策略在现有的增量指针分析场景中仍可以保持相当高的精度.

4 讨　论

4.1 内存占用

静态程序分析没有银弹, 一种好的静态分析技术必然要在多个相互制衡的方面之间做出权衡取舍. 增量分析虽然在分析效率上具有显著优势, 但这意味着其必然需要在其他方面付出一些代价——对于非增量的静态分析技术而言, 在分析过程中, 它们往往可以通过删除在后续分析中不再需要的中间结果来节省内存使用; 而增量分析在分析的全流程中必须维护所有与最终结果相关的中间结果, 以用于下一轮的增量分析, 这会导致增量分析的内存使用量大幅增加. 目前, 我们的 DDoop 增量框架也存在内存占用相对较高的问题. 虽然实验已经证明其在流行的 Java 项目 (如 Jedis) 上能支持 Doop 中的许多指针分析精度, 但与原始的 Doop 相比, DDoop 的内存使用量仍然是其数倍. 在这种情况下, 尽管我们在分析耗时上将可扩展性向前推进了一步, 但当前的内存问题成为了阻止我们的 DDoop 框架进一步分析更大规模 (数百万行) 程序的主要障碍.

4.2 精度影响

在我们的框架前端中, 为了提高增量输入事实生成的效率, 引入了剪枝优化技术. 然而, 在提升输入事实生成速度的同时, 剪枝优化技术也给指针分析的精度带来了挑战. 尽管我们的框架在第 1 轮全量生成中处理所有可能引用的类库, 确保了初始的精度, 但具体到后续的增量事实生成前端中的剪枝操作, 精度的问题开始浮现. 具体而言, 我们的剪枝操作会将基于类依赖关系加载类的过程“截断”到类依赖库类中的第 1 层, 这样, 如果程序的改变并未导致引用类库的变化, 或者只是停止引用某些类库, 那么这种剪枝操作不会对精度产生负面影响. 这种情况下, 剪枝可以看作是一种优化策略, 既提高了分析效率, 又保持了精度. 然而, 当程序的变更引发新的依赖库类引用时, 剪枝操作可能会对精度产生负面影响. 因为这些新引用的依赖库类在静态分析层面对指针流数据流的完整建模可能需要额外载入更多的依赖库类, 而剪枝操作已经将其截断, 于是我们的静态分析算法就无法通过获取经过库函数相关的数据流建模, 从而影响精度. 在实验评估中, 我们也确实观察到了剪枝操作对精度的影响, 但总体上精度损失非常细微, 处于可控范围之内. 此外, 在静态代码扫描的设置下, 更注重于尽早发现问题, 并快速地提供结果, 可以接受一定程度的误报或漏报, 对分析精度的要求并不如程序验证那么苛刻.

4.3 潜在适用性

我们的 DDoop 增量框架是针对 Java 指针分析框架 Doop 设计的, 因此, 目前的 DDoop 增量框架实现无法针对其他更加广泛的编程语言编写的程序运行增量分析. 但是, DDoop 框架在理论上具有对于这些使用非 Java 语言编写的程序的潜在分析能力. 具体而言, DDoop (Doop) 框架的指针分析在原理上就是将 Andersen 指针分析在 PAG 上计算不动点的问题转换为了在 Datalog 程序中不断迭代生成新的 facts 以达到不动点的问题. 显然, 从方法的角度上来说, Andersen 算法部分是平台无关的, 只是在实现框架的过程中, 不同的语言对应的具体的和语言特性

© 中国科学院软件研究所 http://www.jos.org.cn

相关的工程实现细节和编程语言相关. 因此, DDoop 框架拥有对如 C++, Rust, Python 等其他语言的指针分析能力的潜在适应性. 这为 DDoop 框架的未来发展提供了广阔的可能性. 此外, 我们的增量框架技术本身是框架无关的, 可以应用于其他基于 Datalog 的分析框架. 因此我们的技术具有更广阔的潜在适用性.

5 相关工作

在本节中, 我们主要通过以下 3 个方面来介绍和讨论我们的增量指针分析框架的相关工作: (1) 指针分析框架; (2) 增量分析算法; (3) 增量计算.

5.1 指针分析框架

鉴于指针分析的基础分析地位及其重要性, 指针分析研究人员已经开发了一系列指针分析框架, 包含已有指针分析算法的实现以及支持新的指针分析算法的开发.

Soot[22]中提供了两个指针分析框架 SPARK[23]和 Paddle[24], 其中 SPARK 是上下文不敏感的, 而 Paddle 支持多种基于标签的上下文敏感指针分析. WALA[25]提供了一个 Andersen 风格的流不敏感指针分析框架. 目前已在其基础上开发了一些新的指针分析工作, 例如 IPA[9]和 SHARP[10]. Doop[11]是一个基于 Datalog 的声明式指针分析框架, 其支持现有大多数指针分析算法, 被认为是当前最主流的指针分析框架. Qilin[26]是一个在 Soot 基础上开发的指针分析框架, 其支持各种细粒度的 Java 上下文敏感指针分析. Tai-e[27]是最新提出的一个 Java 程序分析框架, 其精心设计了一个易学、易用、高效的指针分析框架和基于指针分析的分析插件系统. CodeQL[28]是一个语义代码分析引擎, CodeQL 支持对包括 Java 在内 11 种语言进行静态分析, 提供了基于 Steensgaard 算法的指针分析功能, 其底层分析同样依赖于 Datalog 实现. 此外, 与本工作的思路类似, CodeQL 近期通过更换后端 Datalog 引擎, 实现了一个支持增量分析的增量 CodeQL 原型框架[29].

目前的这些主流的指针分析框架主要面向的是全程序指针分析, 对增量指针分析的支持较弱. 据我们所知, 其中仅有在 Soot[22]之上基于 CFL 可达性的增量指针分析[30], 以及在 WALA[25]之上的 IPA[9]和 SHARP[10]等少数工作提出了增量指针分析算法, 而且它们面向的是特定指针分析算法的增量化, 并未从指针分析框架的层面上提供增量分析的能力. 而 CodeQL 的原型增量框架虽然从框架层面支持了增量能力, 然而由于其算法层面仅提供了基础的指针分析能力, 无法兼容和复用 Doop 中现有的大量指针分析算法实现, 特别是上下文敏感对指针分析精度至关重要的部分. 本文提出的增量指针分析框架 DDoop 则是不仅从分析框架的层面解决了指针分析的增量化问题, 并且能够完全兼容 Doop 框架中现有已实现的各种上下文敏感性的大量指针分析算法.

5.2 增量分析算法

由于能够响应代码变更快速提供最新分析结果的优势, 近年来增量分析在静态分析领域受到了极大的关注. 针对各种静态分析技术, 研究人员分别提出了相应的增量化方案.

Reviser[31]是一种针对基于 IDE/IFDS 的数据流分析框架的增量分析技术, 能够快速响应程序的增量变更. Pacak 等人[32]和 Zwaan 等人[33]各自分别提出了自动推导增量类型检查器的工作, 能够在 IDE 等实时场景下对代码变更立即产生反馈. Lu 等人[30]基于 CFL 可达性提出了一种增量指针分析算法. Liu 等人[9,10]针对上下文不敏感指针分析和上下文敏感指针分析分别提出了相应的增量化方案. Zhao 等人[34]在 CHA 调用图构建算法的基础上提供了一种增量构建算法.

大多数现有的增量分析算法都源自 DRed (deletion-rederivation) 算法[35]. 然而, 不同的分析算法中需要进行增量处理的部分以及增量化的细节不尽相同, 这就导致需要为每一种静态分析算法单独设计相应的增量算法, 这个过程非常冗长且极易出错. 此外, 基于 DRed 的增量分析算法通常会存在过度删除 (over-deletion) 的问题[36], 即大量数据可能先被删除, 但随后又被重新派生, 这可能会带来比较大的性能问题.

与这些传统的在分析算法层面进行增量化的工作相比, 我们的增量指针分析框架没有牺牲分析算法的透明度, 可以兼容并复用现有的分析算法. 因此, 这种方式可以解决一大类 (可以用 Datalog 表述的) 程序分析算法的增量化问题, 而无需为每个具体的分析问题实现特定的增量化方案. 此外, 在我们基于 DDlog 的增量分析框架中, 也

© 中国科学院软件研究所 http://www.jos.org.cn

不存在 DRed 中的过度删除问题, 在这方面可以相比 DRed 可以取得更好的性能.

5.3 增量计算

随着输入变化而高效地更新计算输出, 而无需从头开始重新计算, 这一点对于许多应用来说可能很重要, 甚至是必要的. 增量计算的基本想法是在静态算法执行期间维护某些信息, 这些信息可用于在输入变更时高效地更新输出. 增量计算能够以一种对上层算法透明的方式为算法提供增量化的支持. 增量计算的早期工作包括静态依赖图[37], 函数缓存或记忆化[38], 动态依赖图[39]等. 而将动态依赖图与记忆化相结合的自调整计算 (self-adjusting computation)[40], 显著扩展了增量计算的适用性. Incoop[41]是自调整计算技术的代表工作之一, 它是一种基于 MapReduce 的通用增量计算框架, 无需更改一行应用程序代码即可显著提高性能.

差分计算 (differential computation)[42]是由微软提出的一种新的增量计算模型, 以高效地支持迭代查询, 它扩展了传统的增量计算以允许任意嵌套迭代. 基于差分计算模型的差分数据流 (differential dataflow, DDF) 现在是增量 Datalog 引擎的主流实现方式, 如 DDlog[17], Laddder[36]等. 我们的增量指针分析框架即是基于 DDlog 引擎来提供增量计算支持, 而无需对上层的指针分析算法进行调整.

另一类增量计算模型是增量图计算[43]. GraphBolt[44]通过研究依赖驱动处理的方式来进行增量图计算. Chronos[45]是一个专门为在时间图上运行内存中迭代图计算而设计和优化的图计算引擎, 其设计探索了局部性、并行性和增量计算之间有趣的相互作用. iGraph[46]是一个针对持续更新的动态图的增量图处理系统. 静态分析中的很多分析技术都是基于图形式进行计算的, 如数据流分析依赖于控制流图, 指针分析依赖于指针赋值图, 过程间分析需要调用图. 因此增量图计算可能是未来的增量程序分析的一个重要的可探索方向. 目前 Gu 等人[47]在这个方向进行了初步的探索, 提出了 BigSpa, 这是一个结合了离线批量静态程序分析和在线增量静态程序分析的大数据图处理系统, 对于小批量更新可以实现近乎实时的增量分析.

6 总结与展望

本文设计并实现了一个增量指针分析框架 DDoop, 旨在解决现有指针分析技术在处理大规模程序的连续代码提交时耗时过长的问题. 基于带增量评估机制的 DDlog 引擎, DDoop 框架通过前端输入事实增量生成技术以及后端兼容 Doop 中指针分析规则的自动化增量分析程序生成技术, 能够高效地处理代码变更, 尽可能地复用上一次分析的结果, 从而降低计算量, 大大节省了指针分析过程所消耗的时间. 实验评估展示, 相较于非增量式的原始 Doop 框架, 我们的增量框架可以获得平均约 5×, 最高达 36×的加速效果.

本文的方法和框架实现目前仍存在一些不足, 这也是我们未来的研究方向: 增量分析的内存消耗较大, 我们未来计划结合 DDlog 特性进一步优化重写器, 以及探索基于磁盘的中间结果存储策略, 以降低内存使用量; 此外, 我们的输入事实增量生成的前端采取的剪枝优化技术会带来细微的精度损失, 未来我们计划对优化技术进一步改进, 以消除精度损失. 我们的增量分析框架不仅探索了对指针分析技术的透明增量化支持, 同时也支持其他能基于 Datalog 表述的一大类静态分析技术, 这将为软件开发 CI/CD 流程中的静态代码扫描提供了更加高效的解决方案可能性. 除差分计算之外, 增量图计算也是一类重要的增量计算模型, 我们未来也计划探索基于增量图计算的静态分析透明增量化技术.

References:

[1] Cai YD, Yao PS, Zhang C. Canary: Practical static detection of inter-thread value-flow bugs. In: Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation. New York: ACM, 2021. 1126–1140. [doi: 10.1145/3453483.3454099]

[2] Grech N, Smaragdakis Y. P/Taint: Unified points-to and taint analysis. Proc. of the ACM on Programming Languages, 2017, 1(OOPSLA): 102. [doi: 10.1145/3133926]

[3] Pradel M, Jaspan C, Aldrich J, Gross TR. Statically checking API protocol conformance with mined multi-object specifications. In: Proc. of the 34th Int'l Conf. on Software Engineering (ICSE). Zurich: IEEE, 2012. 925–935. [doi: 10.1109/ICSE.2012.6227127]

[4] Zhang QR, Lyu MR, Yuan H, Su ZD. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In: Proc. of the 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation. New York: ACM, 2013. 435–446. [doi: 10.1145/

2491956.2462159]
[5] Smaragdakis Y, Balatsouras G. Pointer analysis. Foundations and Trends® in Programming Languages, 2015, 2(1): 1–69. [doi: 10.1561/2500000014]
[6] Tan T, Li Y, Xue JL. Efficient and precise points-to analysis: Modeling the heap by merging equivalent automata. In: Proc. of the 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Barcelona: ACM, 2017. 278–291. [doi: 10.1145/3062341.3062360]
[7] Li Y, Tan T, Møller A, Smaragdakis Y. Scalability-first pointer analysis with self-tuning context-sensitivity. In: Proc. of the 26th ACM Joint Meeting on European Software Engineering Conf. and the Symp. on the Foundations of Software Engineering. Lake Buena Vista: ACM, 2018. 129–140. [doi: 10.1145/3236024.3236041]
[8] Liu BH, Zhang H, Dong LM. Survey on state of DevOps in China. Ruan Jian Xue Bao/Journal of Software, 2019, 30(10): 3206–3226 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5796.htm [doi: 10.13328/j.cnki.jos.005796]
[9] Liu BZ, Huang J, Rauchwerger L. Rethinking incremental and parallel pointer analysis. ACM Trans. on Programming Languages and Systems, 2019, 41(1): 6. [doi: 10.1145/3293606]
[10] Liu BZ, Huang J. SHARP: Fast incremental context-sensitive pointer analysis for Java. Proc. of the ACM on Programming Languages, 2022, 6(OOPSLA1): 88. [doi: 10.1145/3527332]
[11] Bravenboer M, Smaragdakis Y. Strictly declarative specification of sophisticated points-to analyses. In: Proc. of the 24th ACM SIGPLAN Conf. on Object Oriented Programming Systems Languages and Applications. Orlando: ACM, 2009. 243–262. [doi: 10.1145/1640089.1640108]
[12] Li Y, Tan T, Møller A, Smaragdakis Y. Precision-guided context sensitivity for pointer analysis. Proc. of the ACM on Programming Languages, 2018, 2(OOPSLA): 1–29. [doi: 10.1145/3276511]
[13] Ma WJ, Yang SY, Tan T, Ma XX, Xu C, Li Y. Context sensitivity without contexts: A cut-shortcut approach to fast and precise pointer analysis. Proc. of the ACM on Programming Languages, 2023, 7(PLDI): 128. [doi: 10.1145/3591242]
[14] Ryzhyk L, Budiu M. Differential Datalog. In: Proc. of the 3rd Int'l Workshop on the Resurgence of Datalog in Academia and Industry Co-located with the 15th Int'l Conf. on Logic Programming and Nonmonotonic Reasoning. Philadelphia: CEUR-WS.org, 2019. 56–67.
[15] Jordan H, Scholz B, Subotić P. Soufflé: On synthesis of program analyzers. In: Proc. of the 28th Int'l Conf. on Computer Aided Verification. Toronto: Springer, 2016. 422–430. [doi: 10.1007/978-3-319-41540-6_23]
[16] Zhao D, Subotic P, Raghothaman M, Scholz B. Towards elastic incrementalization for datalog. In: Proc. of the 23rd Int'l Symp. on Principles and Practice of Declarative Programming. New York: ACM, 2021. 20. [doi: 10.1145/3479394.3479415]
[17] Ritsogianni AA. Incremental static analysis with differential Datalog [BS. Thesis]. Athens: University of Athens, 2019.
[18] Tan T, Ma XX, Xu C, Ma CY, Li Y. Survey on Java pointer analysis. Journal of Computer Research and Development, 2023, 60(2): 274–293 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.202220901]
[19] Whaley J, Avots D, Carbin M, Lam MS. Using Datalog with binary decision diagrams for program analysis. In: Proc. of the 3rd Asian Symp. on Programming Languages and Systems. Tsukuba: Springer, 2005. 97–118. [doi: 10.1007/11575467_8]
[20] Antoniadis T, Triantafyllou K, Smaragdakis Y. Porting doop to Soufflé: A tale of inter-engine portability for Datalog-based analyses. In: Proc. of the 6th ACM SIGPLAN Int'l Workshop on State of the Art in Program Analysis. Barcelona: ACM, 2017. 25–30. [doi: 10.1145/3088515.3088522]
[21] Use of Zipper in Doop. 2021. https://bitbucket.org/yanniss/doop/issues/41/use-of-zipper
[22] Vallée-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V. Soot—A Java bytecode optimization framework. In: Proc. of the 1999 Conf. of the Centre for Advanced Studies on Collaborative research. Mississauga: IBM Press, 1999. [doi: 10.5555/781995.782008]
[23] Lhoták O, Hendren L. Scaling Java points-to analysis using SPARK. In: Proc. of the 12th Int'l Conf. on Compiler Construction. Warsaw: Springer, 2003. 153–169. [doi: 10.1007/3-540-36579-6_12]
[24] Lhoták O, Hendren L. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. ACM Trans. on Software Engineering and Methodology, 2008, 18(1): 3. [doi: 10.1145/1391984.1391987]
[25] WALA. Watson Libraries for Analysis (WALA). 2023. https://wala.sourceforge.net/
[26] He DJ, Lu JB, Xue JL. Qilin: A new framework for supporting fine-grained context-sensitivity in Java pointer analysis. In: Proc. of the 36th European Conf. on Object-oriented Programming (ECOOP 2022). Berlin: Leibniz-Zentrum für Informatik, 2022. 30. [doi: 10.4230/LIPIcs.ECOOP.2022.30]
[27] Tan T, Li Y. Tai-e: A developer-friendly static analysis framework for Java by harnessing the good designs of classics. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 1093–1105. [doi: 10.1145/3597926.3598120]
[28] CodeQL. 2024. https://codeql.github.com/

© 中国科学院软件研究所 http://www.jos.org.cn

[29] Szabó T. Incrementalizing production CodeQL analyses. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 1716–1726. [doi: 10.1145/3611643.3613860]
[30] Lu Y, Shang L, Xie XW, Xue JL. An incremental points-to analysis with CFL-reachability. In: Proc. of the 22nd Int'l Conf. on Compiler Construction. Rome: Springer, 2013. 61–81. [doi: 10.1007/978-3-642-37051-9_4]
[31] Arzt S, Bodden E. Reviser: Efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In: Proc. of the 36th Int'l Conf. on Software Engineering. Hyderabad: ACM, 2014. 288–298. [doi: 10.1145/2568225.2568243]
[32] Pacak A, Erdweg S, Szabó T. A systematic approach to deriving incremental type checkers. Proc. of the ACM on Programming Languages, 2020, 4(OOPSLA): 127. [doi: 10.1145/3428195]
[33] Zwaan A, Van Antwerpen H, Visser E. Incremental type-checking for free: Using scope graphs to derive incremental type-checkers. Proc. of the ACM on Programming Languages, 2022, 6(OOPSLA2): 140. [doi: 10.1145/3563303]
[34] Zhao ZL, Wang XZ, Xu ZG, Tang ZH, Li YC, Di P. Incremental call graph construction in industrial practice. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering: Software Engineering in Practice (ICSE-SEIP). Melbourne: IEEE, 2023. 471–482. [doi: 10.1109/ICSE-SEIP58684.2023.00048]
[35] Gupta A, Mumick IS, Subrahmanian VS. Maintaining views incrementally. ACM SIGMOD Record, 1993, 22(2): 157–166. [doi: 10.1145/170036.170066]
[36] Szabó T, Erdweg S, Bergmann G. Incremental whole-program analysis in Datalog with lattices. In: Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation. New York, ACM, 2021. 1–15. [doi: 10.1145/3453483.3454026]
[37] Demers A, Reps T, Teitelbaum T. Incremental evaluation for attribute grammars with application to syntax-directed editors. In: Proc. of the 8th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. Williamsburg: ACM, 1981. 105–116. [doi: 10.1145/567532.567544]
[38] Pugh W, Teitelbaum T. Incremental computation via function caching. In: Proc. of the 16th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. Austin: ACM, 1989. 315–328. [doi: 10.1145/75277.75305]
[39] Carlsson M. Monads for incremental computing. ACM SIGPLAN Notices, 2002, 37(9): 26–35. [doi: 10.1145/583852.581482]
[40] Anderson D, Blelloch GE, Baweja A, Acar UA. Efficient parallel self-adjusting computation. In: Proc. of the 33rd ACM Symp. on Parallelism in Algorithms and Architectures. New York: ACM, 2021. 59–70. [doi: 10.1145/3409964.3461799]
[41] Bhatotia P, Wieder A, Rodrigues R, Acar UA, Pasquin R. Incoop: MapReduce for incremental computations. In: Proc. of the 2nd ACM Symp. on Cloud Computing. Cascais: ACM, 2011. 7. [doi: 10.1145/2038916.2038923]
[42] McSherry F, Murray DG, Isaacs R, Isard M. Differential dataflow. In: Proc. of the 6th Biennial Conf. on Innovative Data Systems Research. Asilomar: CIDR, 2013.
[43] Fan WF, Liu MY, Tian C, Xu RQ, Zhou JR. Incrementalization of graph partitioning algorithms. Proc. of the VLDB Endowment, 2020, 13(8): 1261–1274. [doi: 10.14778/3389133.3389142]
[44] Mariappan M, Vora K. GraphBolt: Dependency-driven synchronous processing of streaming graphs. In: Proc. of the 14th EuroSys Conf. Dresden: ACM, 2019. 25. [doi: 10.1145/3302424.3303974]
[45] Han WT, Miao YS, Li KW, Wu M, Yang F, Zhou LD, Prabhakaran V, Chen WG, Chen EH. Chronos: A graph engine for temporal graph analysis. In: Proc. of the 9th European Conf. on Computer Systems. Amsterdam: ACM, 2014. 1. [doi: 10.1145/2592798.2592799]
[46] Ju WY, Li JX, Yu WR, Zhang RC. iGraph: An incremental data processing system for dynamic graph. Frontiers of Computer Science, 2016, 10(3): 462–476. [doi: 10.1007/s11704-016-5485-7]
[47] Gu R, Zuo ZQ, Jiang X, Yin H, Wang ZK, Wang LZ, Li XD, Huang YH. Towards efficient large-scale interprocedural program static analysis on distributed data-parallel computation. IEEE Trans. on Parallel and Distributed Systems, 2021, 32(4): 867–883. [doi: 10.1109/TPDS.2020.3036190]

附中文参考文献:

[8] 刘博涵, 张贺, 董黎明. DevOps 中国调查研究. 软件学报, 2019, 30(10): 3206–3226. http://www.jos.org.cn/1000-9825/5796.htm [doi: 10.13328/j.cnki.jos.005796]
[18] 谭添, 马晓星, 许畅, 马春燕, 李樾. Java 指针分析综述. 计算机研究与发展, 2023, 60(2): 274–293. [doi: 10.7544/issn1000-1239.202220901]

© 中国科学院软件研究所 http://www.jos.org.cn

沈天琪(1999－), 男, 硕士生, CCF 学生会员, 主要研究领域为程序分析.

宾向荣(2000－) 男, 博士生, CCF 学生会员, 主要研究领域为程序分析.

王熙灶(1995－) 男, 博士生, CCF 学生会员, 主要研究领域为程序分析与验证, 程序设计语言.

卜磊(1983－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为模型检验, 形式化方法, 信息物理系统, 复杂软件分析与验证.

www.jos.org.cn
www.jos.org.cn