无上下文的上下文敏感性

一种快速且精确的指针分析 Cut-Shortcut 方法

WENJIE MA* 和 SHENGYUAN YANG*，南京大学，中国TIAN TAN†，
XIAOXING MA、CHANG XU 和 YUE LI†，南京大学，中国

在过去几十年中，上下文敏感性一直被认为是提高 Java 指针分析精度的最有效理念之一。与极其快速的上下文不敏感方法不同，上下文敏感性要求在不同上下文下分析每个程序方法，以区分该方法的变量和堆对象在不同动态实例化中的静态抽象，从而减少由方法调用引入的虚假对象流。然而，尽管上下文敏感性能够显著提升精度，但由于每个方法都要在每个上下文下分别克隆和分析，因此会带来巨大的效率开销。近年来，为了将指针分析扩展到大型且复杂的 Java 程序，人们提出了许多选择性上下文敏感方法：仅对选定的方法应用上下文，而对其余方法进行上下文不敏感地分析。然而，由于这些选择性方法并未从根本上改变上下文敏感性的主要方法（因而也没有消除其效率瓶颈），它们虽然显著改善了结果，但仍然存在局限。

在这项工作中，我们提出了一种名为 Cut-Shortcut 的全新方法，用于对 Java 进行快速而精准的指针分析。其核心思想很简单：在不同上下文中克隆方法，主要是为了过滤被合并到被调用方法中的虚假对象流；从典型指针流图（PFG）的角度来看，可以通过截断（Cut）那些导致特定指针精度损失的边，并在 PFG 上绕过该方法，直接从源指针向目标指针添加快捷边，来模拟这种效果。因此，我们无需上下文也能实现上下文敏感性的效果。我们识别出三种通用程序模式，并据此开发算法，以便在 PFG 上安全地截断边并添加快捷边；我们对这些模式进行了形式化描述，并正式证明了其可靠性。为全面验证 Cut-Shortcut 的有效性，我们针对两种先进的 Java 指针分析框架实现了 Cut-Shortcut 的两个版本：一个用 Datalog 编写，适用于声明式的 Doop；另一个用 Java 编写，适用于命令式的 Tai-e。我们还考察了近期文献中使用的、符合实验要求的所有大型复杂程序。评估结果极具前景：在两个框架中，对于大多数受评估程序，Cut-Shortcut 的运行速度甚至快于上下文不敏感分析，同时还能达到较高精度，与上下文敏感分析（若其具备可扩展性）的精度相当。这是我们首次能够针对那些难以分析的程序，实现如此出色的效率与精度权衡；我们希望 Cut-Shortcut 能为未来开发更有效的 Java 指针分析提供新的思路。

CCS 概念：• **计算理论 → 程序分析**。

附加关键词和短语：指针分析、别名分析、上下文敏感性、Java

*两位作者对本文的贡献相同。
†通讯作者。

作者地址：Wenjie Ma，191250103@smail.nju.edu.cn；Shengyuan Yang，mf21320184@smail.nju.edu.cn，新软件技术国家重点实验室，南京大学，中国；Tian Tan，tiantan@nju.edu.cn；Xiaoxing Ma，xxm@nju.edu.cn；Chang Xu，changxu@nju.edu.cn；Yue Li，yueli@nju.edu.cn，新软件技术国家重点实验室，南京大学，中国。

允许为个人或课堂使用而免费制作本作品的部分或全部内容的数字或纸质副本，但不得为营利或商业利益制作或分发副本，且副本必须附有本声明及首页上的完整引文。本作品中第三方组件的版权必须得到尊重。如需用于其他用途，请联系所有者/作者。© 2023 版权所有者/作者保留版权。2475-1421/2023/6-ART128https://doi.org/10.1145/3591242

ACM 引用格式：
Wenjie Ma、Shengyuan Yang、Tian Tan、Xiaoxing Ma、Chang Xu 和 Yue Li。2023年。无上下文的上下文敏感性：一种快速且精确的指针分析 Cut-Shortcut 方法。*ACM*程序设计语言会议录7，PLDI，第 128 号文章（2023年6月），26 页。https://doi.org/10.1145/3591242

1 引言

指针分析是一类静态分析技术，旨在计算程序指针在程序执行期间可能指向的一组抽象值。指针分析的结果为各种静态分析应用程序奠定了基础，例如错误检测 [Cai 等人，2021年；Chandra 等人，2009年；Naik 等人， 2006], 安全分析 [Arzt 等人，2014年；Grech 和 Smaragdakis2017], 程序优化 [Sridharan 和 Bodík，2006年；Zhang 等人， 2013], 验证 [Fink 等，2008 年；Pradel 等人，2012], 以及理解 [Li 等人，2016年；Sridharan 等人， 2007]。因此，研究人员在过去几十年中投入大量精力，尝试构建快速且精确的指针分析算法，这并不令人意外 [Smaragdakis 和 Balatsouras 2015]。

Andersen 风格指针分析 [Andersen 1994] 被认为是 Java 的标准上下文不敏感指针分析 [Smaragdakis 和 Balatsouras，2015；Sridharan 等人 2013]。我们可以将其视为一种应用于典型指针流图（PFG）的迭代算法 [Li 等，2018a，2020a；Tonella 和 Potrich，2005；WALA 2006] 其中，节点表示程序指针（包括变量和实例字段），边表示指针指向集合之间的子集约束。在分析过程中，抽象对象沿着PFG的边传播。由于上下文不敏感方法无法区分同一方法的不同调用点，被调用方中的输入指向集合会被合并；因此，这种方法速度极快，但精度较低。

为了提高精度，指针分析采用了上下文敏感性。基本上，对于每个对被调用方（例如 m）的方法调用（例如 c），m 中的所有元素（如变量、实例字段和堆对象）都会被克隆，并在与某个程序元素（例如 e）相关的上下文中进行分析，该元素可将 c 与其他调用区分开来；据此，出现了不同形式的上下文敏感性，例如调用点敏感性（若 e 是一个调用点） [Sharir 和 Pnueli，1981；Shivers 1991], 对象敏感性（若 e 是一个分配点） [Milanova 等人 2002, 2005], 以及类型敏感性（若 e 是一种类型） [Smaragdakis 等人 2011]。由于在被调用方中合并的对象流会在不同上下文中分离，因此可以过滤掉方法调用引入的伪对象流，从而提高精度。然而，上下文敏感性会带来高昂的效率成本，因为分析必须计算并维护大量上下文敏感中间结果。因此，上下文敏感性仍难以扩展（无法在数小时内完成分析）；例如，应用最广泛的 2obj（即上下文长度为二的对象敏感性）在标准 DaCapo 基准测试中的若干复杂程序上仍无法扩展 [Jeong 等人，2017；Lhoták 和 Hendren，2006；Li 等人，2018b]。

为适应大型且复杂的程序，近年来提出了大量选择性上下文敏感方法 [Hassanshahi 等人 2017年；Heo 等人 2017年；Jeon 等，2020；Jeong 等人 2017年；Li 等，2018a、b、2020a；Lu 等，2021；Oh 等人 2014年、2015；Smaragdakis 等人 2014年；Tan 等人 2021年；Wei 和 Ryder 2015]。通常，这些方法依靠预分析来选择一组方法：这些方法对于提高精度至关重要，但在上下文敏感的情况下进行分析时不会危及可扩展性。随后，它们仅对选定的方法应用上下文敏感性分析，而对其余方法进行上下文不敏感分析。尽管选择性上下文敏感性显著提升了指针分析的可扩展性，但其效率尚未得到充分释放，因为上下文敏感性的核心方法并未改变：我们仍然需要消耗可能大量的计算资源，复制一组方法，并在不同上下文下分别分析每个方法中的程序元素；而一旦其中若干方法威胁到可扩展性，分析就可能运行得非常缓慢，甚至无法扩展 [Li 等，2020a；Smaragdakis 等人 2014]。

在本文中，我们提出了一种新颖的 Cut-Shortcut 方法，从根本上改变当前的现状：Java 的指针分析主要依赖上下文敏感性来获得更高的精度。

方法。 关键观察是，当对象流在方法 m 中合并，随后合并后的流流出该方法 m时，上下文不敏感性会导致不精确性。不同于像上下文敏感性那样，将一个方法 m 复制成多个副本，以区分来自不同调用点的对象流，我们的方法通过截断会导致某些指针精度损失的流，并直接将快捷流从精确的源指针连接到目标指针，从而绕过方法 m，来模拟上下文敏感性的效果。因此，我们无需应用上下文 m，也能实现上下文敏感性的效果。更具体地说，Cut-Shortcut 不需要预分析，而只需在即时构建的图 PFG′(N, E') 上执行上下文不敏感指针分析算法。该图可以看作是从原始 PFG(N, E) 转换而来的图，其中 N和 E分别表示 PFG 的节点集和边集，而 $E' = E \setminus \{$要截断的边$\} \cup \{$要添加的快捷边 $\}$（换言之，对于标准的上下文不敏感分析，只需将其 PFG 更改为 PFG′）。接下来，我们需要回答两个关键问题：

- 要截断哪些边？我们截断那些可能将方法内部的合并对象流带到外部某处的边（例如，带到调用点左侧、用于接收被调用方法返回变量值的变量处）。换言之，只要将混沌限制在方法内部，对象流本身的合并通常不会损害精度。主要的精度损失发生在合并对象流被传播到方法外部时。
- 在哪里添加快捷边？我们定位被调用方内部对象流合并之前的所有源节点（例如 s），然后在 s 与被剪切的边所对应的目标节点之间创建新边，以推导出可靠且精确的分析结果。

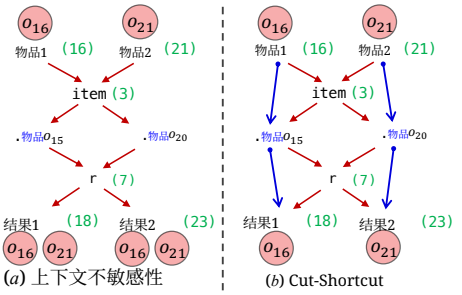
尽管基本思路看起来很简单，但要识别应截断哪些边以及在哪里添加快捷边都颇具挑战。对于前者，我们需要识别那些严重损害精度的边，并评估截断这些边是否有益，这需要进行大量调整。对于后者，在截断了一些 PFG 边之后，如果添加的快捷边无法捕获所有必要的对象流源，可靠性就会受到损害。为确保可靠性，我们没有试图处理各种类型的精度损失，而是在本文中提出了三种具有明确特征的程序模式，这些模式适合采用我们的方法进行处理，并且造成了很大一部分精度损失。

评估。 我们从理论上证明了 Cut-Shortcut 的可靠性，并进一步通过召回率实验间接验证了这一点；此外，为了全面验证其有效性，我们为两个最先进的 Java 指针分析框架实现了 Cut-Shortcut 的两个版本：一个使用 Datalog（≈100 Datalog 规则），用于声明式的 Doop [Bravenboer 和 Smaragdakis 2009]；另一个使用 Java（≈2000 行代码），用于命令式的 Tai-e [Tan 和 Li 2022]。我们选取了近期文献中所有满足实验要求的大型复杂 Java 程序，并将 Cut-Shortcut 与上下文不敏感分析、传统的上下文敏感性方法以及选择性上下文敏感方法进行比较。 **结果极具前景：** *Cut-Shortcut* **在两个框架中，对于大多数评估程序，其运行速度甚至快于上下文不敏感分析，同时还能达到很高的精度，与（若具备可扩展性）上下文敏感分析相当。**这是我们首次能够针对这些难以分析的 Java 程序，在效率与精度之间实现如此出色的权衡；使用随附的制品即可获得实验结果 [Ma 等人 2023]。

2 动机示例

我们使用图1 中的示例来阐释Cut-Shortcut 的基本思想，并展示它与指针分析中的上下文不敏感性和上下文敏感性有何不同。类`Carton` 具有一个字段`item`，并提供`setItem()` 和`getItem()`，分别用于修改和获取`item` 值。在`main()`中，两个`Item` 对象（o_{16}、o_{21}）通过字段`item` 存储在两个`Carton` 对象（o_{15}、o_{20}）中

```
1  类纸箱 {
2    物品 物品;
3    void setItem(物品 物品){
4      this.物品  =物品;
5    }
6    物品 getItem() {
7      物品 r = this.物品;
8      返回 r;
9    }
10 }
11 类 物品 {}
12
13 //用法代码
14 无效 main() {
15   纸箱 c1 = 新建纸箱();//o15
16   物品 item1 =新建物品();//o16
17   c1.setItem(物品1);
18   物品结果1 = c1。getItem();
19
20   纸箱 c2 = new Carton();//o20
21   物品 物品2 =new 物品();//o21
22   c2.setItem(物品2);
23   物品结果2 = c2.获取物品();
24 }
```



(a) 上下文不敏感性　(b) Cut-Shortcut

图1。用于阐释 Cut-Shortcut 基本思想的示例。

通过方法 setItem() 存储，之后通过 getItem() 获取（我们使用 o_i 表示在第 i 行分配的抽象堆对象）。执行后，result1（result2）将只指向 o_{16}（o_{21}）。下面我们将以此示例说明上下文不敏感性、上下文敏感性以及我们的方法是如何工作的。

上下文不敏感性。图1(*a*)展示了左侧代码对应的简化 PFG（PFG 的一条边 e 表示，e 的源指针所指向的内容应流向 e 的目标指针）。由于它无法区分 setItem() 的两次调用，pt(item1)（我们使用 $pt(x)$ 表示 x 的指向集合）和 pt(item2) 将合并到 pt(item) 中（其中 item 是第 3 行的参数），最终导致第 4 行的存储语句为 o_{15}.物品 和 o_{20}.物品 生成不精确的指向集合。此外，两次 getItem() 调用也以相同方式处理，因此合并后的对象（$\{o_{16}, o_{21}\}$）会在第 7 行的加载语句中流向 r，使 result1 和 result2 的指向集合变得不精确。

上下文敏感性。参见图1(*a*)，上下文敏感性的目标是分离在 item（第3行）和 r（第7行）处合并的流，并让 o_{16} 仅流向 result1，让 o_{21} 流向 result2。为了区分来自不同调用点的对象流（即 setItem() 的第17行和第22行，以及 getItem() 的第18行和第23行），setItem() 中的每个程序元素（即 item 和 this.item）以及 getItem() 中的每个程序元素（即 this.item 和 r）都应使用上下文进行限定并分别分析；这等价于在两个不同的上下文下分别分析这些方法两次。可以预见，在现实世界中，计算和维护上下文敏感信息的成本会非常高，尤其是在程序规模较大或较为复杂时。选择性上下文敏感性可以缓解这一问题，但并未从根本上改变它，因为在复杂程序中，即使只选择少量方法，也可能威胁可扩展性，并在对这些方法进行上下文敏感分析时导致分析规模急剧膨胀 [Li 等，2020a；Smaragdakis 等人 2014]。

Cut-Shortcut。图1(*b*)展示了 Cut-Shortcut 的工作原理，其中带叉号的箭头表示要截断的边，蓝色箭头表示要添加的快捷边。o_{15}.item 和 o_{20}.item 的指向集合不精确性源于边 item $\rightarrow o_{15}$.item 和 item $\rightarrow o_{20}$.item（由第4行的存储语句引入），因此我们将它们截断，并从精确的源（即 item1 或 item2）直接向目标节点添加快捷边。同样，对于节点 result1 和 result2，我们截断从 r 开始的、会引入不精确性的边，并添加快捷边 o_{15}.item $\rightarrow$ result1 和 o_{20}.item $\rightarrow$ result2。然后，在修改后的 PFG 上，应用与上下文不敏感性相同的算法（沿 PFG 边传播指向结果），最终将得到对于此示例而言与上下文敏感性同样精确的指向结果。

3 Cut-Shortcut 方法简介

我们介绍 Cut-Shortcut 背后的总体原则（第 3.1 节），三种程序模式都基于该原则，包括字段访问模式（第 3.2 节）、容器访问模式（第 3.3 节）和局部流模式（第 3.4 节）。

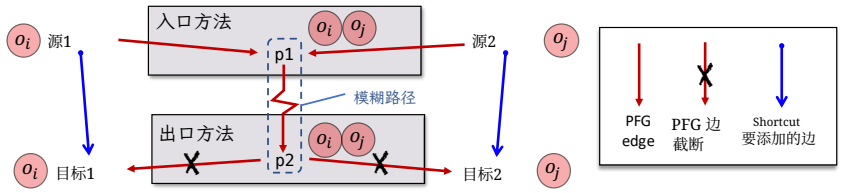


图2。Cut-Shortcut 关于在 PFG 中截断哪些边以及在哪里添加快捷边的见解。

3.1 概述

简而言之，Cut-Shortcut 遵循一项通用原则，通过在 PFG 上切断承载合并流的边，并添加快捷边作为替代，解决上下文不敏感指针分析中的精度损失问题。为了更好地阐释这一原则，下面我们首先明确一些术语。

定义 *3.1*（局部/非局部指针）。如果指针 p 是方法 m 的局部变量，则 p 对 m 来说是局部的；否则 p 对 m 来说是非局部的，即 p 是在 m 外部声明的变量，或是一个实例字段。

定义 *3.2*（模糊路径、入口和出口）。如果 PFG 的路径 p 满足以下条件：（1）它以一个汇合点开始，在该汇合点处，两个（或更多）对于 方法 m 来说是非局部的指针汇合为一个对于 m 来说是局部的指针 s（此时 s 是 p 的起始节点）；（2）它以一个分叉点结束，在该分叉点处，一个对于 方法 n 来说是局部的指针 e（此时 e 是 p 的结束节点）指向两个（或更多）对于 n 来说是非局部的指针，则 p 是一条模糊路径，其中 m 是其入口，n 是其出口。

以模糊路径 物品 $\rightarrow o_{15}$.物品 $\rightarrow$ r 在图 1中*a*）为例。 物品 和 r 分别是该路径的起始节点和结束节点，其 入口 是 `setItem()`，而 出口 是 `getItem()`。路径 物品 （在第3行）同样是模糊的，其起始节点和结束节点相同，即 物品。

定义*3.3*（目标 和 源 指针）。给定一条模糊路径 p，目标指针（简称目标）是其 p结束节点的任意后继节点。在PFG中，目标接收来自出口的合并指向结果。源指针（简称源）是 p起始节点的任意前驱节点，其指向结果流向入口，并促成PFG上的合并流。

现在，我们通过图2解释该原理的核心思想。p1、p2、目标1、目标2、源1和源2是PFG节点，表示程序指针（即变量和实例字段），而 o_i和o_j表示两个抽象堆对象。假设p1和p2是模糊路径的起始节点和结束节点，那么源1和源2会被识别为源，目标1和目标2会被识别为目标。在动态执行期间，目标1仅指向来自源1的 o_i；然而，由于模糊路径造成的上下文不敏感性，其指向集合同时包含 o_i和o_j。在许多实际程序中，借助对传统指针分析所忽略的语义信息的理解，人们可以凭直觉轻易识别出精确的指向结果。我们的方法实际上也是如此：利用某些语义线索更好地捕获程序的动态行为，从而通过截断离开模糊路径的边（例如，p2→目标1/目标2），并添加快捷边连接精确的源和目标（如源1→目标1），有效提高精度。需要注意的是，“截断边”这一操作是相对于原始PFG而言的；在Cut-Shortcut中，我们不会截断已经添加到PFG中的边，因为边一旦被添加后又被移除，可能会沿着这些边引入虚假的对象流。因此，在构建PFG时，我们从不添加应被截断的边。

为了获得更高的精度，我们尝试尽可能找出更多的目标，并截断那些通向目标且会造成精度损失的边；对于每个目标，我们尽可能准确地识别其对应的源，并在源与目标之间创建快捷边。然而，这很难，因为

真实程序通常会表现出错综复杂的行为，几乎不可能提供一种通用方案来处理所有精度损失情况 [Li 等，2018a、2020a]；更重要的是，在操作 PFG 时，我们必须确保分析的可靠性。因此，我们针对三种程序模式指定了 Cut-Shortcut：字段访问模式、容器访问模式和局部流模式。这些模式在程序中被广泛使用，并导致指针分析产生了很大一部分精度损失。在以下小节中，我们将分别讨论如何识别每种模式的 入口 和 出口、源 和 目标，以及如何精确且可靠地截断边并添加快捷边。

Cut-Shortcut 与选择性上下文敏感性有着根本区别 [Jeon 等人 2019年，2020年；Li 等，2018a，b，2020a；Smaragdakis 等人 2014年；Tan 等人 2021]。选择性上下文敏感方法通过不同的启发式策略选择方法，并直接为这些方法应用上下文（等同于克隆这些方法并对其进行分析）；这类分析的主要成本来自克隆方法并多次分析它们。与此不同，为避免克隆带来的开销，Cut-Shortcut 既不选择方法，也不以上下文敏感的方式分析方法；相反，它通过删除和添加 PFG 的边来模拟上下文敏感性的效果（挑战在于删除边时确保可靠性），并在修改后的 PFG 上传播指向信息。在 Cut-Shortcut 中不会对任何方法应用上下文。

3.2 字段访问模式

Java 中有两种字段访问：字段存储和字段加载（在本文中，我们关注实例字段）。Java 程序通常会将字段访问封装在方法（例如 setter/getter）中，因此客户端代码需要通过调用这些方法来访问字段。然而，在上下文不敏感性下，对不同对象的字段进行访问时所涉及的对象流会在这些方法内部合并，从而导致不精确性。在这一模式中，我们根据 Cut-Shortcut 的原则处理这种精度损失，具体如下所述。为便于理解，我们首先使用图 1 中的示例介绍字段存储（第 3.2.1 节）和字段加载（第 3.2.2 节）的两个基本情况。该示例依赖于字段访问和参数的信息。随后，我们在图 3 中描述一个更一般的情况，以展示我们的基于语义的方案如何通过参数跟踪从调用者（以及调用者的调用者.....）流出的值，直到这些值最终存储在目标字段中。

3.2.1 存储处理。我们解决实例字段指向结果中的不精确性。对于字段存储语句 s：$x.f = y$，如果 x 和 y 指向的对象都来自包含 s的 method m 的参数（包括 this 变量），那么来自不同调用（ m的调用）的实参的传入对象流会在 m内部合并，这可能导致 $pt(o_i.f)$ 中出现不精确性，其中 $o_i \in pt(x)$。图1第4行给出了这样一个例子：this 和 item 是方法 setItem() 的参数。以上下文不敏感的方式分析 setItem() 时，来自第17行和第22行调用点的对象会在其中合并，导致结果不精确，即 $pt(o_{15}.\text{item}) = pt(o_{20}.\text{item}) = \{o_{16}, o_{21}\}$。

在 Cut-Shortcut 中，我们的想法是截断将合并对象流传播到目标的存储边，并在包含字段存储的 method 的调用点处查找源。我们搜索这样的存储语句 s：$x.f = y$： x和 y都是 m（包含 s）的参数，且在 m中没有被重新定义（这保证了 x和 y的值都来自 m的调用点的实参）。如果找到这样的 s，那么 m的不同调用就构成一条模糊路径（仅有一个节点 y，例如图 1(a)中的物品(3)），使 m 同时成为入口和出口。因此，目标是通过 $x.f$访问并存储在 s的实例字段（例如，第4行存储的 o_{15}.item 和 o_{20}.item）。为防止合并对象流传播到目标，我们截断指向它们的存储边（即图1中的 item $\rightarrow o_{15}$.item 和 item $\rightarrow o_{20}$.item）。

由于 x和 y的值都来自 m的调用点中的实参，因此我们可以在每个调用点找到匹配的目标和源。也就是说，源是传递给 y 的实参（例如，该实参在第17行是 item1，而 y在第3行是参数 item），目标是 $o_i.f$，

其中 $o_i \in pt(a)$， a 是传递给 x 的实参（例如，实参 a 在第17行是 c1，而x 是参数 this）。例如，在第17行，item1 是目标 o_{15}.item 的源（因为pt(c1) $=\{o_{15}\}$），同样地，item2 是第22行目标 o_{20}.item 的源（因为 pt(c2) $=\{o_{20}\}$）。因此，我们添加快捷边 item1 $\rightarrow o_{15}$.item 和 item2 $\rightarrow o_{20}$.item，并得到精确结果，即 $pt(o_{15}.\text{item}) = \{o_{16}\}$ 和 $pt(o_{20}.\text{item}) = \{o_{21}\}$（如图1($b$)上半部分所示）。

3.2.2 加载处理。对于值从加载实例字段的方法返回的左值变量，我们要解决其指向结果中的不精确性。对于字段加载语句s： $x = y.f$，如果 y 指向的对象来自包含 s的方法m的参数（包括 this 变量），并且 x是 m的返回变量，那么从 $y.f$加载的对象（其中 y指向来自对 m的不同调用的实参的传入对象）会在 m中合并，这可能导致 m的调用点处左值变量的指向集合不精确。图1中的第 7 行给出了这样一个例子：this 是方法 getItem() 的参数，r 是其返回变量。当以上下文不敏感的方式分析 getItem() 时，第 18 行和第 23 行调用点的对象（即 o_{15} 和 o_{20}）会在其中合并，从 o_{15}和 o_{20}（即 o_{16}和 o_{21}）加载的对象也会在 r 处合并，并传播到 result1 和 result2，导致结果 pt（result1）、 $= pt$（result2） $=\{o_{16}, o_{21}\}$不精确。

在 Cut-Shortcut 中，我们的思路是截断从 m 出发的返回边（这些边会返回从字段加载的值），因为它们会将合并对象流传播到目标；然后在 m的调用点查找源。我们搜索加载语句 s： $x = y.f$其 x和 y 的值都与 m的调用点中的变量直接相关（其中包含 s），即 y是 m的参数，且在 m中未被重新定义（这保证了 y的值都来自 m的调用点的实参），并且 x是 m的返回变量。如果找到这样的 s和 m，我们就截断从 m到 m的每个调用点的左值变量（即目标）的返回边，以避免传播合并流；然后在调用点识别源，即 $o_i.f$，其中 $o_i \in pt$（a），而 a是传递给 y的实参（例如， a是第18行的 c1， y是 this）。例如，对于图1中第7行的字段加载，我们截断边 r $\rightarrow$ result1 和 r $\rightarrow$ result2。对于 result1，我们在第18行找到它的源 o_{15}.item，（如 pt(c1) $=\{o_{15}\}$），并添加快捷边o_{15}.物品 $\rightarrow$ result1。同样地，我们在第 23 行添加 o_{20}.物品 $\rightarrow$ result2。最后，我们获得精确结果， pt(result1) $=\{o_{16}\}$ 和 pt(result2) $=\{o_{21}\}$（见图1(b)下半部分）。

3.2.3 字段访问的嵌套调用处理。在实际程序中，当涉及嵌套方法调用时，字段访问会变得复杂。例如，调用构造函数时，它可能会调用另一个构造函数或 setter 方法，最终执行存储语句。此外，当使用继承时，使用嵌套调用进行字段访问也很常见。

```
1 类 A{                          6 // 用法代码
2   T f;                         7 T t1 = new T(); //o7
3   A(T t){this.set(t);}         8 一个 一个1 = 新的 A(t1); //o8
4   设置(T p){this.f=p;}         9 T t2 = new T(); //o9
5 }                             10 A a2 = new A(t2); //o10
```

图3。字段访问的嵌套调用示例。

例如，在图3中，A.set() 是由A的构造函数（第 3 行）调用的设置器。对它存在两个嵌套调用：8 $\rightarrow$ 3 $\rightarrow$ A.set() 和 10 $\rightarrow$3 $\rightarrow$ A.set()（调用点由其行号表示）。在此，如果我们添加快捷边，t $\rightarrow o_8$.f 和t $\rightarrow o_{10}$.f，在A.set() 的直接调用点（第 3 行）处，我们仍然会损失精度，因为 this 和 t 都是 A() 的参数，来自第 8 行和第 10 行调用点的对象流在此处合并，导致不精确的结果 $pt(o_8.f) = pt(o_{10}.f) = \{o_7, o_9\}$。类似地，字段加载中的嵌套调用也可能导致精度损失。为处理字段访问中的嵌套调用，我们将方法扩展为一种更通用的方式，以获得更高的精度。具体而言，对于包含字段访问的方法，我们分析其嵌套调用（从该方法本身到其调用者、调用者的调用者，以此类推，在可能的情况下），以在适当的调用点截断边并添加快捷边。对于 A.set() 中的字段存储，我们追踪到第 8 行和第 10 行的调用点，添加快捷边 t1 $\rightarrow o_8$.f 和 t2 $\rightarrow o_{10}$.f，并推导出精确结果 $pt(o_8.f) = \{o_7\}$ 和 $pt(o_{10}.f) = \{o_9\}$。有关我们处理字段存储和加载中嵌套调用的方法详情，请参阅第 4.2 节。

3.3 容器访问模式

容器（例如 ArrayList）在 Java 程序中被广泛使用。由于大量对象流入和流出容器，它们对指针分析的精度至关重要。在上下文不敏感性下，存储在不同容器中的对象可能流入同一个容器方法并在其中合并；当合并后的对象流出时，可能会造成严重的精度损失。为了分离不同容器中的对象流，传统方法通过以上下文敏感的方式分析容器方法来提高精度。然而，正如近期工作所指出的 [Antoniadis 等人，2020；Fegade 和 Wimmer 2020]，，对容器方法应用上下文敏感性会带来巨大的效率开销，尤其是对于大型程序。因此，为了在精确分析容器的同时降低分析开销，研究人员 [Antoniadis 等人，2020；Fegade 和 Wimmer 2020]提出手动重写容器实现，以组合出比原始实现更简单的代码模型，同时仍保留容器对指针分析的副作用。然后，他们在分析过程中对重写后的容器代码应用上下文敏感性。在 Cut-Shortcut中，我们同样需要处理容器以获得良好的精度。但我们将采用不同的方式，遵循 Cut-Shortcut的原则。

3.3.1 处理容器。 图4 展示了一个 ArrayList 容器示例（目前只需关注第 1–9 行）。我们创建了两个 ArrayList，o_1（第 1 行）和 o_6（第 6 行），并分别向其中添加两个对象 o_2（第 3 行）和 o_6（第 8 行）。在第 4 行和第 9 行，我们从 o_1 和o_6 中取出元素，并分别将它们赋给 x 和 y。显然，在运行时，x （y）只指向 o_2 （o_7）。

容器实现通常会提供用于向容器添加元素或从容器中取回元素的应用程序编程接口。因此，自然可以将添加和取回接口分别视为容器的 入口 和出口 方法，因为对象通过这些接口存入容器并从容器中流出。我们将 入口 和 出口 的调用点分别记为 入口$_{cs}$ 和 出口$_{cs}$；例如，如图4中的注释所示，add() （get()）是 入口 （出口）ArrayList的。

```
1  ArrayList l1 = new ArrayList();//o1,pt_H(l1)={o1}
2  对象a = new Object(); //o2
3  第1行。添加(a); //Entrance，a 是Source
4  对象x = l1.get(0); //Exit，x 是 Target
5
6  ArrayList l2 = new ArrayList(); //o6,pt_H(l2)={o6}
7  对象b = new Object(); //o7
8  l2.add(b); //Entrance，b 是Source
9  对象y = l2.get(0); //Exit，y 是 Target
10
11 迭代器it1 = l1.iterator(); //pt_H(it1)={o1}}
12 对象r1 = it1.next(); //Exit，r1 是 Target
13 迭代器it2 = l2.iterator(); //pt_H(it2)={o6}
14 对象r2 = it2.next(); //Exit，r2 是Target
```

图 4。ArrayList 的示例。

随后，相关的实参被视为一个入口$_{cs}^{arg}$（例如入口$_{cs}^{arg}$）；例如，第 3 行的 a 是第 3 行 add() 调用点的入口$_{cs}^{arg}$。出口$_{cs}$ 的左值变量被视为目标指针（例如出口$_{cs}^{lhs}$）；例如，第 4 行的 x 是第 4 行 get() 调用点的出口$_{cs}^{lhs}$。在 Cut-Shortcut 中，我们截断从出口到出口$_{cs}^{lhs}$的返回边，使容器内部合并的对象流不会传播到出口$_{cs}^{lhs}$。随后，我们添加一条快捷边，从

每个入口$_{cs}^{arg}$指向对应的出口$_{cs}^{lhs}$，以补充出口$_{cs}^{lhs}$的指向集合。在上下文不敏感性分析中，o_2 和 o_7 会在 ArrayList 的方法中合并，并通过 get() 分别流向 x 和 y，从而导致不精确性。在 Cut-Shortcut 中，为避免精度损失，我们截断从 get() 返回到出口$_{cs}^{lhs}$、x 和 y 的边。要添加快捷边，我们需要找出匹配的入口$_{cs}^{arg}$和出口$_{cs}^{lhs}$对（如果它们与同一个容器相关，则二者匹配）。在本例中，a 和 x 是一对匹配的入口$_{cs}^{arg}$和出口$_{cs}^{lhs}$，因为它们对应的入口$_{cs}$（第 3 行的 add()）和出口$_{cs}$（第 4 行的 get()）与同一个容器（o_1）相关。

一种直接判断入口$_{cs}$ 和出口$_{cs}$ 是否相关的方法，是检查对应入口$_{cs}^{rv}$和出口$_{cs}^{rv}$的指向集合是否存在重叠，其中入口$_{cs}^{rv}$ 和出口$_{cs}^{rv}$ 是入口$_{cs}$ 的接收者变量（例如第 3 行 add() 的 l1）

以及出口$_{cs}$（例如第 9 行 get() 的 l2）。然而，这种方法无法处理迭代器和 Map 的 keySet() 等其他容器情况。因此，我们引入一个名为指针-宿主映射（pt_H）的概念，以统一解决上述问题。

pt_H将宿主相关指针映射到与之相关的一组容器对象（称为宿主）。例如，ArrayList o_1 被赋值给 l1（第 1 行），因此我们有 pt_H(l1) = {o_1}（稍后我们会介绍更多映射规则）。接着，如果 pt_H(入口$^{rv}_{cs}$) 与 pt_H(出口$^{rv}_{cs}$) 重叠（即它们可能共享同一个宿主容器对象），我们就从相应的入口$^{arg}_{cs}$到出口$^{lhs}_{cs}$添加一条快捷边。在上面的例子中，由于 pt_H(l1)（其中 l1 是第 3 行 add() 的接收者变量）与pt_H(l1)（其中 l1 是第 4 行 get() 的接收者变量）重叠，我们添加一条快捷边 a → x；同理，对于 b → y 也如此。因此，x（y）只能精确地指向 o_2（o_7）。在这种情况下，pt_H发挥着与典型指向关系 pt相同的作用。我们将在下文解释 pt_H的其他用途。

3.3.2 处理依赖宿主对象。 图 4中的第 11-14 行展示了容器的一种非常常见的用法，即通过迭代器访问元素。为了以更高的精度全面处理容器，我们需要考虑这种用法。通常，每个迭代器都依赖于一个容器（即宿主），而从迭代器中检索的元素就是存储在宿主中的元素。我们将迭代器对象称为依赖宿主对象[1]。

为了处理依赖宿主对象，我们的原则与第3.3.1节中的原则保持一致：截断从出口到出口$^{lhs}_{cs}$的返回边，并在 pt_H(入口$^{rv}_{cs}$)与 pt_H(出口$^{rv}_{cs}$)重叠时，从每个入口$^{arg}_{cs}$向对应的出口$^{lhs}_{cs}$添加一条快捷边。区别在于，对于依赖宿主对象，我们需要扩展入口、出口和 pt_H 的集合。在我们的示例中，容器元素可以通过 next()（第 12 行和第 14 行）检索，因此将方法 next() 添加为出口（在此情况下无需扩展入口）。至于 pt_H，在我们的示例中，我们需要通过以下方式扩展这一映射关系：(1) 将一个宿主相关指针，例如 $Hptr$（如第 11 行的迭代器指针 it1）添加到其键集合中；(2) 将其相关宿主对象，例如 H（如 o_1ArrayList 对象）添加到其值集合中。接下来的问题是如何通过 $Hptr$识别 H？为回答这一问题，我们首先定义一种称为 Transfer 方法的记法。如果一个依赖宿主对象（即 $Hptr$所指向的对象）是通过调用宿主 H所属类的方法 m创建的，我们就称 m为 Transfer。在我们的示例中，第 11 行的 iterator() 是一个 Transfer，因为它是 H所属类（ArrayList）的方法，并且调用它会创建/返回一个由 $Hptr$（第 11 行的 it1）指向的迭代器对象。因此，为了将$Hptr$ 映射到 H，对于 Transfer 的每个调用点，我们将其接收者变量的宿主对象（第 11 行的pt_H(l1)）传播到其左值变量（第 11 行的 it1）。因此，我们得到新的 pt_H关系 pt_H(it1)= {o_1}，因为 pt_H(it1) ⊇ pt_H(l1) = {o_1}。我们将 iterator() 称为 Transfer，因为它将宿主对象从其接收者变量传递到其左值变量。请注意，构建 pt_H 需要指针分析的结果，而我们的容器访问模式检测需要与指针分析一起在线完成。借助指针分析，我们的方法还可以处理构建 pt_H 时涉及别名的情况，其详细信息将在第4.3节的规则中形式化。

在我们的方法中，只需指定哪些应用程序编程接口是入口，出口或Transfer，其余工作则由我们的分析自动完成。为保证通用性，我们只考虑 JDK 容器，因为它们的应用程序编程接口稳定。由于这些接口文档完善，而且可以根据名称轻松识别相关应用程序编程接口的类型，因此作者之一仅用五个小时就完成了 JDK 中所需应用程序编程接口的指定。与以往需要重写相关容器应用程序编程接口实现的工作相比 [Antoniadis 等人，2020；Fegade 和 Wimmer 2020] ，我们的方法更通用，也更简单。

[1]实际上，依赖宿主对象不仅包括迭代器，还包括一些依赖特定容器的其他对象，例如 Map 的集合视图（如 Map.keySet() 的返回值），这些对象同样可以通过我们的方法进行处理。

3.4 局部流模式

这是一种简单的模式，用于处理这样一种精度损失情况：方法参数的值通过一系列局部赋值流向返回变量。当这样的一个方法在多个调用点被调用时，各调用点的实参值会被合并到该方法中，并一起传播到每个调用点处的左值变量，导致指向结果不精确。我们用图5 中的示例来说明这种模式。在方法`select()`中，其参数`p1` 和`p2` 的值会流向返回变量`r`。它在第 12 行和第 16 行有两个调用点。在上下文不敏感的分析中，对象o_{10}、 o_{11}、 o_{14} 和 o_{15} 会被传入`select()`并在其中合并，随后不精确地一起流向`r1`和`r2`，因为在任何一次执行中，`r1` (`r2`)只会指向 o_{10} 或 o_{11} (o_{14} 或 o_{15})。

```
1 A 选择(A p1,Ap2){        9 // 用法代码
2   A r;                  10 Aa1 = 新建A(); //o10
3   如果(...)             11 A a2 = new A(); //o11
4       r= p1;            12 A r1 = 选择(a1, a2);
5   else                  13
6       r= p2;            14 A a3 = new A(); //o14
7   返回 r;               15 Aa4 = 新的A(); //o15
8 }                       16 A r2 = 选择(a3, a4);
```

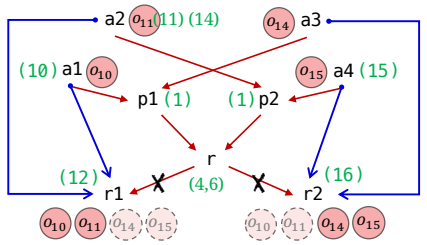


图 5. 局部流模式示例。

在 Cut-Shortcut 中，对于这种模式，入口和出口是同一个方法，其参数值会流向返回变量。我们关注这样一种情况：方法的返回值必须全部来自其参数，并通过局部赋值传递（不包括字段加载和方法调用等其他返回值来源）；检测这类情况依赖于过程内值流分析。因此，我们可以安全地将实参（与参数对应）用作源，并从它们向目标（即调用点的左值变量）添加快捷边。在此示例中，我们截断了来自 r 的返回边，并在第 12 行的调用点添加快捷边 a1 → r1 和 a2 → r1，在第 16 行的调用点添加快捷边 a3 → r2 和 a4 → r2。这样，在 Cut-Shortcut 中，r1（r2）仅指向 {o_{10}，o_{11}}（{o_{14}，o_{15}}），而是精确的。我们在第 4.4 节正式描述如何分析这种模式。

局限性。请注意，我们的方法仅适用于 *Java*，因此不能直接应用于其他编程语言；不过，*Cut-Shortcut*（图2）的高层思路，即移除精度损失边并向 PFG 添加语义保持边，或许能启发用于提高其他语言指针分析精度的技术。关于将 Cut-Shortcut 应用于上下文敏感性等其他抽象，我们目前的方法并非设计为可插入其他（选择性）上下文敏感分析中。我们不是选择方法并对其进行上下文敏感分析，而是在不应用任何上下文的情况下直接操作 PFG 边。不过，尝试将两者结合起来会很有意思。例如，对于 PFG 边未受我们方法影响、但会被其他选择性上下文敏感方法考虑的方法 [Jeon 等人 2019；Li 等，2018a，b，2020a；Smaragdakis 等人 2014],我们可以对其进行上下文敏感分析，以期获得更高的精度。至于流敏感性，与以往文献中的几乎所有 Java 全程序指针分析一样 [Jeon 和 Oh，2022；Lu 等，2021；Smaragdakis 和 Balatsouras，2015；Sridharan 等人，2013；Tan 等人 2021], 我们的方法也是流不敏感的。将我们当前的方法应用于流敏感分析并非易事，但未来尝试这一方向会很有意思。

4 形式化表示与可靠性

本节将使用 Cut-Shortcut 方法（第 4.1 节）形式化指针分析，并介绍三种模式（第 4.2– 4.4 节），同时证明 Cut-Shortcut 的可靠性（第 4.5 节）。

方法 $m \in \mathbb{M}$　　$i \xrightarrow{call} m$ 从调用点 i 到方法 m 的调用边 指令标签 i，
$j \in \mathbb{L}$　　i_{ak} 调用点的第 k 个实参 i变量 x，$y \in \mathbb{V}$　　m_{pk} 方法
m的第 k 个形参 堆对象 o_i，$o_j \in \mathbb{O}$　　m_{ret} 方法 m的返回变量 字段
$f \in \mathbb{F}$　　def_x 定义变量 x的语句集合 指针 x，$o_i.f \in \mathbb{P} = \mathbb{V} \cup (\mathbb{O} \times \mathbb{F})$ PFG
$G = (N, E)$ 指向关系　$pt: \mathbb{P} \to \mathrm{P}(\mathbb{O})$ 节点 $n \in N = \mathbb{P}$ 边 $s \to t \in E \subseteq N \times N$

图 6. 形式化表示中使用的域和符号。

$$\frac{i: x = newT()}{o_i \in pt(x)}\ [\textsc{New}] \qquad \frac{x = y}{y \to x \in E}\ [\textsc{Assign}] \qquad \frac{s \to t \in E}{pt(s) \subseteq pt(t)}\ [\textsc{Propagate}] \qquad \frac{e \in E_{SC}}{e \in E}\ [\textsc{Shortcut}]$$

$$\frac{x = y.f \quad o_i \in pt(y)}{o_i.f \to x \in E}\ [\textsc{Load}] \qquad \frac{i: x.f = y \quad o_j \in pt(x) \quad \boxed{i \notin cutStores}}{y \to o_j.f \in E}\ [\textsc{Store}]$$

$$\frac{i: x = y.m(arg_1, ..., arg_n) \quad o_j \in pt(y) \quad m' = dispatch(o_j, m)}{i \xrightarrow{call} m' \quad o_j \in pt(m'_{p0})}\ [\textsc{Call}]$$

$$\frac{i: x = y.m(arg_1, ..., arg_n) \quad i \xrightarrow{call} m'}{\forall\, 1 \le k \le n : arg_k \to m'_{pk} \in E}\ [\textsc{Param}] \qquad \frac{i: x = y.m(...) \quad i \xrightarrow{call} m' \quad \boxed{m'_{ret} \notin cutReturns}}{m'_{ret} \to x \in E}\ [\textsc{Return}]$$

图 7. 使用 Cut-Shortcut 的指针分析规则。

4.1 使用 Cut-Shortcut 的指针分析

本节将对集成了我们的 Cut-Shortcut 方法的指针分析进行形式化。为便于说明，我们在图 6 中给出了形式化表示中将使用的域和符号，其中大多不言自明。这里，$pt(x)$ 给出分析结果：它将每个指针 x 映射到其指向集合（$\mathcal{P}(\mathbb{O})$表示 $\mathbb{O}$的幂集）；对于 m_{pk}和 i_{ak}，我们用 $m_{p0}(k{=}0)$ 表示方法 中的 this 变量 m，并用 i_{a0} 表示调用点 i的接收者变量。

以 PFG 视角看待指针分析。图 7 展示了我们的指针分析形式化表示（目前暂时忽略规则 [Shortcut] 以及黑框中的前提，这些内容与 Cut-Shortcut 相关）。本质上，我们的形式化表示与现有文献中出现的形式化表示等价 [Milanova 等人，2005；Smaragdakis 和 Balatsouras，2015；Sridharan 等人，2013]，因为它们都表达了针对 Java 的 Andersen 风格指针分析。然而，为了与 Cut-Shortcut 集成，我们通过 PFG（指针流图）的边显式表达指针之间的子集约束。具体而言，PFG 的所有节点都是指针，并且根据 [Propagate]，PFG 中从 $s \to t$ 指向 s 的边表示由 s 指向的对象也应当由 t 指向（即子集约束）。其他规则描述了如何为不同类型的语句生成 PFG 边：对象分配（[New]）、局部赋值（[Assign]）、实例字段加载（[Load]）和存储（[Store]），以及方法调用（[Call],[Param], [Return]）。

Cut-Shortcut。Cut-Shortcut 是一种通用方法，始终遵循同一原则：首先截断可能引入精度损失的边，然后添加快捷边，将适当的源连接到目标。为形式化这一原则，我们定义了三个集合：*cutStores* 和 *cutReturns* 用于截断边，E_{SC} 用于添加快捷边，具体解释如下。

- *cutStores* 是一组存储语句（由其标签标识）。通过 [Cut]，如果某个存储语句 j位于 *cutStores* 中，那么本应为 j 生成的存储边将被截断。

$$\frac{i \xrightarrow{call} m \quad m_{pk} = x \quad def_x = \emptyset}{i_{ak} \mapsto x} \text{ [Arg2Var]} \qquad \frac{i : x.f = y \quad j_{ak_1} \mapsto x \quad j_{ak_2} \mapsto y}{i \in cutStores \quad \langle j_{ak_1}, f, j_{ak_2} \rangle \in tempStores} \text{ [CutStore]}$$

$$\frac{\langle base, f, from \rangle \in tempStores \quad j_{ak_1} \mapsto base \quad j_{ak_2} \mapsto from}{\langle j_{ak_1}, f, j_{ak_2} \rangle \in tempStores} \text{ [PropStore]}$$

$$\frac{\langle base, f, from \rangle \in tempStores \quad base \text{ in } m \quad o_j \in pt(base) \quad ((def_{base} \neq \emptyset) \vee (def_{from} \neq \emptyset) \vee (\nexists k : m_{pk} = base) \vee (\nexists k : m_{pk} = from))}{from \rightarrow o_j.f \in E_{SC}} \text{ [ShortcutStore]}$$

图 8. 字段访问模式中处理字段存储的规则。

$$\frac{(i : \quad to = base.f) \vee (to, \quad base, \quad f) \in tempLoads) \quad m_{ret} = to \quad j : r = _ \quad j_{ak} \mapsto base \quad o_n \in pt(base)}{m_{ret} \in cutReturns\ r, j_{ak}, f \in tempLoads \quad o_n.f \rightarrow to \in returnLoadEdges} \text{ [CutPropLoad]}$$

$$\frac{to, base, f \in tempLoads \quad o_i \in pt(base)}{o_i.f \rightarrow to \in E_{SC}} \text{ [ShortcutLoad]} \qquad \frac{i : r = _ \quad i \xrightarrow{call} m \quad m_{ret} \in cutReturns \quad n \rightarrow m_{ret} \in (E \setminus returnLoadEdges)}{n \rightarrow r \in E_{SC}} \text{ [RelayEdge]}$$

图 9. 字段访问模式中字段加载的处理规则。

- $cutReturns$是一组返回变量。通过 [Cut],，如果某个返回变量 m_{ret}位于 $cutReturns$中，那么从 m_{ret} 到 m调用点的左值变量的返回边将被截断。
- E_{SC}是一组快捷边。根据 [Shortcut],，如果将边 $s \rightarrow t$添加到 E_{SC}，它也会被添加到 PFG，并将 s 所指向的对象传播到 t。

接下来，我们将在第 4.2–4.4 节中说明如何针对这三种模式计算这三个集合。

4.2 字段访问模式

本节将形式化说明我们对字段存储和加载的处理方式，包括字段访问中的嵌套调用。

4.2.1 存储的处理。 图 8 给出了计算规则 $cutStores$和 E_{SC}以处理字段存储。直观而言，当我们能够确认存储操作的基变量和右值变量（例如 x和 y的 $x.f = y$）在每次调用时始终指向调用点实参所对应的相同对象时，我们会截断存储语句引入的 PFG 边，并在调用点（即包含该存储操作的方法的调用点）添加适当的快捷边。为帮助建立快捷边，我们定义 $tempStores$，它是一组三元组，例如 $\langle base、\ f、\ from \rangle$，表示一个（潜在的）存储操作 $base.f = from$。为了以更高精度处理第 3.2.3 节提到的嵌套调用，我们沿调用链构造并传播临时存储，从最内层的被调用方（包含实际存储语句）传播到最外层的调用者，具体如下所述。

Cut。为简化规则，我们引入由 $\mapsto$定义的记号 [Arg2Var],，其中 $i_{ak} \mapsto x$表示调用点 i的第 k个实参的值流向方法 m的变量 x，且 x在 m中不会被重新定义，即在 m的每次调用中，x都指向与 i_{ak} 相同的对象。现在我们定义 $cutStores$在 [CutStore]中的含义。对于存储语句 i： $x.f = y$，如果 i所在方法的调用点的实参是 x和 y的来源，那么我们将 i添加到 $cutStores$，以截断由 i引入的 PFG 边。此外，[CutStore]会生成临时存储，这些存储可能会进一步传播到调用方。

*Shortcut.*我们定义两条规则 [PropStore] 和 [ShortcutStore]，用于为字段存储推导 E_{SC}。[PropStore] 描述了构造新的临时存储，并将其从被调用方递归传播到调用者的过程。在 [ShortcutStore], 中，如果析取条件成立，则表示该临时存储无法继续传播到调用者，此时我们会为该临时存储生成快捷边。

$$\frac{m, _ \in \text{出口}s}{m_{ret} \in cutReturns}\ \text{[CutContainer]} \qquad \frac{h \overset{c}{\Leftarrow} s \quad h \overset{c}{\Rightarrow} t}{s \rightarrow t \in E_{SC}}\ \text{[ShortcutContainer]}$$

$$\frac{i: \ x = y.m\ (\ldots) \quad i \xrightarrow{call} m' \qquad m', k, c \in \text{入口}s \qquad h \in pt_H(y)}{h \overset{c}{\Leftarrow} i_{ak}}\ \text{[HostSource]}$$

$$\frac{i: \ x = y.m\ (\ldots) \quad i \xrightarrow{call} m' \qquad m', c \in \text{出口}s \qquad h \in pt_H(y)}{h \overset{c}{\Rightarrow} x}\ \text{[HostTarget]}$$

$$\frac{o_i \in pt(x) \quad typeof(o_i) <: \text{集合}}{o_i \in pt_H(x)}\ \text{[ColHost]} \qquad \frac{o_i \in pt(x) \quad typeof(o_i) <: \text{Map}}{o_i \in pt_H(x)}\ \text{[MapHost]}$$

$$\frac{i: \ x = y.m\ (\ldots) \quad i \xrightarrow{call} m' \qquad m' \in \text{Transfers} \qquad h \in pt_H(y)}{h \in pt_H(x)}\ \text{[TransferHost]}$$

$$\frac{s \rightarrow t \in E \quad \neg(i: \ t = y.m\ (\ldots\ldots) \ \wedge \ i \xrightarrow{call} m' \ \wedge \ m' \in \text{Transfers} \ \wedge \ s = m'_{ret})}{pt_H(s) \subseteq pt_H(t)}\ \text{[PropHost]}$$

图 10。容器访问模式的规则。

4.2.2 加载处理。图 9 展示了处理字段加载的规则。具体来说，如果方法 m 的返回值是从传递给 m（作为实参）的对象中加载的，那么由于来自 m 不同调用点的实参会在 m内部合并，因此可能导致精度损失。对于这种情况，我们将 m 的返回变量添加到 $cutReturns$，以此截断从 m 的返回变量到其调用点的左值变量的 PFG 边。与处理存储类似，我们定义 $tempLoads$ 来辅助添加快捷边。它是一组三元组，例如 $\langle to, base, f \rangle$，用于表示一次加载操作$to = base.f$。此外，为了处理嵌套调用（见第 3.2.3 节），我们会沿调用链生成并传播 tempLoads。

Cut。 在加载处理中，剪切边并传播临时加载的规则几乎相同，因此我们将其合并为一条规则 [CutPropLoad]以避免冗余。基于上文所述的洞见， [CutPropLoad] 会在可能的情况下，剪切从返回变量 m_{ret} 到调用点的 m的左值变量的返回边，并递归地构建临时加载，将其从被调用方传播到调用者。 [CutPropLoad] 中的最右侧前提和结论会将指向返回变量（to）的加载边记录在集合 $returnLoadEdges$中，正如本节末尾所述，该集合将用于确保可靠性。

Shortcut。 在 [ShortcutLoad], 中，我们只需为 $tempLoads$中的三元组生成快捷边 $\ldots$ 在其调用点，中我不要求 m_{ret} 只能由加载语句定义（这种限制会过滤掉很多 $\ldots$ 。然而，在某些情况下，并非所有返回值都来自字段加载（例如， m 同时包含 $x = y.f$和 $x = z$，其中 x 是 m_{ret}）。因此，我们定义[RelayEdge] 以确保可靠性。如果返回变量 m_{ret} 的值来自与字段加载无关的指针（即，不是 $returnLoadEdges$中边的源指针），那么我们会添加快捷边，将这些指针连接到 m调用点的左值变量。

4.3 容器访问模式

为了形式化容器访问模式，我们将容器元素分为三类（记为 c）：集合中的值、Map 中的键以及 Map 中的值。对于 Map 而言，这一类别尤其有用，因为它能让分析判断一个入口/出口 方法操作的是 Map 的键还是值。我们定义了第3.3节中介绍的三个输入关系。

- 入口是一组三元组，例如 m， k， c，这表示 m 是一种容器入口方法，传递给其第 k 个参数的值属于类别 c。
- 出口，一组形如 m， c 的对，表示 m 是一个容器出口方法，返回类别 c的值。

$$\frac{def_{m_{pk}} = \varnothing}{\langle m, k\rangle \rightarrowtail m_{pk}}\ [\textsc{Param2Var}] \qquad \frac{\forall i \in def_x : (i : x = y) \wedge (\langle m, _\rangle \rightarrowtail y) \quad i : x = y' \quad \langle m, k\rangle \rightarrowtail y'}{\langle m, k\rangle \rightarrowtail x}\ [\textsc{Param2VarRec}]$$

$$\frac{\langle m, _\rangle \rightarrowtail m_{ret}}{m_{ret} \in cutReturns}\ [\textsc{CutLFlow}] \qquad \frac{i \xrightarrow{call} m \quad i : r = _ \quad \langle m, k\rangle \rightarrowtail m_{ret}}{i_{ak} \rightarrow r \in E_{SC}}\ [\textsc{ShortcutLFlow}]$$

图 11. 局部流模式的规则。

• Transfers，一组 Transfer 方法，这些方法将宿主从调用点的接收者变量传递到其左值变量，例如，对于 $x = y.m(...)$，从 y 传递到 x。

现在，我们可以将容器访问模式形式化，如图10所示。

Cut。In [CutContainer], 我们只需从 Exit 中指定的方法移除所有返回边s .

Shortcut。 在 [ShortcutContainer], 中，我们基于作为前提的两个关系添加快捷边，

^{c} ^{c}记为 $h \Leftarrow s$和 $h \Rightarrow t$，这两个关系将宿主（容器）分别关联到其源和目标。前者表示 s 是宿主 h 的源，即， s 指向流入 h 的对象，而这些对象会成为类别为 c 的 h 的元素。后者表示 t是宿主 h的目标，即，它接收从 h返回的对象（类别为 c）。通过 [ShortcutContainer],，当两个前提均成立时，我们从 s向 t添加一条快捷边。

$_{c}$ $_{c}$[HostSource] 和 [HostTarget] 分别展示了如何推导 $h \Leftarrow s$和 $h \Rightarrow t$。推导这两个关系的关键是指针-宿主映射 pt_H，如第3.3节所介绍。图10中的底部四条规则描述了如何计算宿主相关指针的 pt_H。具体来说，如果x指向 o_i，则 [ColHost]和 [MapHost] 会将集合或映射对象 o_i（$t_1 <: t_2$ 表示 t_1 是 t_2的子类型）添加到 $pt_H(x)$ 中。通过 [TransferHost],，如果调用点 i调用了 Transfer 方法，那么我们会将与 i的接收者变量关联的宿主转移到 i的左值变量，例如，对于 $it = l.iterator()$，将宿主从 l转移到 it。通过 [PropHost],，我们沿 PFG 边传播宿主，但边是 Transfer 方法的返回边的情况除外。我们排除这些情况的原因是：(1) [TransferHost]已经处理了 Transfer 方法的返回值；(2) 不同的宿主可能流入同一个 Transfer 方法 m，如果不排除这些情况，这些宿主就会被传播到其所有 m的调用点的左值变量中，从而导致不精确性。

4.4 局部流模式

图11 给出了计算 $cutReturns$ 和 E_{SC} 局部流模式的规则。为了便于形式化表示，我们引入记号 $\rightarrowtail$，其定义为 [Param2Var] 和 [Param2VarRec]。$m,\ k \rightarrowtail x$ 表示 x 是方法 m中的局部变量，而 x所指向的对象来自 m的第 k个参数。注意， [Param2Var] 和 [Param2VarRec] 保证了当$m,\ k \rightarrowtail x$成立时，x 的值只能通过零个或多个局部赋值来自 m的参数，而不会来自其他源，例如字段加载和方法调用。基于这一记号，我们可以轻松定义局部流模式中需要截断或添加的边。

截断。在 [CutLFlow],中，如果我们发现返回变量 m_{ret}的值全部来自 m的参数，那么就将 m_{ret} 添加到 $cutReturns$中。

Shortcut。 在 [ShortcutLFlow], 成立的情况下，对于方法 m的每个调用点 i，我们从 k的第 i 个实参到其左值变量添加一条快捷边。

4.5 可靠性

本节基于前文给出的规则，给出 Cut-Shortcut 可靠性的证明概要。我们使用 E_{cut} 表示根据这些规则被截断的边集合。对于一个目标 t， $Source_t$ 表示

一组通过快捷边与 t 相连的源，即 $Source_t = \{s \mid s \to t \in E_{SC}\}$。我们使用 PA 表示指针分析，并添加下标以指代特定的分析，例如，PA_{ci} 表示上下文不敏感的指针分析，PA_{csc} 表示 Cut-Shortcut。对于分析 PA_a，$G_a = (N_a, E_a)$ 表示其 PFG。我们证明，PA_{csc} 能够为一个程序生成可靠的指向结果。

我们首先证明，从某个分配点的左值变量（即最先接收堆对象的指针，也就是流的起点）到某个 t 以被剪切的边结束的指针 e 在 E_{cut} 中必定经过某个 $s \in Source_t$。这意味着，即使我们截断边 e，流向 t 的对象仍可在运行时通过指向 e 的快捷边传播至 t 在 t 中。我们在引理4.1（针对由 $cutStores$剪切的边）和4.2（针对由 $cutReturns$剪切的边）中对此进行了形式化表述。

引理 4.1。以 P开头且以存储边 $y \to o_k.f \in E_{cut}$ 结束的动态指针流路径必定经过某个 $s \in Source_{o_k.f}$。

证明概要。假设 $x.f = y \in cutStores$和 x指向 o_k。如果 P以 $y \to o_k.f$结束，则根据[CutStore],，它必定经过传递给 y的某些实参。[CutStore] 和 [PropStore] 将沿着从 y到的调用链进行搜索，以找出最外层调用。P必定在其中某个最外层调用（例如 i）处经过 y对应的实参（例如a_y）。根据 [ShortcutStore]，o_k 必定属于 x在 i处对应实参的指向集合，因此 $a_y \in Source_{o_k.f}$。□

假设 1。我们假设，针对容器访问模式的输入关系入口和 *Transfers* 相对于我们所考虑的容器类（即 *JDK* 中的容器类）是完整的。

引理 4.2。在假设 *1*下，动态指针流路径 P 以返回边结束$m_{ret} \to r \in cutReturns$必须经过某个 $s \in Source_r$。

证明概要。我们分别讨论 $cutReturns$ 每种模式。

- 在字段访问模式中：$m_{ret} \in cutReturns$ 由 [CutPropLoad]派生。假设 r是调用 j的左值变量。不难看出，[CutPropLoad] 是一个归纳过程。$tempLoads$ 中的每个元素都源自一条最内层的加载语句（例如 i）。如果 P经过由 i生成的加载边，由于 $j_{ak} \mapsto base$，P必须经过 j_{ak}。根据 [ShortcutLoad]，$j_{ak} \in Source_r$；否则，根据 [RelayEdge],，其他类型的边将连接到 m_{ret}的后继节点。
- 在容器访问模式中：$m_{ret} \in cutReturns$ 源自 [CutContainer]。根据容器访问模式中出口的语义，r 应指向添加到相关容器（宿主）中的所有对象。根据假设1，入口和 Transfers 均已完成。根据 [ColHost]—[PropHost],pt_H 将包含相关指针的所有宿主，并且根据 [HostSource] 以及 [HostTarget],，源和目标将与这些宿主相关联。由于 r是一个从容器接收元素的目标，P必须流经传递给入口的某些实参，这些实参将被 [HostSource] 捕获，并属于 $Source_r$。
- 在局部流模式中：$m_{ret} \in cutReturns$ 源自 [CutLFlow]。$m, k \rightarrowtail m_{ret}$ 表示m_{ret}的指向集合仅来自 m的若干参数，包括 m_{pk}。因此，如果 P以 $m_{ret} \to r$结束，则它必然来自这些参数之一（例如 m_{pk}）。假设 r是调用 i的左值变量。在动态执行期间，i_{ak} 是m_{pk}的前驱，这意味着P会流经它。根据 [ShortcutLFlow]，$i_{ak} \in Source_r$。□

定理 4.3（PA_{csc}的可靠性）。在假设 *1*下，PA_{csc} 是可靠的。

证明概要。我们将证明，对于从起点（例如 s）到任意程序指针（例如 t）的任意动态指针流路径 P，t都可以从 s沿着 G_{csc}到达。假设 P中的节点为 $[s = p_0, p_1, ..., p_n = t]$，那么我们通过归纳法证明，对于 G_{csc} 上的任意 $i \in [0, n]$，s都可以到达 p_i。

起初，对于 $i = 0$，这显然成立，即 s 可以到达自身（p_0）。根据归纳法，我们证明：如果对于任意 $k \le i$（$i < n$），s 都能到达 p_k，那么它也能到达 p_{i+1}。考虑 P 上的边 $p_i \to p_{i+1}$。存在

两种情况：（1）G_{csc} 上的 $p_i \rightarrow p_{i+1}$ 未被截断；（2）它被截断，且 G_{csc} 包含与之对应的快捷边。对于情况（1），s 可以通过从 s 到 p_i 的路径，再加上 $p_i \rightarrow p_{i+1}$，在 G_{csc} 上到达 p_{i+1}。对于情况（2），根据引理 4.1 和 4.2，存在一个 $s_{p_{i+1}} \in Source_{p_{i+1}}$，其中 $s_{p_{i+1}}$ 是路径 $[p_0, p_1, ..., p_i]$ 中的一个节点。于是，根据归纳假设，s 可以到达 $s_{p_{i+1}}$；又因为 G_{csc} 上存在一条快捷边 $s_{p_{i+1}} \rightarrow p_{i+1}$，所以 s 也可以到达 p_{i+1}。因此，根据归纳法，我们证明了 s 可以在 G_{csc} 上到达任意 p_i，包括 t。这意味着，对于动态执行期间 t 指向的每个对象 o，PA_{csc} 都能确保 o 包含在 pt_{csc} 中（t）。因此，PA_{csc} 是可靠的。□

5 评估

本节将针对以下研究问题进行评估。

研究问题1。 Cut-Shortcut 与上下文不敏感指针分析相比表现如何？**研究问题2**。 Cut-Shortcut 与传统上下文敏感指针分析相比表现如何？**研究问题3**。 Cut-Shortcut 与目标相似的最先进选择性上下文敏感指针分析技术相比表现如何（即在保证良好精度的同时实现高效率）？

实现。为了全面评估 Cut-Shortcut 的有效性并展示其通用性，我们在两个截然不同的指针分析框架（最新版本）上实现了该方法：声明式的 Doop [DOOP 2022]（使用 Datalog 编写）和命令式的 Tai-e [Tai-e 2022；Tan and Li 2022]（使用 Java 编写）。Doop 是一个指针分析框架，在过去十年中一直被视为实现和评估 Java 指针分析的主流平台；Tai-e 则是一个较新的 Java 静态分析框架，配备了高效的指针分析系统。我们通过将反射相关类加入封闭世界，修复了 Doop 中的一个不可靠问题，从而提高了可靠性。由于 Datalog 禁止在递归环中使用否定，我们无法直接在 Doop 上实现图 9 中的规则 [CutPropLoad]（以及图 7 中的规则 [Return], [Propagate] 和 [Call]），而消除该环的变通方案会使分析成本高得难以承受。因此，我们省略了 Doop 中字段访问模式的加载处理。总体而言，Cut-Shortcut 的核心实现仅包含 Doop 上约 100 条 Datalog 规则，以及 Tai-e 上约 2000 行 Java 代码。我们将发布并维护 Tai-e 上 Cut-Shortcut 的源代码，网址为 https://github.com/pascal-lab/Tai-e；Doop 上的源代码已包含在我们的制品中。

对比分析。 我们将 Cut-Shortcut 与三类指针分析进行比较：上下文不敏感性（CI）、主流上下文敏感性和最先进的选择性上下文敏感性。CI 实际上是 Java 中速度最快的指针分析。对于传统上下文敏感性，我们选择了常用的二对象敏感（2obj） [Milanova 等人 2002, 2005] 和二类型敏感（2type） [Smaragdakis 等人 2011] 提出的分析。 2obj 具有很高的精度，而 2type 通常比 2obj 实现显著更好的可扩展性，同时保持相当的精度。对于选择性上下文敏感性，我们考虑了一种最先进的方法 Zipper^e [Li 等，2020a],，其效率与精度之间的权衡优于许多其他选择性方法。 Zipper [Li 等，2018a]（近期文献中经常将其评估为最先进的方法，具有相当的精度，但效率远低于Zipper^e）未被纳入考虑，因为对于我们评估的大多数程序而言，它无法实现可扩展性。

实验设置。 所有实验均在一台配备 128GB 内存的 Intel Xeon 2.2GHz 机器上进行。每项分析的时间预算设置为 2 小时。我们选取近期相关文献中用于评估的所有大型且复杂的 Java 程序 [Jeon 等，2019年、2018年；Jeong 等人，2017年；Li 等人，2018a、b、2020a； Lu 和 Xue，2019年；Tan 等人 2021]，但不包括那些无法执行的程序，因为我们无法对其进行召回率实验来评估可靠性。我们使用近期研究中普遍采用的大型 Java 程序库 JDK1.6 对这些程序进行分析 [He 等人，2021年、2022年； Jeon 等，2020； Jeon 和 Oh，2022； Lu 等人 2021]。

精度指标。为衡量精度，我们使用四种彼此独立且各自有用的客户端；在指针分析文献中，它们被广泛用作精度指标 [Jeon 等人 2019年，2020年；Jeong 等人 2017年；Li

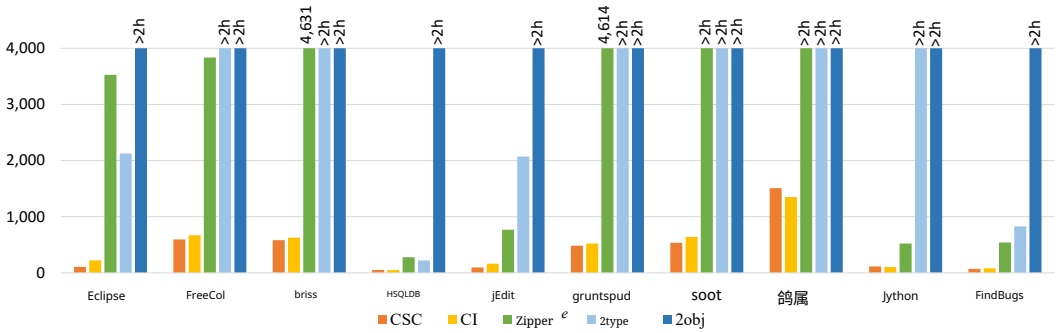


图 12. Cut-Shortcut 方法（CSC）、上下文不敏感（CI）、选择性二对象敏感（Zipper^e）以及传统上下文敏感（2obj、2type）指针分析所用的分析时间（秒），基于Doop。CSC 在 Tai-e 上也展现出与 CI 类似的效率优势（因篇幅有限，未展示图表），同时在Doop （表1）和Tai-e （表2）上均比 CI 获得了显著更高的精度。

等人 2018a，b，2020a；Lu 等，2021；Smaragdakis 等，2011，2014年；Tan 等人 2017]。这些客户端分别是类型转换解析分析（指标：可能失败的类型转换数量——#fail-cast）、方法可达性分析（指标：可达方法数量——#reach-mtd）、去虚拟化分析（指标：无法消歧为单态调用的虚拟调用点数量——#poly-call）以及调用图构建分析（指标：调用图边数量——#call-edge）。

5.1 研究问题1：Cut-Shortcut 与上下文不敏感（CI）的比较

在本节中，我们考察Cut-Shortcut 与速度最快的指针分析 CI 相比表现如何。但要确信Cut-Shortcut的效率优势可靠，我们必须先确认其可靠性。除第4.5节中的理论可靠性证明外，下面我们还通过召回率实验进一步验证其可靠性。

可靠性（召回率）。在召回率实验中，我们使用各个评估程序的默认测试执行它们（例如，对于 DaCapo基准测试 [Blackburn 等人 2006]），或使用我们输入的测试执行它们（例如，对于 GUI程序，我们通过点击与其交互，并记录结果，随后将结果提供在制品中）。然后，在这些输入下，我们动态记录程序执行期间的可达方法和调用图边（其他动态信息（如详细的指向关系）很难进行插桩），并检查其中有多少能够被我们的评估中的Cut-Shortcut 及其他分析召回（过近似）。结果表明，Cut-Shortcut能够召回其他可靠分析发现的几乎所有真实可达方法和调用图边。对于其他情况，在Doop 上，6 个程序总共遗漏了 20 条调用图边；在Tai-e上，3 个程序遗漏了 11 条边。经过人工检查，我们发现这些遗漏的边并不真实，而是由插桩工具错误记录的。因此，Cut-Shortcut 在召回率实验中实现了与其他分析相同的可靠性。再加上理论上的可靠性证明，我们可以相信其余的效率和精度结果是可靠的。

效率与精度。 图 12 以图形方式展示了所有分析在Doop(Tai-e的分析结果因篇幅限制未展示）。表1 (针对Doop)和表2(针对Tai-e)完整而详细地列出了所有效率和精度结果。

我们可以看到，持续集成的效率显著高于上下文敏感分析；在 Doop 和 Tai-e 上，对于 10 个程序中的 7 个，Cut-Shortcut（CSC）的运行速度甚至快于持续集成。对于其余程序，Cut-Shortcut 的速度与持续集成相当，或仅略慢一些。Cut-Shortcut 出色的效率源于其方法和精度改进。首先，与上下文敏感性（通过在不同上下文中克隆方法和堆对象来提高精度）不同，Cut-Shortcut 通过修改 PFG 来提高精度，而这通常

表1. 在声明式 Doop 框架上，上下文不敏感（持续集成，CI）、传统上下文敏感（2obj、2type）和选择性二对象敏感（Zipper^e）指针分析，以及我们的 Cut-Shortcut 方法（CSC）的效率和精度结果。对于所有数值，越小越好。

程序	分析	时间 (s)	精度指标 #fail-cast	#reach-mtd	#poly-call	#call-edge
Eclipse	CI	223	5,077	23,549	10,744	183,478
	2obj	>2h	–	–	–	–
	2type	2,126	4,334	22,928	9,955	164,742
	Zipper^e	3,526	4,198	23,206	10,049	170,172
	CSC	107	4,082	23,437	10,475	170,740
FreeCol	CI	671	9,971	44,838	14,626	309,033
	2obj	>2h	–	–	–	–
	2type	>2h	–	–	–	–
	Zipper^e	3,836	8,492	44,159	12,630	265,446
	CSC	595	7,435	44,472	12,354	265,241
briss	CI	627	8,000	41,787	12,853	294,428
	2obj	>2h	–	–	–	–
	2type	>2h	–	–	–	–
	Zipper^e	4,631	6,650	41,381	11,640	261,081
	CSC	582	6,129	41,563	11,302	261,029
HSQLDB	CI	50	1,723	11,143	1,914	61,929
	2obj	>2h	–	–	–	–
	2type	221	1,276	10,769	1,598	54,652
	Zipper^e	278	1,209	10,793	1,618	54,725
	CSC	52	1,295	10,986	1,744	56,513
jEdit	CI	163	4,349	25,157	6,336	149,761
	2obj	>2h	–	–	–	–
	2type	2,071	3,458	24,475	5,493	124,798
	Zipper^e	768	3,401	24,552	5,594	125,883
	CSC	97	3,188	24,629	5,880	125,671

程序	分析	时间 (s)	精度指标 #fail-cast	#reach-mtd	#poly-call	#call-edge
gruntspud	CI	523	6,982	39,555	12,328	272,944
	2obj	>2h	–	–	–	–
	2type	>2h	–	–	–	–
	Zipper^e	4,614	5,846	39,092	11,277	230,831
	CSC	484	5,415	39,354	11,826	230,921
soot	CI	641	16,724	32,696	16,877	418,910
	2obj	>2h	–	–	–	–
	2type	>2h	–	–	–	–
	Zipper^e	>2h	–	–	–	–
	CSC	536	10,991	32,497	16,257	418,610
鸽属	CI	1,352	10,824	56,431	17,679	413,829
	2obj	>2h	–	–	–	–
	2type	>2h	–	–	–	–
	Zipper^e	>2h	–	–	–	–
	CSC	1,508	5,606	56,050	17,135	413,826
Jython	CI	104	2,381	12,623	2,937	119,227
	2obj	>2h	–	–	–	–
	2type	>2h	–	–	–	–
	Zipper^e	523	2,193	12,423	2,796	115,639
	CSC	115	1,886	12,325	2,904	115,159
FindBugs	CI	81	3,488	16,985	4,575	106,233
	2obj	>2h	–	–	–	–
	2type	827	2,472	16,512	3,866	88,923
	Zipper^e	541	2,495	16,696	4,034	93,485
	CSC	70	2,568	16,782	4,287	93,800

只会带来很小的开销。其次，与现有的选择性上下文敏感性方法不同，PFG 不需要预分析：它在即时构建的 PFG 上执行上下文不敏感指针分析算法。第三，Cut-Shortcut 可防止大量虚假的指向结果在分析过程中传播到程序的其他部分。与精度提升带来的效率收益相比，PFG 操作的分析成本通常可以忽略不计；因此，Cut-Shortcut 的效率通常优于持续集成。

现在让我们考察精度。对于所有精度指标，数值越小表示结果越好。在 Doop 和 Tai-e 上，Cut-Shortcut 在所有精度指标上的表现都优于 CI，且精度提升通常较为显著，尤其是在 #fail-cast 和 #call-edge 方面。Cut-Shortcut 通过根据模式修改 PFG 来影响精度，而这些改进表明，我们为 Cut-Shortcut 设计的三种模式有助于提高精度。

有人可能会想知道，Cut-Shortcut 的哪种模式对精度的影响最大？为回答这一问题，我们在 Tai-e 上针对 Cut-Shortcut 开展了三组实验，以衡量三种模式对不同客户端的影响。对于每组实验，仅启用一种模式，禁用另外两种模式。为衡量影响，我们计算每种单独模式相对于持续集成的精度提升，并将其除以三种模式共同启用时的总体提升。我们发现，不同客户端受到的模式影响各不相同。例如，对于客户端 #fail-cast，字段访问模式、容器模式和局部流模式平均分别将精度提升了 11.9%、75.8% 和 11.8%；而对于客户端 #reach-mtd，分别提升了 53.2%、40.5% 和 2.0%。（请注意，三个百分比之和小于 100%，因为三种模式结合使用时，可以通过减少分析过程中的虚假值传播，进一步提升各自的精度。）

表2。上下文不敏感（持续集成，CI）、传统上下文敏感（2obj、2type）、选择性二对象敏感（Zipper^e）指针分析，以及我们在命令式 Tai-e 框架上提出的 Cut-Shortcut 方法（CSC）的效率和精度结果。所有数值均为越小越好。

程序	分析	时间 (s)	#失败- cast	#可达- cast	#多态- mtd	#调用- call	#调用- edge
Eclipse	CI	21	5,091	23,920	10,610	184,294	
	2obj	5,218	3,659	23,195	9,690	164,143	
	2type	261	4,230	23,338	9,876	165,823	
	Zipper^e	557	4,154	23,583	9,907	170,947	
	CSC	12	3,977	23,746	10,198	171,556	
FreeCol	CI	54	9,779	46,896	15,496	323,141	
	2obj	>2h	–	–	–	–	
	2type	>2h	–	–	–	–	
	Zipper^e	179	7,199	46,064	13,546	279,502	
	CSC	40	6,511	46,261	13,543	279,839	
briss	CI	50	7,815	41,851	12,755	294,038	
	2obj	>2h	–	–	–	–	
	2type	>2h	–	–	–	–	
	Zipper^e	179	6,098	41,427	11,575	261,811	
	CSC	58	5,669	41,462	11,891	262,948	
HSQLDB	CI	4	1,674	11,588	1,817	64,393	
	2obj	>2h	–	–	–	–	
	2type	25	1,094	11,185	1,472	56,712	
	Zipper^e	26	1,027	11,214	1,496	56,827	
	CSC	3	1,169	11,335	1,618	57,986	
jEdit	CI	15	4,113	25,261	6,339	150,159	
	2obj	7,192	2,578	24,214	4,933	121,896	
	2type	407	3,104	24,263	5,097	122,782	
	Zipper^e	39	3,042	24,342	5,212	123,783	
	CSC	10	2,913	24,641	5,825	131,248	

程序	分析	时间 (s)	#fail-cast	#reach-cast	#poly-mtd	#调用-call	#调用-edge
gruntspud	CI	44	6,775	39,800	12,342	274,872	
	2obj	>2h	–	–	–	–	
	2type	>2h	–	–	–	–	
	Zipper^e	162	5,308	39,201	10,978	232,046	
	CSC	39	5,035	39,444	11,675	233,281	
soot	CI	107	16,659	32,918	16,372	415,728	
	2obj	>2h	–	–	–	–	
	2type	>2h	–	–	–	–	
	Zipper^e	1,845	12,901	32,178	13,835	281,751	
	CSC	51	10,509	32,660	14,152	283,562	
鸽属	CI	117	10,492	56,787	17,700	425,149	
	2obj	>2h	–	–	–	–	
	2type	>2h	–	–	–	–	
	Zipper^e	1,163	8,257	55,973	15,931	346,346	
	CSC	122	7,862	56,427	16,963	347,537	
Jython	CI	11	2,421	13,058	2,951	121,528	
	2obj	>2h	–	–	–	–	
	2type	>2h	–	–	–	–	
	Zipper^e	42	1,898	12,617	2,597	111,866	
	CSC	11	1,864	12,683	2,884	113,364	
FindBugs	CI	6	3,391	17,353	4,462	107,339	
	2obj	1,369	1,984	16,821	3,560	88,890	
	2type	83	2,379	16,874	3,752	90,018	
	Zipper^e	22	2,453	17,099	3,917	95,543	
	CSC	5	2,215	17,099	4,167	92,451	

5.2 研究问题2：Cut-Shortcut 与传统上下文敏感性

在本节中，我们将研究 Cut-Shortcut 相较于主流的传统上下文敏感指针分析方法 2obj 和 2type 的表现。

效率。如表 1 和表 2 所示，在 Doop 上，2obj 对全部 10 个程序都超时；在 Tai-e 上，10 个程序中有 7 个超时。2type 的可扩展性优于 2obj，但在 Doop 和 Tai-e 上，它仍无法扩展到 6 个程序。相比之下，Cut-Shortcut 可以完成对所有程序的分析。对于 2obj 和 2type 能够扩展到的程序，Cut-Shortcut 的运行速度明显更快。对于 2obj，Cut-Shortcut 在 Tai-e 上的平均速度快了 475.9×（2obj 在 Doop 上对所有程序都不具备可扩展性，因此我们无法计算其加速比）；对于 2type，Cut-Shortcut 在 Doop 和 Tai-e 上的平均加速比分别为 14.3× 和 21.8×。总之，与传统上下文敏感性相比，Cut-Shortcut 的效率显著更高，可扩展性也更强。

精度。当 2obj 具备可扩展性时（在 Tai-e 的 10 个程序中有 3 个满足此条件），在所有精度指标上，它都比包括 Cut-Shortcut 在内的其他分析更精确。当 2type 具备可扩展性时（在 Doop 和 Tai-e 的 10 个程序中分别有 4 个满足此条件），它通常比 Cut-Shortcut 更精确，但在 Doop 的 4 个程序中有 2 个以及 Tai-e 的 4 个程序中有 3 个上，其 #fail-cast 表现更差。这些结果证明了传统上下文敏感性方法具有很高的精度；然而，由于可扩展性较差，我们无法在许多程序中依赖它们来提高精度。

5.3 RQ3：Cut-Shortcut 与最先进的选择性上下文敏感性

在本节中，我们将 Cut-Shortcut 与 Zipper^e [Li 等，2020a], 这一最先进的选择性上下文敏感方法进行比较。我们采用 Zipper^e 在 [Li 等，2020a],中使用的默认配置，该配置经过良好调优，在效率和精度之间实现了很好的权衡。

表3。 De在 Doop（左半部分）和 Tai-e（右半部分）上，Zipper^e 与 Cut-Shortcut 的尾部比较 部分）。

程序	Zipper^e 总计时间	Pre-分析	Main 分析	已选方法	CSC Time	涉及方法	重叠方法	Zipper^e 总计时间	Pre-分析	Main 分析	已选方法	CSC Time	涉及方法	重叠方法
Eclipse	3,526	251	3,275	3,616	107	3,827	42.8%	557	29	528	3,430	12	4,168	34.3%
FreeCol	3,836	992	2,844	4,497	595	7,650	27.7%	179	109	70	3,855	40	7,211	22.7%
briss	4,631	974	3,657	3,931	582	7,208	24.3%	179	100	79	3,206	58	6,990	19.0%
HSQLDB	278	57	221	1,846	52	1,914	41.0%	26	7	19	1,494	3	1,819	32.8%
jEdit	768	214	554	2,811	97	5,241	25.2%	39	28	11	2,245	10	4,241	22.3%
gruntspud	4,614	1,066	3,548	4,058	484	6,980	25.8%	162	101	61	3,432	39	6,506	22.3%
soot	>2h	806	>2h	9,802	536	4,585	60.6%	1,845	121	1,724	12,436	51	6,072	67.2%
鸽属	>2h	1,936	>2h	6,137	1,508	9,322	27.0%	1,163	352	811	4,247	122	9,072	19.7%
Jython	523	324	199	786	115	2,504	14.9%	42	33	9	1,534	11	2,557	20.3%
FindBugs	541	93	448	2,438	70	3,037	36.4%	22	11	11	2,188	5	2,909	33.0%

效率。Zipper^e 的执行由三个部分组成：预分析（上下文不敏感指针分析）、Zipper^e 本身（根据其策略选择一组方法）以及主分析（仅对选定方法应用上下文敏感性的指针分析）；因此，Zipper^e 的耗时是这三个部分执行时间之和。表1和表2清楚地表明，在 Doop 和 Tai-e 上，Zipper^e 都比 2obj 和 2type 具有更好的可扩展性且速度更快。在 Doop 上，Zipper^e 可扩展至10个程序中的8个；在 Tai-e 上，Zipper^e 可扩展至所有程序。尽管如此，就效率而言，Cut-Shortcut 仍然优于 Zipper^e：对于 Zipper^e 能够扩展的程序，Cut-Shortcut 在 Doop 上的平均加速比为 10.3×，在 Tai-e 上为 12.5×。为了全面比较 Zipper^e 和 Tai-e 的效率，我们在表3中详细列出了 Zipper^e 和 Cut-Shortcut 的耗时；表格左半部分和右半部分分别展示 Doop 和 Tai-e 上的数据。对于 Zipper^e，列“总计时间”列出 Zipper^{e}三个部分的总分析时间（与图1和图2中相同）；列“预分析”展示 CI 和 Zipper^e 本身的分析时间之和；列“主分析”给出 Zipper^{e}引导的上下文敏感指针分析的分析时间。如我们所见，Zipper^e 的预分析成本平均占 Zipper^{e}总耗时的25.6%（Doop）和44.8%（Tai-e）；即使只考虑 Zipper^{e}的主分析，Cut-Shortcut 在 Doop 上仍快 8.4×，在 Tai-e 上快 10×于 Zipper^e，这再次证明了 Cut-Shortcut 的高效率。

精度。对于失败类型转换，Cut-Shortcut 在 Doop 的所有程序（findbugs 除外）以及 hsqldb 的所有程序上都优于 Zipper^e。对于其余精度指标，Cut-Shortcut 仍可与 Zipper^e 相媲美（尽管 Cut-Shortcut 在分析速度方面具有显著优势）。对于某些程序，例如 Doop 上的 jython，Cut-Shortcut 甚至在 4 项精度指标中的 3 项上优于 Zipper^e。我们预计，未来基于 Cut-Shortcut 原理设计更多程序模式后，可以进一步提高 Cut-Shortcut 的精度（和效率）。

尽管 Zipper^e 和 Cut-Shortcut 以根本不同的方式提高精度，但人们可能会想知道，Zipper^e 所选择的方法是否与 Cut-Shortcut所识别的切边和快捷边中涉及的方法相同。表 3 详细回答了这一问题。平均而言（包括 Doop 和 Tai-e），Cut-Shortcut 会考虑 5191 个方法（这些方法涉及切边和快捷边），占每个程序所有可达方法的 17%；而 Zipper^e 选择了 3899 个方法，并且由Cut-Shortcut 所识别的边中只有 31% 涉及的方法也被Zipper^e 选中（见“重叠方法”列）。

6 相关工作

上下文敏感性对于全程序 Java 指针分析发挥着至关重要的作用，我们已在前文讨论了部分相关工作。其他相关研究将在下文介绍。

近年来，人们提出了许多选择性上下文敏感方法，以在效率和精度之间实现权衡，尤其是针对规模庞大且复杂的 Java 程序。我们可以从两个方面理解“选择性”：选择更有效的上下文元素来分析每个方法（选择上下文元素），以及选择一组需要进行上下文敏感性分析的程序方法（选择程序方法）。下面我们将依次讨论这两类选择性上下文敏感方法。

选择上下文元素。传统的（k-限制）上下文敏感性使用 k 连续的上下文元素，例如 $[c_k..., c_2, c_1]$ 来分析方法 m，其中 c_1 是 m的调用点，而 c_2 是包含该方法的方法的调用点 c_1，以此类推。然而，Tan 等人 [2016] 发现，这种方法可能产生许多对提高精度没有作用的上下文元素，同时占用由 k限制的上下文元素槽位。因此，Tan 等人 [2016] 提出了一种方法，通过利用他们提出的所谓对象分配图来识别这类冗余上下文元素。类似地，Jeon 等人 [2018] 开发了一种名为上下文隧穿的机器学习方案，用于选择这类对精度无用的上下文元素，以实现更有效的分析。此外，在上下文隧穿的基础上， [Jeon 和 Oh 2022] 提出了一种有趣的方法，将对象敏感性转换为调用点敏感性，并证明后者可以模拟前者（但反之不成立）。

选择程序方法。传统上下文敏感性会将上下文统一应用于给定程序中的每个方法。然而，对于某些程序方法而言，上下文敏感性并不能帮助提高精度，只会带来额外的分析成本。因此，研究人员提出选择一组有助于提高分析精度的方法，仅对这些方法应用上下文敏感性，而在分析其余方法时采用上下文不敏感的方式。为实现效率与精度之间良好的权衡，总体原则是选择对精度至关重要 [Li 等，2018a] 但不会威胁可扩展性的方法 [Li 等，2020a]。为此，已有多种选择策略：依赖参数化启发式方法（其阈值基于专家经验） [Hassanshahi 等人，2017年；Smaragdakis 等人 2014], 机器学习方法 [Jeon 等人，2019年、2020年；Jeong 等人 2017], 抽象内存容量 [Li 等，2018b], 程序模式 [Li 等，2018a、2020a]，或能够利用上述方法优势的方案 [Tan 等人 2021]；此外，还可以将上下文敏感性应用于选定的变量（而非方法） [He 等人，2021年；Lu 和 Xue 2019] ，其依据是 CFL 可达性 [Reps 1998]。

混合上下文敏感性。 在一些指针分析中，同一方法的上下文元素可能有所不同。Kastrinis 和 Smaragdakis [2013] 提出了一种混合上下文敏感性方法，其中一个方法可以同时采用调用点敏感性和对象敏感性进行分析；在某些情况下，这种组合可能比单一类型的上下文元素产生更有效的结果。 Thakur 和 Nandivada [2020] 将对象敏感性上下文与层级汇总的相关值上下文（这是他们早期工作中提出的一种上下文抽象 [Thakur 和 Nandivada 2019]）结合起来分析每个方法，从而获得比单独使用其中任一种方法更精确的结果。

无论是上述选择性上下文敏感性方法还是混合上下文敏感性方法，它们仍然依赖上下文敏感性的核心思想：复制（一组）方法，并在不同（类型的）上下文下分别分析其中的程序元素。我们的方法从根本上不同于此，它通过切断精度损失边，并在 PFG 上直接从源连接快捷边至目标，来模拟上下文敏感性的效果，正如全文所述。

其他相关工作。 在图修改方面，Li 等人 [2020b， 2022] 提出了一种算法，用于简化交错Dyck可达性框架的输入图，该框架能够表示各种分析问题（例如，污点分析 [Huang 等人 2015],需求驱动的上下文敏感指针分析 [Späth 等， 2016；Sridharan 和 Bodík 2006]）。Cut-Shortcut 与之在目标问题和底层图结构上都不同。谈到如何消除 Java 中虚假的指向信息，除上下文敏感性之外，De 和 D’Souza [2012] 提出了一种流敏感方法，可对堆对象部分执行强更新。他们通过以下方式实现这一点

通过访问路径表示指向关系 [Fink 等，2008 年；Kanvar 和 Khedker 2016]；不同的是，Cut-Shortcut 及许多其他方案通常通过类似 PFG 的图来表示指向信息。轨迹划分 [Blanchet 等，2003 年；Rival 和 Mauborgne 2007]对代码 if (c) {S1} else {S2} <rest> 的分析方式，就如同代码为 if (c) {S1;<rest>} else {S2;<rest>}。 <rest> 会被克隆两次，分别放入两个分支中，然后将原始起始位置处对 <rest>的合并延迟到 <rest>末尾。这与通过克隆代码实现上下文敏感性的思想更为相似。Cut-Shortcut 的本质不同之处在于，我们不会克隆代码；此外，我们不是延迟合并，而是通过移除那些使不精确性传播到方法外部的边来避免合并，从而提高精度。Zhang 等人 [2014] 提出了一种结合自顶向下与自底向上分析的混合方法。自顶向下和自底向上方法是许多静态分析采用的通用思路。与其他主流 Java 指针分析一样 [Smaragdakis 和 Balatsouras 2015], , Cut-Shortcut 也采用自顶向下的方式；如果将自底向上分析视为一种摘要，那么我们添加快捷边这一做法的高层思想可视为一种指针特定摘要。然而，Cut-Shortcut 的目标问题及其底层具体方法，都与 [Zhang 等人 2014]相似之处不大。最后，还有一些工作通过合并堆对象来加速上下文敏感的 Java 指针分析 [Chen 等，2021 年；Tan 等人 2017], , 这些工作与 Cut-Shortcut 正交。

7 结论

在过去 20 年中，上下文敏感性一直被认为是提高 Java 指针分析精度的最有效技术之一。然而，它会带来巨大的效率开销，尤其是对于大型且复杂的程序。选择性上下文敏感方法虽然能够部分缓解这一问题，但尚未解决其效率瓶颈，因为很难正确选择那些对精度至关重要、却又不会威胁可扩展性的方法；因此，这些方法仍然可能需要计算和维护大量上下文，以区分虚假对象流，最终只能取得有限的效果。

为了解决这一困境，我们提出了一种名为 Cut-Shortcut 的根本不同的方法，旨在不应用上下文的情况下模拟上下文敏感性的效果。其实现方式是在典型的指针分析 PFG 中，剪去会导致精度损失的边，并添加用于连接相关指针的快捷边。我们利用三种程序模式，并依据 Cut-Shortcut 的原理设计相应规则，从而实现了 Cut-Shortcut。随后，我们对其进行了形式化并证明了其可靠性。为了全面验证 Cut-Shortcut 的有效性（以及作为一种原则性方法的通用性），我们针对两个最先进的 Java 指针分析框架分别实现了两个版本：一个使用 Datalog 实现，用于声明式的 Doop；另一个使用 Java 实现，用于命令式的 Tai-e。实验结果极具前景：在两个框架中，对于大多数评估程序，Cut-Shortcut 的运行速度甚至快于上下文不敏感性，同时还能获得显著更高的精度（接近上下文敏感性的高精度）。鉴于这些令人鼓舞的结果，预计还可以在 Cut-Shortcut 的基础上进一步探索更多程序模式或洞见。我们希望，这一方法能够为未来开发更有效的 Java 指针分析提供一些新的视角。

致谢

我们感谢指导人 Alexey Loginov 和匿名审稿人提出的宝贵意见。本工作得到国家自然科学基金项目编号 62025202、62002157 和 61932021，以及江苏省自然科学基金前沿技术计划项目编号 BK20202001 的资助。作者还要感谢中国江苏新型软件技术与产业化协同创新中心提供的支持。

数据可用性声明

我们提供了一个制品 [Ma 等人 2023]，用于复现第 5 节中的所有表格和图，以及第 5.1 节中的召回率实验。您可以从 https://doi.org/10.5281/zenodo.7808384 下载该制品，并按照 README.pdf 中的说明使用它复现结果。

参考文献

Lars Ole Andersen. 1994. *C* 编程语言的程序分析与特化. Ph. D. 学位论文.

Anastasios Antoniadis、Nikos Filippakis、Paddy Krishnan、Raghavendra Ramesh、Nicholas Allen 和 Yannis Smaragdakis. 2020年。Java 企业应用程序静态分析：框架与缓存——房间里的大象。在 第 *41* 届 *ACM SIGPLAN* 编程语言设计与实现会议论文集 （英国伦敦）（*PLDI 2020*）。计算机协会，纽约州纽约市，美国，794–807。https://doi.org/10.1145/3385412.3386026

Steven Arzt、Siegfried Rasthofer、Christian Fritz、Eric Bodden、Alexandre Bartel、Jacques Klein、Yves Le Traon、Damien Octeau 和 Patrick McDaniel。2014年。FlowDroid：面向 Android 应用程序的精确上下文、流、字段、对象敏感和生命周期感知污点分析。在 第 *35* 届 *ACM SIGPLAN* 编程语言设计与实现会议论文集 （爱丁堡，英国）（*PLDI '14*）。计算机协会，纽约州纽约市，美国，259–269。https://doi.org/10.1145/2594291.2594299

Stephen M. Blackburn、Robin Garner、Chris Hoffmann、Asjad M. Khan、Kathryn S. McKinley、Rotem Bentzur、Amer Diwan、Daniel Feinberg、Daniel Frampton、Samuel Z.Guyer、Martin Hirzel、Antony L.Hosking、Maria Jump、Han Bok Lee、J. Eliot B. Moss、Aashish Phansalkar、Darko Stefanovic、Thomas VanDrunen、Daniel von Dincklage 和 Ben Wiedermann。2006 年。DaCapo基准测试：Java 基准测试的开发与分析。载于 第 *21* 届 *ACM SIGPLAN* 面向对象编程、系统、语言与应用年度会议论文集，*OOPSLA 2006*，*2006* 年 *10* 月 *22* 日至 *26* 日，美国俄勒冈州波特兰，Peri L. Tarr 和 William R. Cook（编）。ACM，第 169–190 页。https://doi.org/10.1145/1167473.1167488Bruno Blanchet、Patrick Cousot、Radhia Cousot、Jérôme Feret、Laurent Mauborgne、Antoine Miné、David Monniaux 和 Xavier Rival。2003 年。大型安全关键软件的静态分析器。载于 *ACM SIGPLAN 2003* 年编程语言设计与实现会议论文集，*2003* 年，*2003* 年 *6* 月 *9* 日至 *11* 日，美国加利福尼亚州圣迭戈，Ron Cytron 和 Rajiv Gupta（编）。ACM，第 196–207 页。https://doi.org/10.1145/781131.781153

Martin Bravenboer 和 Yannis Smaragdakis。2009年。复杂指向分析的严格声明式规范。收录于第 *24* 届 *ACM SIGPLAN* 面向对象编程、系统、语言与应用年度会议论文集，*OOPSLA 2009*，*2009* 年 *10* 月 *25—29* 日，美国佛罗里达州奥兰多，Shail Arora 和 Gary T. Leavens（编）。ACM，243–262 页。https://doi.org/10.1145/1640089.1640108

Yuandao Cai、Peisen Yao 和 Charles Zhang。2021年。Canary：线程间值流漏洞的实用静态检测。收录于第 *42* 届 *ACM SIGPLAN* 编程语言设计与实现国际会议论文集（线上举办，加拿大）（*PLDI 2021*）。计算机协会，纽约州纽约市，1126–1140 页。https://doi.org/ 10.1145/3453483.3454099

Satish Chandra, Stephen J. Fink, 以及 Manu Sridharan. 2009. Snugglebug：一种实现最弱前置条件的强大方法. 载于第*30*届 *ACM SIGPLAN* 编程语言设计与实现会议论文集（爱尔兰都柏林）（*PLDI '09*）。计算机协会，纽约州纽约市，363–374。https://doi.org/10.1145/1542476.1542517 Yifan Chen、Chenyang Yang、Xin Zhang、Yingfei Xiong、Hao Tang、Xiaoyin Wang 和 Lu Zhang。2021年。通过合并库事实加速 Datalog 中的程序分析。载于静态分析：第*28*届国际研讨会，*SAS 2021*，美国伊利诺伊州芝加哥，*2021*年*10*月*17*日至*19*日，论文集（美国伊利诺伊州芝加哥）。Springer-Verlag，柏林、海德堡，77–101。https://doi.org/10.1007/978-3-030-88806-0_4

Arnab De Raj、Jeenu James 和 Jayant Haritsa。2012。发表于 *PLDI 2012* – 面向对象编程、语言与应用年度会议论文集的 Spa 敏感指针分析. 载于 *ACM SIGPLAN 2012* – 面向对象编程、语言与应用年度会议论文集，第*47*卷，第*10*期，第9号文章（2008年5月），34页。https://doi.org/10.1145/ 1348250.1348255 Springer Berlin Heidelberg，德国柏林，665–687。

DOOP。2022年。Java 指针与污点分析框架。https://bitbucket.org/yanniss/doop。页

ratik Fegade 和 Christian Wimmer。2020年。利用语义模型对数据结构进行可扩展指针分析。发表于第 *29* 届编译器构造国际会议论文集（美国加利福尼亚州圣迭戈）（*CC 2020*）。计算机协会，美国纽约州纽约市，39–50。https://doi.org/10.1145/3377555.3377885

Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, 和 Emmanuel Geay. 2008. 存在所有时间有效类型状态验证。*ACM 软件工程与方法学汇刊*，第2期，

Neville Grech 和 Yannis Smaragdakis。2017年。P/Taint：统一的指向分析和污点分析。*PACMPL*1，OOPSLA（2017年），102:1–102:28。https://doi.org/10.1145/3133926

Behnaz Hassanshahi、Raghavendra Kagalavadi Ramesh、Padmanabhan Krishnan、Bernhard Scholz 和 Yi Lu。2017年。面向大型代码库的高效可调选择性指向分析。载于第六届 ACM SIGPLAN 程序分析领域最新进展国际研讨会论文集（西班牙巴塞罗那）（SOAP 2017）。计算机协会，美国纽约州纽约市，13–18。https://doi.org/10.1145/3088515.3088519 Dongjie He、Jingbo Lu、Yaoqing Gao 和 Jingling Xue。2021年。通过利用对象包含关系和可达性加速对象敏感指针分析。载于第35届欧洲面向对象编程会议（ECOOP 2021）（莱布尼茨国际信息学论文集（LIPIcs），第194卷），Anders Møller 和 Manu Sridharan（编）。达格斯图尔城堡–莱布尼茨信息学中心，德国达格斯图尔，16:1–16:31。https://doi.org/10.4230/LIPIcs.ECOOP.2021.16 Dongjie He、Jingbo Lu、Yaoqing Gao 和 Jingling Xue。2022年。模块化选择上下文敏感性以加速对象敏感指针分析。IEEE 软件工程汇刊（2022年），1–1。https://doi.org/10.1109/TSE. 2022.3162236 Kihong Heo、Hakjoo Oh 和 Kwangkeun Yi。2017年。面向可扩展且精确的全局静态分析，对上下文敏感性与八边形域进行选择性合取。Softw. Pract. Exp. 47，11（2017年），1677–1705。https://doi.org/10.1002/spe.2493 Wei Huang、Yao Dong、Ana L. Milanova 和 Julian Dolby。2015年。面向 Android 的可扩展且精确的污点分析。载于2015年国际软件测试与分析研讨会论文集，ISSTA 2015，美国马里兰州巴尔的摩，2015年7月12–17日，Michal Young 和 Tao Xie（编）。ACM，106–117。https://doi.org/10.1145/2771783.2771803 Minseok Jeon、Sehun Jeong、Sungdeok Cha 和 Hakjoo Oh。2019年。用于数据驱动程序分析的带析取模型的机器学习算法。ACM 程序语言汇刊 41，2，文章13（2019年6月），41页。https://doi.org/10.1145/3293607 Minseok Jeon、Sehun Jeong 和 Hakjoo Oh。2018年。通过数据驱动的上下文隧穿实现精确且可扩展的指向分析。ACM程序设计语言会议录 2，OOPSLA，文章140（2018年10月），29页。https://doi.org/10.1145/3276510 Minseok Jeon、Myungho Lee 和 Hakjoo Oh。2020年。无需手工设计应用特定特征，学习基于图的指针分析启发式方法。ACM程序设计语言会议录 4，OOPSLA，文章179（2020年11月），30页。https://doi.org/10.1145/3428247 Minseok Jeon 和 Hakjoo Oh。2022年。CFA 回归：即使对于面向对象的程序，调用点敏感性也可能优于对象敏感性。ACM程序设计语言会议录 6，POPL，文章58（2022年1月），29页。https://doi.org/10.1145/3498720 Sehun Jeong、Minseok Jeon、Sung Deok Cha 和 Hakjoo Oh。2017年。面向指向分析的数据驱动上下文敏感性。PACMPL 1，OOPSLA（2017年），100:1–100:28。https://doi.org/10.1145/3133924 Vini Kanvar 和 Uday P. Khedker。2016年。静态分析的堆抽象。ACM Comput. Surv. 49，2，文章29（2016年6月），47页。https://doi.org/10.1145/2931098 George Kastrinis 和 Yannis Smaragdakis。2013年。指向分析的混合上下文敏感性。载于 ACM SIGPLAN 编程语言设计与实现会议，PLDI '13，美国华盛顿州西雅图，2013年6月16–19日，Hans- Juergen Boehm 和 Cormac Flanagan（编）。ACM，423–434。https://doi.org/10.1145/2491956.2462191 Ondřej Lhoták 和 Laurie Hendren。2006年。上下文敏感的指向分析：是否值得？载于国际编译器构造会议。Springer，47–64。Yue Li、Tian Tan、Anders Møller 和 Yannis Smaragdakis。2018a。面向指针分析的精度引导上下文敏感性。ACM程序设计语言会议录 2，OOPSLA，文章141（2018年10月），29页。https://doi.org/10.1145/3276511 Yue Li、Tian Tan、Anders Møller 和 Yannis Smaragdakis。2018b。采用自调优上下文敏感性的可扩展性优先指针分析。载于第12届欧洲软件工程会议与 ACM SIGSOFT 软件工程基础研讨会联合会议论文集（ESEC/FSE）。ACM，129–140。https://doi.org/10.1145/3236024.3236041 Yue Li、Tian Tan、Anders Møller 和 Yannis Smaragdakis。2020a。指针分析选择性上下文敏感性的原则化方法。ACM 程序语言汇刊 42，2，文章10（2020年5月），40页。https://doi.org/10.1145/3381915 Yue Li、Tian Tan、Yifei Zhang 和 Jingling Xue。2016年。程序定制：按顺序标准进行切片。载于第30届欧洲面向对象编程会议（ECOOP 2016）（莱布尼茨国际信息学论文集（LIPIcs），第56卷），Shriram Krishnamurthi 和 Benjamin S. Lerner（编）。达格斯图尔城堡–莱布尼茨信息学中心，德国达格斯图尔，15:1–15:27。https://doi.org/10.4230/LIPIcs.ECOOP.2016.15 Yuanbo Li、Qirun Zhang 和 Thomas Reps。2020b。交错 Dyck 可达性的快速图简化。载于第41届 ACM SIGPLAN 编程语言设计与实现会议论文集（英国伦敦）（PLDI 2020）。计算机协会，美国纽约州纽约市，780–793。https://doi.org/10.1145/3385412.3386021 Yuanbo Li、Qirun Zhang 和 Thomas Reps。2022年。交错 Dyck 可达性的快速图简化。ACM 程序语言汇刊 44，2，文章11（2022年5月），28页。https://doi.org/10.1145/3492428 Jingbo Lu、Dongjie He 和 Jingling Xue。2021年。Eagle：基于 CFL 可达性的、保持精度的对象敏感指针分析加速方法，采用部分上下文敏感性。ACM 软件工程方法学汇刊 30，4，文章46（2021年7月），46页。https://doi.org/10.1145/3450492 Jingbo Lu 和 Jingling Xue。2019年。采用部分上下文敏感性的保持精度且快速的对象敏感指针分析。ACM程序设计语言会议录 3，OOPSLA，文章148（2019年10月），29页。https://doi.org/10.1145/3360574

Wenjie Ma、Shengyuan Yang、Tian Tan、Xiaoxing Ma、Chang Xu 和 Yue Li。2023年。无上下文的上下文敏感性：一种用于快速且精确指针分析的 Cut-Shortcut 方法（制品）。https://doi.org/10.5281/zenodo.7808384 Ana Milanova、Atanas Rountev 和 Barbara G. Ryder。2002年。面向 Java 指向分析和副作用分析的参数化对象敏感性。载于《2002年 ACM SIGSOFT 国际软件测试与分析研讨会论文集》（意大利罗马）（ISSTA '02）。计算机协会，美国纽约州纽约市，1–11。https://doi.org/10.1145/566172.566174 Ana Milanova、Atanas Rountev 和 Barbara G. Ryder。2005年。面向 Java 指向分析的参数化对象敏感性。《ACM 软件工程与方法学汇刊》14，1（2005年1月），1–41。https://doi.org/10.1145/1044834.1044835 Mayur Naik、Alex Aiken 和 John Whaley。2006年。Java 的有效静态竞态检测。载于《第34届ACM SIGPLAN编程语言设计与实现会议论文集》（加拿大安大略省渥太华）（PLDI '06）。计算机协会，美国纽约州纽约市，308–319。https://doi.org/10.1145/1133981.1134018 Hakjoo Oh、Wonchan Lee、Kihong Heo、Hongseok Yang 和 Kwangkeun Yi。2014年。由影响预分析引导的选择性上下文敏感性。载于《第34届ACM SIGPLAN编程语言设计与实现会议论文集》（英国爱丁堡）（PLDI '14）。计算机协会，美国纽约州纽约市，475–484。https://doi.org/10.1145/2594291.2594318 Hakjoo Oh、Wonchan Lee、Kihong Heo、Hongseok Yang 和 Kwangkeun Yi。2015年。由影响预分析引导的选择性 X 敏感分析。《ACM 程序语言汇刊》38，2，文章6（2015年12月），45页。https://doi.org/10.1145/2821504 Michael Pradel、Ciera Jaspan、Jonathan Aldrich 和 Thomas R. Gross。2012年。使用挖掘得到的多对象规范静态检查 API 协议一致性。载于《第34届国际软件工程会议》，ICSE 2012，2012年6月，苏黎世，瑞士，Martin Glinz、Gail C. Murphy 和 Mauro Pezzè（编）。IEEE 计算机学会，–。https://doi.org/10.1109/ICSE.2012.6227127 Thomas Reps。1998年。通过图可达性进行程序分析1本文的简略版本作为受邀论文发表于《1997年国际逻辑编程研讨会论文集》[84].1。《信息与软件技术》40，11（1998年），701–726。https://doi.org/10.1016/S0950-5849(98)00093-7 Xavier Rival 和 Laurent Mauborgne。2007年。轨迹划分抽象域。《ACM 程序语言汇刊》29，5（2007年），26。https://doi.org/10.1145/1275497.1275501 Micha Sharir 和 Amir Pnueli。1981年。过程间数据流分析的两种方法。Prentice-Hall，第7章，189–234。Olin Grigsby Shivers。1991年。高阶语言的控制流分析，或驯服 lambda。卡内基梅隆大学。Yannis Smaragdakis 和 George Balatsouras。2015年。指针分析。《编程语言基础与趋势》2，1（2015年），1–69。https://doi.org/10.1561/2500000014 Yannis Smaragdakis、Martin Bravenboer 和 Ondrej Lhoták。2011年。充分混合上下文：理解对象敏感性。载于《第38届ACM SIGPLAN-SIGACT编程语言原理研讨会论文集》（美国得克萨斯州奥斯汀）（POPL '11）。计算机协会，美国纽约州纽约市，17–30。https://doi.org/10.1145/1926385.1926390 Yannis Smaragdakis、George Kastrinis 和 George Balatsouras。2014年。内省分析：全面的上下文敏感性。载于《第34届ACM SIGPLAN编程语言设计与实现会议论文集》（英国爱丁堡）（PLDI '14）。计算机协会，美国纽约州纽约市，485–495。https://doi.org/10.1145/2594291.2594320 Johannes Späth、Lisa Nguyen Quang Do、Karim Ali 和 Eric Bodden。2016年。Boomerang：面向 Java 的需求驱动流敏感与上下文敏感指针分析。载于《第30届欧洲面向对象编程会议》，ECOOP 2016，2016年7月18日至22日，意大利罗马。22:1–22:26。https://doi.org/10.4230/LIPIcs.ECOOP.2016.22 Manu Sridharan 和 Rastislav Bodík。2006年。面向 Java 的基于细化的上下文敏感指向分析。载于《ACM SIGPLAN 2006 编程语言设计与实现会议论文集》，加拿大安大略省渥太华，2006年6月11日至14日，Michael I. Schwartzbach 和 Thomas Ball（编）。ACM，387–400。https://doi.org/10.1145/1133981.1134027 Manu Sridharan、Satish Chandra、Julian Dolby、Stephen J. Fink 和 Eran Yahav。2013年。面向对象程序的别名分析。载于《面向对象编程中的别名：类型、分析与验证》，Dave Clarke、James Noble 和 Tobias Wrigstad（编）。《计算机科学讲义》第7850卷。Springer，196–232。https://doi.org/10.1007/978-3-642-36946-9 8_Manu Sridharan、Stephen J. Fink 和 Rastislav Bodik。2007年。细切片。载于《第28届ACM SIGPLAN编程语言设计与实现会议论文集》（美国加利福尼亚州圣迭戈）（PLDI '07）。计算机协会，美国纽约州纽约市，112–122。https://doi.org/10.1145/1250734.1250748 Tai-e。2022年。Tai-e：面向开发者的 Java 静态分析框架。https://github.com/pascal-lab/Tai-e。Tian Tan 和 Yue Li。2022年。Tai-e：通过吸收经典方法的最佳设计构建 Java 静态分析框架。arXiv:2208.00337 [cs.PL]Tian Tan、Yue Li、Xiaoxing Ma、Chang Xu 和 Yannis Smaragdakis。2021年。释放选择性上下文敏感性的力量，使指针分析更加精确。《ACM程序设计语言会议录》5，OOPSLA，文章147（2021年10月），27页。https://doi.org/10.1145/3485524

Tian Tan、Yue Li 和 Jingling Xue。2016年。在仍然保持 k 限制的情况下提高 k-对象敏感指针分析的精度。载于《静态分析——第23届国际研讨会》，SAS 2016，英国爱丁堡，2016年9月8日至10日，论文集（《计算机科学讲义》，第9837卷），Xavier Rival（编）。Springer，489–510。https://doi.org/10.1007/978-3-662-53413-7_24 Tian Tan、Yue Li 和 Jingling Xue。2017年。高效且精确的指向分析：通过合并等价自动机对堆建模。载于《第38届ACM SIGPLAN编程语言设计与实现会议论文集》，PLDI 2017，西班牙巴塞罗那，2017年6月18日至23日，Albert Cohen 和 Martin T. Vechev（编）。ACM，278–291。https://doi.org/10.1145/3062341.3062360 Manas Thakur 和 V. Krishna Nandivada。2019年。少比较、迟延更多：扩展基于值上下文的全程序堆分析。载于《第28届国际编译器构造会议论文集》（美国华盛顿特区）（CC 2019）。计算机协会，美国纽约州纽约市，135–146。https://doi.org/10.1145/3302516.3307359 Manas Thakur 和 V. Krishna Nandivada。2020年。充分混合上下文：上下文敏感性扩展领域近期进展所释放的机遇。载于《第29届国际编译器构造会议论文集》（美国加利福尼亚州圣迭戈）（CC 2020）。计算机协会，美国纽约州纽约市，27–38。https://doi.org/10.1145/3377555.3377902 Paolo Tonella 和 Alessandra Potrich。2005 年。面向对象代码的逆向工程。Springer。https://doi.org/10.1007/b102522 WALA。2006年。Watson分析库。http://wala.sf.net。Shiyi Wei 和 Barbara G. Ryder。2015年。JavaScript 的自适应上下文敏感分析。载于《第29届欧洲面向对象编程会议》，ECOOP 2015，2015年7月5日至10日，捷克共和国布拉格（《LIPIcs》，第37卷），John Tang Boyland（编）。Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik，712–734。https://doi.org/10.4230/LIPIcs.ECOOP.2015.712 Qirun Zhang、Michael R. Lyu、Hao Yuan 和 Zhendong Su。2013年。用于 Dyck-CFL可达性的快速算法及其在别名分析中的应用。载于《第34届ACM SIGPLAN编程语言设计与实现会议论文集》（美国华盛顿州西雅图）（PLDI '13）。计算机协会，美国纽约州纽约市，435–446。https://doi.org/10.1145/2491956.2462159 Xin Zhang、Ravi Mangal、Mayur Naik 和 Hongseok Yang。2014年。混合式自顶向下和自底向上过程间分析。载于《ACM SIGPLAN编程语言设计与实现会议》，PLDI '14，英国爱丁堡，2014年6月9日至11日，MichaelF. P. O'Boyle 和 Keshav Pingali（编）。ACM，249–258。https://doi.org/10.1145/2594291.2594328

收稿于2022年11月10日；接收于2023年3月31日