计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202220901
60(2): 274−293, 2023

Java 指针分析综述

谭 添[1,2] 马晓星[1,2] 许 畅[1,2] 马春燕[3] 李 樾[1,2]

[1](南京大学计算机科学与技术系 南京 210023)

[2](计算机软件新技术国家重点实验室(南京大学) 南京 210023)

[3](西北工业大学软件学院 西安 710129)

(tiantan@nju.edu.cn)

Survey on Java Pointer Analysis

Tan Tian[1,2], Ma Xiaoxing[1,2], Xu Chang[1,2], Ma Chunyan[3], and Li Yue[1,2]

[1] (*Department of Computer Science and Technology*, *Nanjing University*, *Nanjing* 210023)

[2] (*National Key Laboratory for Novel Software Technology* (*Nanjing University*), *Nanjing* 210023)

[3] (*School of Software*, *Northwestern Polytechnical University*, *Xi'an* 710129)

Abstract In recent years, static program analysis has become one of the key techniques to ensure the reliability, security and efficiency of software. As a fundamental program analysis technique, pointer analysis provides a series of fundamental information about the program for static program analysis, such as the points-to relations of any variables in the program, alias relations between variables, program call graph, and the reachability of heap objects. We introduce the important contents of Java pointer analysis, including pointer analysis algorithm, context sensitivity, abstraction of heap objects, handling of complex language features, non-whole program pointer analysis, especially we sort-out and discuss selective context sensitivity, which is the research hotspot of pointer analysis in recent years.

Key words pointer analysis; alias analysis; Java; static analysis; context sensitivity

摘 要 近年来静态程序分析已成为保障软件可靠性、安全性和高效性的关键技术之一.指针分析作为基础程序分析技术为静态程序分析提供关于程序的一系列基础信息,例如程序任意变量的指向关系、变量间的别名关系、程序调用图、堆对象的可达性等.介绍了 Java 指针分析的重要内容:指针分析算法、上下文敏感、堆对象抽象、复杂语言特性处理、非全程序指针分析,特别是对近年来指针分析的研究热点选择性上下文敏感技术进行了梳理和讨论.

关键词 指针分析;别名分析;Java;静态分析;上下文敏感

中图法分类号 TP311

指针分析(pointer analysis),又称指向分析(points-to analysis)或别名分析(alias analysis),是计算程序中的指针(或变量、引用)在运行时所能指向的内存位置(或对象)的一种静态程序分析技术.指针分析的结果通常可表示为指针与内存位置之间的指向关系(points-to relation)或每个指针的指针集(points-to set).指针分析提供了程序中基础的数据流信息,对于一系列技术如编译优化[1-3]、故障检测[4-6]、安全分析[7-11]、程序理解[12-13]、程序验证[14-15]等具有重要作用.作为公认的最基础的静态分析技术之一[1,16],指针分析这

收稿日期:2022−10−26;**修回日期**:2023−01−11

基金项目:国家自然科学基金项目(61932021, 62025202, 62002157);航空科学基金项目(20185853038, 201907053004)

This work was supported by the National Natural Science Foundation of China (61932021, 62025202, 62002157), and the Aeronautical Science Fund (20185853038, 201907053004).

通信作者:李樾(yueli@nju.edu.cn)

一研究领域已有超过 40 年的历史[17]，至今仍是静态程序分析学术研究的重点.

指针分析通过对程序中语句语义的分析来计算指针的指向信息，因此指针分析与被分析语言的语义紧密相关，导致针对不同程序设计语言的指针分析技术呈现出较大的差异性. 目前，主流指针分析的研究主要有两大流派，针对 Java 语言[1,3,18-20] 与 C/C++语言[21-25] 的指针分析研究. Java 作为一门典型的面向对象语言，其程序中绝大部分数据分配在堆上；而 C/C++程序中许多数据分配在栈上，通常栈上的数据仅限于其声明的函数内部使用. 而相比之下，堆上的数据可以跨函数存在，一般具有更大的活动范围和生存周期，因此更难以分析. 此外，Java 具有反射、本地代码调用等 C/C++所没有的语言特性，会给指针分析造成很大挑战. Java 语言具有广泛的应用生态，近十年针对 Java 指针分析的研究工作也多于针对 C/C++的. 因此，本文关注 Java 指针分析技术.

衡量指针分析有效性有 3 个关键指标：效率（efficiency）、精度（precision）和可靠性（soundness）. 快速的指针分析运行时间较短，可以更容易部署到实际生产当中，为相关的应用提供支撑；精准的指针分析，则可以提升相关应用的准确度（例如精度更高的指针分析，能使得程序代码在编译时有更多机会被优化，或使得以它为基础的错误检测工具具备更低的误报率）；可靠性更好的指针分析能覆盖更多程序行为（例如可使以它为基础的安全漏洞检测工具查出更多的安全问题）. 因此，有效提升效率、精度和可靠性一直以来是指针分析领域的主要研究问题. 本文将介绍指针分析提升这 3 个关键指标的经典技术以及近年来的主要研究进展.

具体而言，相比于已有 Java 指针分析及其综述工作[1,3]（最近的一篇综述工作发表于 2015 年），本文的主要贡献包括 3 个方面：

1）描述了更完整且简洁易懂的 Java 指针分析算法；

2）讨论了更多关于 Java 指针分析重要内容的研究工作（例如关于堆抽象、新语言特性处理、增量分析等方面的最新工作）；

3）系统性地梳理并讨论了近年来 Java 指针分析的研究热点——选择性上下文敏感技术.

本节接下来简要介绍本文后续章节讨论的指针分析技术所针对的问题，方便读者理解这些研究的关联性，并了解本文结构.

第 1 节介绍 Java 指针分析的基础规则和算法. 由于 Java 程序的规模性以及 Java 语言的复杂性，基础的算法并不足以在效率、精度和可靠性方面满足对现实世界 Java 程序的分析需求，因此需要利用在第 2~4 节介绍的进阶技术对程序行为进行更有效的抽象与模拟.

Java 程序中的一个方法在运行时可能被多次调用，每次被调用时都处于不同的调用上下文（calling context）中，并可能具有不同的指向关系. 第 2 节介绍上下文敏感（context sensitivity）技术. 该技术本质上是对程序运行时调用栈（call stack）的抽象，通过对调用栈中上下文进行细化地建模，以减少指向关系混淆，是提升 Java 指针分析精度最主要的技术.

Java 作为典型的面向对象语言，运行时会在堆上创建大量对象，因此研究人员引入堆对象抽象技术，在指针分析中对动态运行时的对象进行合理的抽象以保证指针分析的有效性. 第 3 节介绍堆对象抽象的主要技术.

第 1~3 节介绍的技术主要针对 Java 基础特性，但 Java 作为一门工业级语言，其具有复杂的语言特性，其中一些关键特性（如反射）对指针分析的结果，尤其是对可靠性会产生较大影响. 第 4 节介绍现有技术如何在指针分析中处理这些关键的复杂语言特性.

第 1~4 节介绍的指针分析技术主要针对全程序指针分析，全程序指针分析一般开销相对较大（尤其是对于复杂程序），但用户通常希望能够尽快获得指针分析结果. 研究人员发现在一些特定的实际应用场景中，并不需要分析整个程序，因此提出非全程序指针分析技术，在只分析部分程序的前提下也能获得满足用户需求的结果. 第 5 节介绍这一类技术的代表性工作.

1 指针分析算法

为了便于介绍 Java 指针分析技术和详细理解指针分析，本文设计了一套较为完整且易于理解的 Java 指针分析算法. 本节先介绍该算法分析的 Java 中的指针和语句，然后详细介绍算法本身.

1.1 Java 中的指针

在指针分析中，我们主要关注引用类型（reference type）指针. 由于原子类型（primitive type）变量与字段不能指向堆上的对象，因此它们通常不在指针分析所考虑的范围内. Java 指针可分为 4 类：

1）局部变量，如 x，即声明在方法内部的变量. 这也是程序中数量最多的一种指针.

2）静态字段，如 $C.f$，静态字段的处理方式与局

部变量类似(文献[3]将其称为全局变量(global variable)). 为了简化算法, 本文接下来忽略静态字段及其相关语句的处理.

3) 实例字段, 如 *x.f*, Java 程序中可以写出复杂的实例字段访问表达式, 如 *x.f.g.h*, 但这种复杂的表达式不易于分析, 因此通常在分析前先引入临时变量将程序转换为三地址码(如语句 $v=x.f.g$ 会被转换为 $t=x.f$ 和 $v=t.g$;)再进行分析.

4) 数组元素, 如 *a*[*i*]. 由于许多情况下静态分析无法获取准确的数组长度以及索引值, 因此指针分析通常会忽略数组长度, 且不区分具体的索引值, 并将数组对象建模为只有一个特殊实例字段 *arr* 的对象, 且该字段指向所有被存入数组的元素, 如表 1 的例子所示.

Table 1 Comparison of Real Code and Array Modeling of Pointer Analysis

表 1 真实代码与指针分析对数组建模的对比

真实代码	指针分析数组建模
array = new String[10];	*array* = new String[];
array[0] = "x";	*array*.arr = "x";
array[1] = "y";	*array*.arr = "y";
s = *array*[0];	*s* = *array*.arr;

指针分析对数组进行特殊的建模后, 对数组元素的处理与实例对象一致, 因此本文在算法中不再讨论对数组元素及其相关语句的处理.

综上所述, 为了简化算法, 本文只考虑局部变量与实例字段.

1.2 影响指针的 Java 语句

Java 有许多种语句, 但在指针分析中, 只需要关注直接影响指针的语句. 当只考虑局部变量与实例字段时, 影响指针分析的语句有 5 种:

1) 对象创建, 如 $x=\text{new } T()$;

2) 复制, 如 $x=y$;

3) 字段存储, 如 $x.f=y$;

4) 字段读取, 如 $y=x.f$;

5) 方法调用, 如 $r=x.m(a,\cdots)$.

这 5 种语句最复杂的是方法调用语句. Java 有 3 种方法调用: 静态调用(static invocation)、特殊调用(special invocation)和虚调用(virtual invocation). 其中虚调用的处理最为复杂, 而静态调用与特殊调用的处理逻辑均可视为虚调用处理逻辑的简化. 因此本文关注虚调用的处理.

1.3 算 法

指针分析算法的输入是一个 Java 程序, 输出是程序中每个指针可能指向的对象集合, 算法运行过程中在线地解析方法调用, 因此运行结束后也输出程序的调用图. 为了便于理解算法, 本文在表 2 中列出了算法中所用到的符号以及相关域.

Table 2 Notations and Domains Used in Pointer Analysis Algorithm

表 2 指针分析算法符号与域的说明

	符号	域
方法	m	M
变量	x, y	V
对象	o_i, o_j	O
字段	f, g	F
实例字段	$o_i.f, o_j.g$	$O\times F$
指针	s, t	$Pointer=V\cup(O\times F)$
指向关系	pt	$Pointer\rightarrow\mathcal{P}(O)$

Pointer 表示程序中指针的集合. 本文关注变量与实例字段, 因此 *Pointer* 由变量集合(V)与实例字段集合($O\times F$, 即对象集合 O 与字段集合 F 的笛卡尔乘积)组成. 本文用指向关系 pt 表示指针分析的结果, $pt(p)$表示指针 p 指向的对象集合, 即 p 的指针集(points-to set). 此处$\mathcal{P}(O)$表示对象集合 O 的幂集.

本节介绍的指针分析是一种 Andersen 风格的指针分析[21], 即它是流不敏感(flow-insensitive)且基于子集约束(subset constraint)的指针分析. 流不敏感意味着指针分析不考虑程序语句的执行顺序[26], 并将程序语句视为一个无序集合[1]. 子集约束指程序中的指针发生赋值时, 指针分析将建立相应指针间的子集约束关系, 例如对于语句 $x=y$, 分析将建立约束 $pt(y)\subseteq pt(x)$并保证分析结果满足约束[3,21]. 此外, 本节介绍的是上下文非敏感(context-insensitive)指针分析, 即不区分每个方法在不同调用上下文(calling context)中指向信息的差别. 本文在第 3 节介绍如何将本节的算法扩展成上下文敏感(context- sensitive)指针分析从而提升精度.

1)指针分析规则. 表 3 给出了指针分析对 1.2 节所述的影响指针语句的处理规则(下文将在描述算法时介绍表 3 中的列 4“PFG 边”, 此处读者只需关注左边 3 列). 这套规则描述了标准的 Java 指针分析规则, 反映不同语句如何影响相应指针的指针集, 同时也描述了指针之间子集约束的建立, 本质上与文献[1,3]中所描述的指针分析等价.

表 3 中的规则大部分较为直观, 对于对象创建语句, 本文定义 *Heap*(i)函数, 将程序中的对象创建点

Table 3 Rules for Pointer Analysis

表 3 指针分析规则

语句种类	语句	指针分析规则	PFG 边
对象创建	i: x = new T()	$\dfrac{o_i = Heap(i)}{o_i \in pt(x)}$	
复制	$x = y$	$\dfrac{o_i \in pt(y)}{o_i \in pt(x)}$	$x \leftarrow y$
字段存储	$x.f = y$	$\dfrac{o_i \in pt(x), o_j \in pt(y)}{o_j \in pt(o_i.f)}$	$o_i.f \leftarrow y$
字段读取	$y = x.f$	$\dfrac{o_i \in pt(x), o_j \in pt(o_i.f)}{o_j \in pt(y)}$	$y \leftarrow o_i.f$
方法调用	l: $r = x.k(a_1,\cdots,a_n)$	$\dfrac{\begin{array}{c}o_i \in pt(x), m = Dispatch(o_i, k)\\ o_u \in pt(a_j), 1 \leqslant j \leqslant n\\ o_v \in pt(m_{\text{ret}})\end{array}}{\begin{array}{c}o_i \in pt(m_{this})\\ o_u \in pt(m_{p_j}), 1 \leqslant j \leqslant n\\ o_v \in pt(r)\end{array}}$	$\begin{array}{c}a_1 \rightarrow m_{p_1}\\ \vdots\\ a_n \rightarrow m_{p_n}\\ r \leftarrow m_{\text{ret}}\end{array}$

(allocation site)i映射成对应的抽象对象(abstract object)o_i, $Heap()$函数的具体实现对应不同的堆抽象技术, 本文在第 3 节更详细地讨论了堆抽象. 对于方法调用语句的分析规则相对复杂, 本文定义辅助函数 $Dispatch(o_i, k)$, 根据对象 o_i 的类型以及调用点 k 的方法签名, 计算出调用点 l 以 o_i 作为接收者对象(receiver object)时具体的目标方法 m. 此过程遵循标准的类继承结构查找算法[1,27]. 得到 m 后, 分析规则描述如何在 l 与 m 的相关变量之间传递对象, 其中 m_{ret} 表示 m 中指向返回值的变量, m_{this} 表示 m 的 this 变量, m_{pj} 表示 m 的第 j 个形参.

2)指针分析算法. 本文设计的指针分析算法是表 3 中指针分析规则的命令式风格实现, 而实现指针分析算法的关键在于如何满足各相关指针间的子集约束关系. 本文使用指针流图(pointer flow graph, PFG)来表示指针间的子集约束关系, 具体地说, PFG 的每一个结点都对应程序中的一个指针, 若指针 s 与 t 存在约束关系 $pt(s) \subseteq pt(t)$, 则算法在 PFG 上添加边 $s \rightarrow t$, 并在分析过程中沿着 $s \rightarrow t$ 传播指向关系, 确保 s 指向的所有对象都被包含于 $pt(t)$中. 本质上, PFG 类似于 Andersen 风格指针分析技术[21]中的约束图(constraint graph)[28]. 表 3 给出指针分析规则, 其中在列 4 给出了添加 PFG 边的规则. 而指针分析算法的核心思路就是根据程序中的语句和表 3 的规则构造 PFG, 并在 PFG 上传播指向关系, 直至算法到达不动点.

算法 1 给出了本文设计的指针分析算法. 该算法遵循上述思路, 构建 PFG 并在此基础上传播和计算程序中各指针的指针集. 此外, 算法 1 是一个全程序分析, 在指针分析的过程中同步构造出程序的调用图.

算法 1. 指针分析算法.

输入: 程序入口方法 m^{entry};

输出: 指向关系 pt, 调用图 CG.

① $Solve(m^{\text{entry}})$
② WL=[ ], PFG={}, S={}, RM={}, CG={};
③ $AddReachable(m^{\text{entry}})$;
④ while WL is not empty do
⑤ remove $\langle n, pts\rangle$ from WL;
⑥ $\Delta = pts - pt(n)$;
⑦ $Propagate(n, \Delta)$;
⑧ if n represents a variable x then
⑨ foreach $o_i \in \Delta$ do
⑩ foreach $x.f = y \in S$ do
⑪ $AddEdge(y, o_i.f)$;
⑫ foreach $y = x.f \in S$ do
⑬ $AddEdge(o_i.f, y)$;
⑭ $ProcessCall(x, o_i)$;

⑮ $AddReachable(m)$
⑯ if $m \notin RM$ then
⑰ add m to RM;
⑱ $S \cup = S_m$;
⑲ foreach i: x = new $T(\,) \in S_m$ do
⑳ $o_i = Heap(i)$;
㉑ add $\langle x, \{o_i\}\rangle$ to WL;
㉒ foreach i: $x = y \in S_m$ do
㉓ $AddEdge(y, x)$;

㉔ $Propagate(n, pts)$
㉕ if pts is not empty then
㉖ $pt(n) \cup = pts$;

```
㉗      foreach n → s ∈ PFG do
㉘          add ⟨s, pts⟩ to WL;

㉙ AddEdge(s, t)
㉚  if s → t ∉ PFG then
㉛      add s → t to PFG;
㉜      if pt(s) is not empty then
㉝          add ⟨t, pt(s)⟩ to WL;

㉞ ProcessCall(x, o_i)
㉟  foreach l: r = x.k(a_1, ⋯, a_n) ∈ S do
㊱      m = Dispatch(o_i, k);
㊲      add ⟨m_this, {o_i}⟩ to WL;
㊳      if l → m ∉ CG then
㊴          add l → m to CG;
㊵          AddReachable(m);
㊶          foreach parameter p_i of m do
㊷              AddEdge(a_i, p_i);
㊸          AddEdge(m_ret, r);
```

Solve(m^{entry})是整个算法的主函数,它的输入 m^{entry} 是程序的入口方法(对应 Java 程序的 main 方法). *Solve*()首先初始化 5 个贯穿算法始终的数据结构:

① *WL*, 表示算法的工作列表(work list). *WL* 中的元素是指针与对象集合组成的对(pair),元素表示待处理的指向关系. 例如 *WL* 中若存在元素⟨*n*, *pts*⟩,则表示算法发现集合 *pts* 中的对象应当加入到指针集 *pt*(*n*)中. 注意,当算法发现新的指向关系时,并不是直接更新 *pt*,而是将信息加入 *WL* 等待后续处理,这是因为对于新指向关系的处理并不只是更新 *pt*,还包括沿着 PFG 传播新信息以及对以字段存储/读取、方法调用等相关语句的一系列处理,因此算法将所有新的指向信息加入 *WL*,便于统一操作.

② *PFG*, 表示指针流图 PFG 边的集合.

③ *S*, 表示被程序的可达语句集合.

④ *RM*, 表示程序的可达方法(reachable method)集合.

⑤ *CG*, 表示程序的调用边(call edge)集合,每条调用边的源点是程序中的一个调用点(call site),终点是调用的目标方法.

Solve()调用 *AddReachable*()函数并传入入口方法 m^{entry}. 当算法发现新的可达方法时,会调用 *AddReachable*()处理新方法内的语句. 对于参数 *m*, *AddReachable*()首先检查 *m* 是否已经在可达方法集合 *RM* 中,如果是,说明 *m* 已经处理过,可以直接返回从而避免冗余计算. 对于此前未遇到过的方法 *m*, *AddReachable*()将其加入 *RM*,并将 *m* 中的语句(S_m 表示方法 *m* 中的语句集合)加入可达语句集合 *S*. 随后处理 *m* 的对象创建以及复制语句.

对于对象创建语句,算法使用 *Heap*(*i*)函数,将程序中的对象创建点(allocation site)*i* 映射成对应的抽象对象(abstract object)o_i,得到 o_i 后,算法将其作为新发现的变量 *x* 的指向关系加入 *WL* 中.

对于复制语句的处理很简单,只需调用 *AddEdge*()函数添加对应的 PFG 边即可. 此处的 *AddEdge*(*s*, *t*)函数用于向集合 *PFG* 中添加 $s \to t$ 的边. 由于算法添加边 $s \to t$ 时,*pt*(*s*)可能已经指向了一些对象,为了确保 PFG 边表示的约束成立,即 *pt*(*t*)指向 *pt*(*s*)中所有对象,算法在添加 $s \to t$ 时,也将⟨*t*, *pt*(*s*)⟩加入工作列表 *WL* 中. 边 $s \to t$ 添加之后,*s* 指向的新对象将由 *Propagate*()函数传播给 *t*.

分析完入口方法之后,*Solve*()进入主循环,反复地取出 *WL* 中的元素进行处理,处理的过程中也可能发现新的指向关系并加入 *WL*,直至 *WL* 为空(不再有新的指向关系被发现). 对于从 *WL* 取出的元素⟨*n*, *pts*⟩,随着算法进程的推进,*pts* 很可能包含许多 *pt*(*n*)已经指向的对象,因此为了减少冗余传播与计算,提升算法效率,*Solve*()在处理⟨*n*, *pts*⟩时使用差异传播(difference propagation)技术[18,29],即在处理 *pts* 中的对象之前,先将其与 *pt*(*n*)做差集,取出 *pt*(*n*)此前未包含的对象并存入差集 Δ,而 Δ 中的对象才代表真正新的指向关系. 在 *Solve*()接下来的部分都对通常规模更小的 Δ(而非 *pts*)进行操作.

得到 Δ 之后,*Solve*()调用 *Propagate*(*n*, Δ)将 Δ 中的对象全部加入到指针 *n* 对应的指针集中(即 *pt*(*n*)),并传播至 *n* 在 PFG 中的所有后继.

若 *n* 表示一个变量 *x* 对应的指针,则 *Solve*()需要处理与 *x* 相关的字段存储/读取,以及方法调用语句. 本文基于指针流图 PFG 的设计使得算法对于字段存储/读取语句的处理相当简单. 以字段存储语句为例,若可达语句集合 *S* 中存在语句 *x.f* = *y*,且算法发现 *x* 指向对象 o_i(o_i 属于新发现的对象集合 Δ),则算法只需调用 *AddEdge*(*y*, $o_i.f$)在 PFG 上添加一条变量 *y* 到实例字段 $o_i.f$ 的边,使得 $o_i.f$ 的指针集包含 *y* 所指向的对象. 而对于字段读取语句的处理与字段存储对称,也只需调用 *AddEdge*()即可. 关于方法调用的处理相对复杂,算法用一个辅助函数 *ProcessCall*()实现相关逻辑.

ProcessCall(*x*, o_i)处理与变量 *x* 以及其指向的对象 o_i 相关的调用,当算法发现变量 *x* 指向新的对象

o_i 时，*ProcessCall*( )枚举变量 x 指向接收者对象的调用点(形如 $x.k(\cdots)$)，并对每个调用点进行处理：

1)解析目标方法. 本文定义辅助函数 *Dispatch*(o_i, k)，根据对象 o_i 的类型以及调用点 k 的方法签名计算出以 o_i 作为接收者对象的具体目标方法 m.

2)传递接收者对象. 将接收者对象 o_i 传入目标方法的 this 变量中(m_{this} 表示方法 m 的 this 变量).

3)处理调用边. 当 *ProcessCall*( )在第 1 步通过 *Dispatch*( )得到调用目标方法 m 时，实际上也分析出了程序的一条调用边 $l \to m$(l 表示调用点的标号). *ProcessCall*( )首先检查 $l \to m$ 是否已在调用边集合 *CG* 中，如果是，说明 $l \to m$ 已经处理过，可以直接返回从而避免冗余计算；若 $l \to m$ 是新发现的调用边，则将其加入 *CG*，并且 m 有可能是新发现的可达方法，因此调用 *AddReachable*(m)对其进行处理. 然后，算法使用 *AddEdge*( )将调用点的实参传入目标方法的形参(本文用 a_i 表示调用点 l 的第 i 个实参，p_i 表示方法 m 的第 i 个形参)，并将目标方法的返回值传回调用语句的左值(本文用 m_{ret} 表示方法 m 中指向返回值的变量).

算法结束后，可得到程序中各变量、实例字段的指针集(存放于 *pt*)以及程序的调用图(存放于 *CG*).

2 上下文敏感

Java 程序中的一个方法在运行时可能被多次调用，每次被调用时都处于不同的调用上下文(calling context)中，上下文敏感(context sensitivity)技术[30]就是研究如何在静态分析(如指针分析)中对动态运行时的上下文进行建模和分析. 上下文敏感可以区分同一方法在不同上下文中的数据流，从而减少数据流的混淆并提升精度. 图 1 用一个例子解释上下文敏感的思路及其作用.

在图 1 的代码片段中，方法 *identity*( )分别被 *foo*( )和 *bar*( )调用，*foo*( )调用 *identity*( )时传给其字符串“foo”作为参数，然后由变量 r_1 接收其返回值，显而易见，运行时变量 r_1 指向字符串“foo”. *bar*( )与之类似，运行时 r_2 将指向字符串“bar”. 然而，若使用 1.3 节介绍的上下文非敏感指针分析算法分析这段程序，则 *identity*( )中的变量 s 会指向{“foo”, “bar”}，且这 2 个字符串会传播给 r_1 与 r_2，使得 r_1 与 r_2 都指向{“foo”, “bar”}，导致指针分析结果不精确(实际运行时，r_1 或 r_2 不指向“bar”或“foo”).

造成图 1 的例子中精度丢失的原因在于方法

```
1  void foo( ) {
2     String s1 = “foo”;
3     String r1 = identity(s1);
4  }
5  void bar( ) {
6     String s2 = “bar”;
7     String r2 = identity(s2 );
8  }
9 String identity(String s) {
10    return s;
11 }
```

Fig. 1 An example for illustrating context sensitivity

图 1 解释上下文敏感的例子

identity( )在运行时有 2 个调用上下文(来自 *foo*( )与 *bar*( ))，并且 *identity*( )内的变量 s 在 2 个上下文中指向不同对象(分别为“foo”与“bar”). 然而上下文非敏感分析并不区分这 2 个上下文，因此 2 个上下文中的数据流在 *identity*( )内部混淆，并且传递给了 r_1 与 r_2. 而上下文敏感分析的思路是将 *identity*( )的不同调用上下文加以区分并分别分析，从而避免数据流的混淆以提升精度.

目前，上下文敏感是提升 Java 指针分析精度公认最有效的方法[20,31-34]，多年来一直是该领域的研究重点. 2.1 节将 1.3 节介绍的指针分析算法扩展成上下文敏感指针分析算法，2.2 节与 2.3 节分别介绍传统上下文敏感技术以及近年来的相关研究热点，选择性上下文敏感技术.

2.1 上下文敏感指针分析算法

在上下文敏感指针分析中，程序中的每个方法会被冠以上下文，用 *c*:*m* 表示(*c* 表示某个上下文)，称为上下文敏感方法. 每个上下文可以被视为一个标识符，用于将该上下文中的方法(如 *c*:*m*)与其它上下文中的同一方法(如 *c'*:*m*)加以区分. 此外，通常每个方法中的变量与创建出的对象也继承该方法的上下文，成为上下文敏感变量与对象. 不同上下文中的变量以及对象的实例字段可指向不同对象，从而达到实现对不同上下文数据流的区分. 而具体上下文的生成取决于指针分析使用的上下文敏感技术，本文在 2.2~2.3 节进行介绍.

表 4 列出了上下文敏感指针分析算法用到的符号以及相关域. 表 4 与表 2 相比的区别在于，上下文敏感分析中程序的方法、变量、对象被冠以上下文. 相应地，指向关系 *pt* 相关的域，即指针集合(*CSPointer*)和对象集合中的元素也都带有上下文.

1)指针分析规则. 表 5 给出了上下文敏感指针分析的规则. 假设表中语句所在方法的当前上下文为 *c*,

Table 4 Notations and Domains Used in Context-Sensitive Pointer Analysis Algorithm

表 4 上下文敏感指针分析算法符号与域的说明

	符号	域
上下文	c, c', c''	C
上下文敏感方法	$c{:}m$	$C \times M$
上下文敏感变量	$c{:}x, c'{:}y$	$C \times V$
上下文敏感对象	$c{:}o_i, c'{:}o_j$	$C \times O$
字段	f, g	F
实例字段	$c{:}o_i.f, c'{:}o_j.g$	$C \times O \times F$
上下文敏感指针	s, t	$CSPointer = (C \times V) \cup (C \times O \times F)$
指向关系	pt	$CSPointer \to \mathcal{P}(C \times O)$

则相关语句的变量的上下文都是 c, 如复制语句的变量 x 和 y、字段存储语句的变量 x 和 y 等, 因为同一语句的变量必然声明在同一方法内, 因此它们具有相同的上下文. 表 5 给出的分析规则与表 3 的上下文非敏感指针分析在本质上都可视为 Andersen 风格指针分析(描述了程序各指针之间如何建立子集约束), 表 5 与表 3 的区别在于上下文敏感分析中的指针都带有上下文, 可以使不同上下文中的指向关系得以分开分析, 从而提升精度.

在上下文敏感指针分析中, 方法调用的规则最为重要和复杂, 因为它涉及到上下文的生成. 具体地说, 本文定义 *Select*$(c, l, c'{:}o_i)$函数, 根据调用点的信息(当前调用者上下文 c, 调用点标号 l, 接收者对象 $c'{:}o_i$)生成目标方法 m(与表 3 中上下文非敏感分析一样, 由 Dispatch 得到)的上下文 c^t. *Select*()函数的具体实现对应不同的上下文敏感技术, 本文将在 2.2~2.3 节展开讨论. 得到目标方法的上下文 c^t 后, 指针分析规则在 c 中 l 的变量与 m 在 c^t 中的变量之间互相传递对象. 由于 m 的 c^t 是根据调用点的信息生成的, 因此 m 也与当前调用建立了关联, 从而可以与 m 的其它上下文(例如由其它调用点发起的调用)区分开, 从而避免不同上下文数据流混淆造成的精度丢失.

2)指针分析算法. 本文设计的上下文敏感指针分析算法如算法 2 所示. 该算法是由算法 1(上下文非敏感指针分析算法)扩展而成, 算法的思路仍是构建指针流图 PFG 用于表达子集约束关系, 并沿着 PFG 传播指向关系直至到达不动点. 表 5 的列 4 给出了添加 PFG 边的规则. 在上下文敏感分析中, PFG 的结点都是带有上下文的指针. 由于算法 2 的流程与算法 1 一致, 因此本文不再赘述.

接下来介绍算法 2 中与上下文敏感相关的改动部分. 与算法 1 算法一样, 算法 2 中的 *Solve*()首先初始化 5 个关键数据结构. 这些数据结构起到的作用与算法 1 一致, 区别在于它们的元素(除了可达语句集合 S)都带有上下文, 例如可达方法集合 RM 中的元素是上下文敏感方法(如 $c{:}m$). 对于可达语句集合 S 中的元素(语句), 其上下文信息可从该语句的变量或语句所在的方法中获得, 因此 S 中的语句无需携带上下文. 然后, *Solve*()调用 *AddReachable*()处理入口方法 m^{entry}. 由于算法 2 做上下文敏感分析, 因此 *AddReachable*()的参数是带上下文的方法, 如 $c{:}m$, 这意味着同一个方法可能会在不同上下文中被 *AddReachable*()分析多次. 此处 m^{entry} 方法并没有调用者, 因此算法给予其空上下文"[ ]". 在 *AddReachable*($c{:}m$)内部处理对象创建以及复制语句时, 都会将方法的上下文 c 赋予相关的变量以及对象. 然后 *Solve*()进入主循环. 算法 2 使用的辅助函数 *AddEdge*()和 *Propagate*()与算法 1 所定义的完全一致, 因

Table 5 Rules for Context-Sensitive Pointer Analysis

表 5 上下文敏感指针分析规则

语句种类	语句	指针分析规则(在上下文c中)	PFG 边
对象创建	$i{:}\ x = new\ T()$	$\dfrac{o_i = Heap(i)}{c : o_i \in pt(c : x)}$	
复制	$x = y$	$\dfrac{c' : o_i \in pt(c : y)}{c' : o_i \in pt(c : x)}$	$c : x \leftarrow c : y$
字段存储	$x.f = y$	$\dfrac{c' : o_i \in pt(c : x), c'' : o_j \in pt(c : y)}{c'' : o_j \in pt(c' : o_i.f)}$	$c' : o_i.f \leftarrow c : y$
字段读取	$y = x.f$	$\dfrac{c' : o_i \in pt(c : x), c'' : o_j \in pt(c' : o_i.f)}{c'' : o_j \in pt(c : y)}$	$c : y \leftarrow c' : o_i.f$
方法调用	$l{:}\ r = x.k(a_1, \cdots, a_n)$	$\dfrac{\begin{array}{c} c' : o_i \in pt(c : x), \\ m = Dispatch(o_i, \mathrm{k}), c^t = Select(c, l, c' : o_i) \\ c'' : o_u \in pt(c : aj), 1 \leqslant j \leqslant n \\ c''' : o_v \in pt(c^t : m_{\text{ret}}) \end{array}}{\begin{array}{c} c' : o_i \in pt(c^t : m_{\text{this}}) \\ c'' : o_u \in pt\left(c^t : m_{\text{pj}}\right), 1 \leqslant j \leqslant n \\ c''' : o_v \in pt(c : r) \end{array}}$	$\begin{array}{c} c : a_1 \to c^t : m_{p_1} \\ \vdots \\ c : a_n \to c^t : m_{p_n} \\ c : r \leftarrow c^t : m_{\text{ret}} \end{array}$

此就不再重复给出. 在主循环的最后, 算法调用扩展后的 *ProcessCall*( )处理方法调用.

算法 2. 上下文敏感指针分析算法.

输入: 程序入口方法 m^{entry};

输出: 指向关系 *pt*, 调用图 *CG*.

① *Solve*(m^{entry})
② WL=[ ], PFG={}, S={}, RM={}, CG={};
③ *AddReachable*([ ]:m^{entry});
④ while WL is not empty do
⑤ remove $\langle n, pts\rangle$ from WL;
⑥ $\Delta = pts - pt(n)$;
⑦ *Propagate*(n, Δ);
⑧ if n represents a variable c:x then
⑨ foreach c':$o_i \in \Delta$ do
⑩ foreach $x.f = y \in S$ do
⑪ *AddEdge*(c:y, c':o_i.f);
⑫ foreach $y = x.f \in S$ do
⑬ *AddEdge*(c':o_i.f, c:y);
⑭ *ProcessCall*(c:x, c':o_i);

⑮ *AddReachable*(c:m)
⑯ if c:$m \notin RM$ then
⑰ add c:m to RM;
⑱ $S \cup = S_m$;
⑲ foreach i: x = new T( ) $\in S_m$ do
⑳ o_i = Heap(i);
㉑ add $\langle c$:x, {c:o_i}$\rangle$ to WL;
㉒ foreach i: $x = y \in S_m$ do
㉓ *AddEdge*(c:y, c:x);

㉔ *ProcessCall*(c:x, c':o_i)
㉕ foreach l: $r = x.k(a_1, \cdots, a_n) \in S$ do
㉖ m = *Dispatch*(o_i, k);
㉗ c^t = *Select*(c, l, c':o_i);
㉘ add $\langle c^t$:m_{this}, {c':o_i}$\rangle$ to WL;
㉙ if c:$l \rightarrow c^t$:$m \notin CG$ then
㉚ add c:$l \rightarrow c^t$:m to CG;
㉛ *AddReachable*(c^t:m);
㉜ foreach parameter p_i of m do
㉝ *AddEdge*(c:a_i, c^t:p_i);
㉞ *AddEdge*(c^t:m_{ret}, c:r);

ProcessCall(c:x, c':o_i)总体逻辑与算法 1 中的 *ProcessCall*(x, o_i)相似, 但多了一个关键步骤: 上下文的生成. 如上文所述, 本文定义 *Select*(c, l, c':o_i)函数, 根据调用点的信息, 选择目标方法的上下文. 除了入口方法 m^{entry}, 其余所有方法的上下文都在此处生成. 生成上下文 c^t 后, *ProcessCall*( )将其赋予目标方法 m, 添加相关调用边, 并根据表 5 所列的规则在调用点和目标方法之间传递参数以及返回值.

算法 2 结束后, 可得到上下文敏感的指针分析结果(pt)以及上下文敏感的程序调用图(CG).

关于算法 2 的更多实现细节, 可参考 Tai-e[35-36]. Tai-e 是最新的通用型 Java 程序分析框架(与 Soot、WALA 类似). 本文将上述上下文敏感指针分析算法作为核心算法实现在 Tai-e 的指针分析系统中. 实验结果表明[35], Tai-e 指针分析系统的效率与可靠性均优于已有指针分析工具[37-39]. 此外, 本文为 Tai-e 指针分析系统设计了一套插件机制, 使其具有良好的扩展性, 使得开发者能够方便地开发需要与指针分析交互的分析技术(如反射分析、异常分析、污点分析等), 关于指针分析插件机制的设计可以参考文献 [35].

2.2 传统上下文敏感

本节介绍几种 Java 指针分析最常用的传统上下文敏感技术, 并将给出这些技术对应的 *Select*(c, l, c':o_i)函数(见表 5 与算法 2)的具体定义. 这些传统上下文敏感技术也是 2.3 节介绍的选择性上下文敏感的基础.

1)调用点敏感. 调用点敏感(call-site sensitivity)[30,40-41], 又称调用串敏感(call-string sensitivity)或 k-CFA, 是最早诞生的上下文敏感技术. 调用点敏感的每个上下文由一串调用点组成(具体实现中, 通常会给程序中的每个调用点赋予一个唯一的标号, 并用标号表示相应的调用点), 当分析调用点 l 时, 调用点敏感将 l 所在方法的上下文加上 l 自身, 作为目标方法的上下文. 对应的 *Select*( )函数定义为(下划线表示无关的参数):

$$Select(c, l, \_) = [l', \cdots, l'', l],$$
$$\text{其中, } c = [l', \cdots, l''].$$

因此, 在调用点敏感中, 一个方法的上下文由它的 l 和 l 所在方法的调用点等一系列调用点构成, 本质上是模拟程序动态运行时每个方法的调用栈. 然而, 上述 *Select*( )函数分析递归方法时会产生无穷无尽的上下文; 此外, 真实 Java 程序运行时的调用栈往往很深, 若 *Select*( )函数模拟所有可能的调用栈, 则会产生数量巨大的上下文, 使得指针分析开销过大, 无法在合理时间内完成. 因此, 为了解决这 2 个问题, 实际的调用点敏感会限制上下文的层数, 称为 k 限制(k-limiting), 即选取最靠近目标方法的 k 个调用点作为上下文. 例如, k=1 时, 相应 *Select*( )函数定义为

$Select(\_, l, \_) = [l]$, 即只取最近一个调用点作为上下文. k 限制损失了精度, 但可以显著提升指针分析的效率和可扩展性(scalability), 因此本文介绍的其他上下文敏感技术也通常会采取这种方式.

2)对象敏感. 2002 年, Milanova 等人针对面向对象语言的特征, 提出对象敏感(object sensitivity)技术[42-43]. 具体地说, 对象敏感技术使用调用点的接收者对象作为上下文, 相应的 *Select*()函数定义为:

$$Select(\_, \_, c' : o_i) = [o_j, \cdots, o_k, o_i],$$

$$其中 c' = [o_j, \cdots, o_k].$$

对象敏感技术的思想在于它捕捉了面向对象程序的一个特征, 即许多方法的行为都是对接收者对象(即 this 变量指向的对象)状态的修改(如 *setX*()方法)与访问(如 *getX*()方法), 而以接收者对象作为上下文, 可以区分对于不同对象的操作, 从而提升精度. 与调用点敏感一样, 实际应用中, 对象敏感技术也使用 k-limiting 的方式限制上下文层数. 当上下文层数限制 k 一致时, 理论上调用点敏感与对象敏感的精度无法直接比较(即存在一些情况, 调用点敏感更准确, 也同时存在一些情况使得对象敏感可以获得更高精度), 但已有研究表明, 在实际情况中, 对于 Java 程序, 对象敏感的效率与精度都优于调用点敏感[20,34,42-44]. 因此, 对象敏感也是提升面向对象语言指针分析精度的主要技术.

对象敏感综合表现优于调用点敏感, 但它处理 Java 静态方法调用(static method call)时, 由于静态方法没有接收者对象, 因此已有对象敏感通常只能使用调用者(caller)的上下文作为目标方法的上下文, 不能根据调用点信息进一步区分上下文. 对于这一问题, Kastrinis 等人[45]提出混合上下文敏感(hybrid context sensitivity)技术, 对于实例方法调用使用对象敏感, 而对于静态方法调用使用调用点敏感, 从而结合了这 2 种上下文敏感技术获得了更好的精度.

3)类型敏感. 作为面向对象程序, 许多 Java 程序会创建大量对象, 因此使用对象敏感技术分析这类程序且上下文层数大于 1 时, 容易产生过多上下文, 导致指针分析开销过大. 对此, Smaragdakis 等人[44]在对象敏感的基础上提出类型敏感(type sensitivity)技术以提升指针分析效率与可扩展性, 对应的 *Select*()函数定义为:

$$Select(\_, \_, c' : o_i) = [t, \cdots, t', InType(o_i)],$$

$$其中 c' = [t, \cdots, t'].$$

其中函数 $InType(o_i)$返回对象 o_i 的创建点所在的类型. 以图 2 的代码片段为例, 方法 *foo*()有 2 个接收者对象 o_3 与 o_5, 因此使用对象敏感分析时会产生 2 个上下文 $[o_3]$ 与 $[o_5]$; 而使用类型敏感分析时, 它的上下文是 $InType(o_3) = InType(o_5) = [X]$, 因此只有一个上下文. 不难看出, 类型敏感技术生成上下文时合并了同一个类内创建出的对象, 因此其本质上是对象敏感的一种粗粒度抽象, 生成的上下文数量通常也小于对象敏感, 从而具有更快的速度. 相应的, 类型敏感的精度也差于对象敏感. Smaragdakis 等人[1,44]认为使用一个对象创建点所在的类作为上下文, 也能较好地保留使用该对象本身作为上下文时携带的信息, 因此不会导致太多精度丢失. 实验结果表明, 类型敏感的精度略差于对象敏感, 而对于一些对象敏感难以分析的程序, 类型敏感具有显著更好的效率和可扩展性[20,44].

```
1  class X {
2    void main(){
3      Y y1 = new Y(); // o3
4      y1.foo();
5      Y y2 = new Y(); // o5
6      y2.foo();
7    }
8  }
9  class Y {
10   void foo(){…}
11 }
```

Fig. 2 An example for illustrating type sensitivity

图 2 解释类型敏感的例子

2.3 选择性上下文敏感

给定一个程序, 传统上下文敏感技术将对该程序的所有方法应用上下文, 这种方式一般在上下文层数大于 1 的情况下(例如 2 层的调用点敏感或对象敏感技术), 难以在合理的时间(例如几个小时)内完成对较大规模或复杂程序的分析, 我们也称这种分析不具备良好的可扩展性(scalability). 为了使得指针分析能够对那些较大规模或复杂的程序取得良好精度的同时具备更好的可扩展性(即有效的精度与效率之间的平衡), 选择性上下文敏感技术(selective context sensitivity)被提出, 并成为近年来针对 Java 指针分析的研究热点. 可以从 2 个角度来理解“选择性”, 一个是对程序的每一个方法选择出对其有效的上下文元素来构建其上下文; 另一个是最主流且成效最明显的, 它只针对程序中的一部分被选择出来的方法应用上下文, 其它方法不用上下文(或者对不同方法应用不同种类的上下文). 本节按照这 2 种类别分别阐述不同的选择性上下文敏感技术.

1)选择上下文元素. 传统的上下文敏感使用的

都是连续的上下文元素，例如，如果是调用点敏感技术，l_3 会调用 l_2，l_2 会调用 l_1，那么 $[l_3,l_2,l_1]$ 就会作为 3 层上下文元素来被使用. 然而，Tan 等人[46] 发现连续上下文中的很多上下文元素在很多情况下并不能提升精度，而这些上下文元素由于占用了上下文资源，比如，如果只能用 2 层上下文，那么该例中的 l_3 如果是能够帮助提升精度的上下文元素而 l_2 不是，但是由于传统方法的上下文元素是连续的，因此也只能舍弃 l_3. 为了解决该问题，Tan 等人[46] 提出了对象分配图（object allocation graph），并将识别有效上下文元素的问题转换为在对象分配图上识别有效路径的问题. 如果使用同样的上下文层数，该方法可以被证明总能取得相较于传统方法更好的分析精度.

Jeon 等人[47] 提出的上下文隧道的概念（context tunneling）就是使用机器学习的方法来改进文献 [46] 工作，即选出对精度提升有效的上下文元素，使得上下文敏感下的指针分析不再无条件地随着每次方法调用而更新上下文元素，从而避免了在上下文层数有限的情况下，重要的上下文元素会被不重要的上下文元素无条件覆盖. 与文献 [48] 类似，文献 [47] 工作中的启发式规则是使用一系列程序特征来表述的通过机器学习算法搜索习得的规则. 尽管基于机器学习的方法可解释性差、预分析时间长且存在过拟合的情况，但实验结果表明，应用该技术指导的 1 层上下文敏感指针分析在精度和可扩展性上都优于同类型的 2 层上下文敏感指针分析.

在后续工作中，Jeon 等人[49] 提出了一种 Obj2CFA 技术，它将应用上下文隧道技术的对象敏感指针分析转变为一种精度更高的、应用上下文隧道的调用点敏感指针分析. 这一工作表明，对层数为 k 的上下文敏感指针分析，对象敏感的优越性仅存在于传统的强制使用最新 k 个上下文元素的情况下；而在应用上下文隧道技术后，分析可以自由保留任意 k 个上下文元素序列作为上下文，此时通过 Obj2CFA 技术几乎可以用调用点敏感来完全模拟对象敏感，反之则不行. 实验表明，通过该技术从对象敏感的指针分析转换得到的调用点敏感指针分析在精度和可扩展性方面更佳.

2）选择程序方法. Smaragdakis 等人[50] 为了能够让指针分析有更好的可扩展性，提出了自省式分析（introspective analysis），该方法人工选取 6 种不同的程序指标（metrics）（例如，程序方法参数的指向集合的大小）；通过运行上下文非敏感指针分析作为预分析将这些指标值算出；根据这些值和阈值的比较作为选择哪些方法需要上下文敏感的判断条件，其余方法用上下文非敏感来分析，这样会节省计算和维护上下文信息的开销. 实验数据显示，该方法确实能够使得一些之前难以分析的程序在合理时间内分析出结果，且取得了良好的分析精度.

Jeong 等人[48] 提出了一种依赖于机器学习得到的启发式规则，并用该规则决定进行指针分析时程序中的各个方法应当使用多少层上下文（包括使用 0 层，即不应用上下文）. 在该工作中，1 个方法用 25 个原子特征来描述（例如方法中是否含有某类语句），并通过使用机器学习，执行一种贪心算法来得到所需的启发式规则，这一启发式规则被表示为至多 25 个原子特征的析取式. 该技术能够使程序中仅有部分方法在上下文中被分析；且被启发式规则认为对精度影响重要的方法将在更深层数的上下文中分析. 这种基于机器学习的方法最后选取指标的可解释性较差，比如“当某个方法没有 X 和 Y 关键字和语句时，需要对其应用上下文”，这很难让人理解该选择背后的逻辑. 此外，机器学习方法在一定程度上有过拟合的嫌疑，因为其有效性可能部分来源于在学习阶段分析了样本程序依赖的 Java 标准库，而这在一定程度上会影响分析其它程序的结果（因为很多程序都大量依赖 JDK 标准库）. 尽管如此，实验数据表明，依赖于机器学习挑选出的程序方法，确实在很大程度上能够提高指针分析的效率，同时取得良好的精度.

无论是上述的自省式分析还是机器学习方法，在可解释性方面都不够好，主要原因是这些方法实际上并没有从本质上出发解决选择性上下文敏感的核心问题，即到底为什么有些程序方法可以得益于上下文敏感（Li 等人[51] 将这些方法称为精度关键方法（precision-critical methods），而有些方法即使应用上下文敏感技术来分析也不会提高精度.

Li 等人[51] 首次提出了一套系统的理论解释模型用以回答此问题. 该模型由 3 种基本流（直通流、封装流和拆装流）以及这些流的组合构成. 给定任何一个程序，指针分析在上下文非敏感情况下精度丢失的原因（或上下文敏感情况下的精度提升的原因）都可以通过这些基于流的模型来准确解释. 此外，在文献 [51] 中，Li 等人提出了 Zipper 方法，它将上述理论解释模型通过精度流图来表达，并将识别精度关键方法的问题转换为在精度流图上的图可达性问题.

受理论解释模型的启发，Li 等人的后续工作[20] 在该模型基础上做了进一步拓展，在识别精度关键方法的基础上还能识别速度关键方法（scalability-

threaten methods), 提出了目前针对难以分析的 Java 程序在效率与精度平衡方面表现最佳的方法之一 Zipper-Express(Zipper^e). Zipper^e 具有简单、容易理解且具有很强的可解释性等特点, 能够平均 26 倍加速已有精准指针分析(2 层的对象敏感), 且对于已有文献中(包括国际基准 Java 测试集 DaCapo 的程序)无法在 3 小时内分析出结果的程序, Zipper^e 可以在平均 11 分钟分析出精确的结果; 不仅如此, 对于一些复杂程序, Zipper^e 指导的上下文敏感指针分析的速度甚至比上下文非敏感指针分析还快, 这是由于 Zipper^e 排除了速度关键方法, 从而显著提升了效率; 在此基础上又通过将上下文敏感应用于精度关键方法, 相比于上下文非敏感分析, 减少了大量假指向信息的传播, 从而达到了比上下文非敏感指针分析更高的效率. 这一技术也打破了过去上下文敏感指针分析效率无法超越上下文非敏感指针分析的认识.

值得一提的是, 如文献 [20,51] 中所描述, 识别精度关键方法的理论解释模型(3 种流和流的组合)实际上是识别出方法中具体哪些程序变量是需要在分析中应用上下文的(即静态时这些流覆盖的变量), 这使得方法(Zipper 与 Zipper^e)天然可以在变量或对象的粒度(相较于其实验中基于程序方法的粒度更细)上应用上下文敏感技术.

Lu 等人[52-53] 的工作也是在程序变量/对象的粒度应用上下文敏感, 不同的是, 该工作识别这些变量/对象的方法是基于上下文无关语言可达性(context free language reachability, CFL-reachability)技术完成, 并将其实现出来. 具体而言, 该方法首先将程序中的值的流动形式化为指针赋值图(pointer assignment graph, PAG), 然后基于 PAG 将对象敏感的指针分析表达为一个新的 CFL 可达性公式, 并通过在图上解 CFL 可达性问题来选取出同时满足存在值的流入和流出这一约束条件的节点, 最后对这些节点(包括了变量与对象)应用对象敏感进行指针分析. 实验结果表明该方法能够在加速对象敏感指针分析的同时保持精度不变.

He 等人[54] 提出了一种称为 Turner 的方法来取得对象敏感指针分析效率和精度的新的平衡. 该工作在文献 [52] 的基础上, 进一步减少需要应用上下文的变量或对象. 该技术首先通过定义并找到程序中的 2 类对象: 顶部容器(top container)和底部容器(bottom container), 并对这 2 类对象不应用上下文, 继而选择出对因这 2 类对象不应用上下文而不再满足值的流动约束条件的变量或对象, 并对这些变量或对象也不应用上下文. 实验表明, Turner 能够在 2 个现有指针分析工具 Eagle[52] 和 Zipper[51] 中取得新的效率与精度的平衡: 在精度和 Eagle 几乎一致的同时快于 Eagle; 在精度高于 Zipper 的同时, 在一部分基准程序上快于 Zipper. 然而, 文献 [54] 的工作并没有与 Zipper^e[20] 进行比较, 但是通过与 Zipper 的对比以及实验所展示的数据不难看出, Turner 在分析精度略高于 Zipper^e 的同时, 其分析效率相比于 Zipper^e 应该还有很大差距(有不少 Turner 无法在给定时间内完成分析的程序, Zipper^e 可以完成对这些程序的分析).

针对如何利用选择性上下文敏感指针分析确保分析的可扩展性的同时尽可能地利用内存资源最大程度地提升分析精度, Li 等人[55] 提出了 Scaler 方法来解决这一问题. 该方法将上下文敏感指针分析在不同上下文情况下的可能开销与抽象出的内存承载能力关联起来, 并基于文献 [46] 中提出的对象分配图提出了预估上下文敏感指针分析开销的方法, Scaler 方法可以为不同程序自动分配不同的上下文敏感元素和长度以充分利用内存可承载的能力并最大化分析精度. 实验结果表明, Scaler 具备非常好的分析可扩展性以及良好的分析精度, 且其分析精度与效率的平衡可根据实际情况由用户方便地支配调节.

上文提到的工作 [20,46,51–52,56] 都是先基于某种图的结构来进行预分析处理, 然后根据各自的预分析结果指导后续的选择性上下文敏感指针分析. Jeon 等人[57] 将这些方法归类为一种基于图的启发式方法(graph-based heuristics), 并用机器学习的方法去挖掘图中对于选择性上下文敏感指针分析有用的信息. Jeon 等人[57] 首先定义了一种特征语言来描述基于图的结构, 这一语言用节点自身和前驱后继的入度或出度信息来描述一个节点的特征; 然后设计了一个机器学习算法来学习用该特征语言表述的启发式规则. 具体而言, 该技术会基于程序的对象创建图(object allocation graph, OAG)[46] 和字段指向图(field points-to graph, FPG)[56] 来分别学习 2 种启发式规则, 并将这 2 种启发式规则分别用于指导指针分析的 2 个方面: 决定一个方法应该用何种上下文敏感(包括不使用上下文)以及决定一个堆对象应当用创建点抽象还是基于类型的抽象. 实验结果表明使用该技术指导的指针分析优于部分现有的采用人工设计的启发式规则的技术.

已有的选择性上下文敏感指针分析工作在效率与精度平衡方面一般都侧重于使得分析尽可能快的同时还能保持良好的精度, Tan 等人[34] 的工作侧重平

衡的另一端，即如何取得高精度的指针分析. 它提出了 Baton，一个获取高精度指针分析的元框架. 针对任何程序 P，给定任何一组选择性上下文敏感技术作为输入，只要这些方法能够在合理时间内分析完 P，Baton 就可以被证明总能保证在完成分析的情况下输出比任何方法都更精确的选择性上下文敏感指针分析. Baton 由 2 部分构成："团结"(Unity)组件与"接力"(Relay)组件，Unity 组件最大限度利用已有输入方法提高分析精度，当 Unity 无法在有限时间内完成分析后，Relay 组件会通过"精度累积传递"的方式继续提高分析精度. 实现结果表明，在所有精度指标上，对于所有的待评估程序(包括已有文献及基准测试集中公认的难以分析的程序)，相比于已有工作，Baton 总能取得最好的精度且很多情况下精度的提升是非常显著的.

3 堆对象抽象

Java 指针分析计算程序中的变量在动态运行时所能指向对象的集合，作为典型的面向对象语言，Java 程序通常在运行时会创建大量对象，且由于循环和递归的存在，理论上一个 Java 程序在动态运行时可创建无穷无尽的对象. 因此指针分析需要对程序中的对象采取抽象的表示，以保证分析中只需处理有限数量的对象，这类技术被称为堆对象抽象，或简称堆抽象(heap abstraction)[58]. 使用不同的堆抽象技术可对指针分析的效率与精度带来显著的影响[18,56,58-59].

目前，Java 指针分析使用最广泛的堆抽象技术是创建点抽象(allocation-site abstraction)，它用程序中的每个创建点 i(即对象创建语句，如 i: x = new T())表示动态运行时所有由 i 创建出来的对象. 对应 $Heap()$ 函数(见表 3 与表 5)的定义为 $Heap(i) = i$. 创建点抽象具有良好的精度，几乎所有主流 Java 指针分析框架都支持这种堆抽象[38-39,60].

虽然创建点抽象精度良好，但一些复杂的 Java 程序包含大量的创建点，使用创建点抽象会在指针分析过程中产生大量对象，造成很大的开销. 主流的指针分析框架[38-39,60]都会提供对一些特定对象按类型合并的功能，如对于 StringBuilder 或 StringBuffer 类型的对象、字符串常量等对象，将所有同类型的对象合并抽象成一个对象. 这些对象在程序中通常有许多创建点，因此合并这些创建点对应的对象之后可以对指针分析效率带来一定提升.

然而，合并特定对象并不能通用地解决创建点抽象导致对象数量过多的问题. 对此，Tan 等人[56]提出了一种新的系统性堆抽象技术 Mahjong. Mahjong 的核心思想是判断任意 2 个堆对象的所有嵌套字段指向对象的类型是否相同，这需要枚举字段指向图(field points-to graph)所有可能的字段访问路径，这是具有指数级复杂度的操作. 为了解决该问题，Mahjong 将程序的字段指向图映射成时序自动机(sequential automata)，一种六元组自动机，那么判断 2 个堆对象是否等价的问题就自然地转换成判断等价自动机的问题，该问题可以通过对判别一般自动机等价性的 Hopcroft-Karp 算法[61]稍加修改而解决，最终能够得到近似线性复杂度的判别算法. 对于关注指针指向对象类型的应用，Mahjong 甚至可以成百倍地加速已有精准指针分析(3 层的对象敏感技术)，同时维持了传统指针分析 99.9% 的精度.

在第 2 节中可以看到，堆抽象技术可以与上下文敏感结合，将抽象对象进一步细分，形成上下文敏感堆抽象(context-sensitive heap abstraction). 例如创建点抽象对于创建点 i 产生一个对象 o_i，而假设 i 所在的方法有 3 个上下文，则对应的上下文敏感堆抽象可以将 o_i 细分成 3 个抽象对象(如 c:o_i, c':o_i, c'':o_i)，分别表示不同上下文中创建出的对象. 而这种更精细化的堆抽象也能带来更好的分析精度.

除了创建点抽象(以及由其衍生出的相关技术)之外，还有一类差异较大的堆抽象技术，即基于访问路径(access path)的堆抽象，其中每个访问路径由 1 个变量跟随 0 个或多个字段组成，即形如 x, $x.f$, $x.f.g$ 的指针表达式. 基于访问路径的指针分析通常使用访问路径之间的别名关系表示堆的信息[1,58]，因此会在运行过程中同步计算并维护别名关系(例如分析语句 $x = y$ 时，生成与变量 x, y 相关访问路径的别名，如 $\{x, y\}$, $\{x.f, y.f\}$ 等).

复杂的 Java 程序会包含数量巨大的访问路径. 此外，程序中的循环引用理论上可形成数量无限的访问路径，如语句 $x.f = x$ 可使得 x, $x.f$, $x.f.f$, $x.f.f.f$…都是有效的访问路径，导致指针分析无法穷举. 因此，实际使用访问路径时，通常也会采用类似上下文敏感的 k-limiting 技术，即对访问路径中包含的字段数量设置一个上限 k，长度超过 k 的访问路径则被合并到相同前缀的访问路径[58]，例如 k=1 时，$x.f$*表示所有以 $x.f$ 开头的访问路径，包括 $x.f$, $x.f.g$, $x.f.h$ 等. 由于访问路径可以在一些情况下方便地做强更新(strong update)，因此目前主要是一些流敏感分析[8,62]使用该技术.

4 复杂语言特性处理

作为一门广泛使用的工业级编程语言，Java具有丰富的语言特性，随着不断地演化，Java的语言特性也越来越多. 然而其中的一些复杂语言特性给指针分析带来了很大的挑战. 如果指针分析不能妥善处理这些复杂语言特性，其结果的可靠性将受到严重的影响[63]. 本节介绍这些复杂语言特性如何影响指针分析以及处理这些特性的代表性工作.

4.1 反 射

Java的反射(reflection)是一套功能强大的机制，允许Java程序由动态信息(主要是字符串)指定其行为(如创建对象、读写字段、调用方法等)，显著提升了Java这一静态类型语言的灵活性. 但反射的动态性也给静态分析造成了很大困难，是公认最难以静态分析的语言特性之一[64-66]，图3用一个反射的典型用例进行说明.其中行2的Class.*forName*()方法根据参数字符串*clsName*返回一个类(比如T)对应的Class对象，T的类名就等于*clsName*. 程序得到T类的Class对象之后，可以通过*newInstance*()方法创建T类的对象(行3)；或通过*getMethod*()方法获取T类某个方法对应的Method对象(行4)，而获取的方法取决于*getMethod*()的参数，即方法名(字符串*mtdName*)与方法的参数列表*X.class*. 得到方法对象*mtd*后，程序可以通过*invoke*()方法对相应的目标方法发起调用(行6). 图3例子中决定反射行为的关键在于字符串*clsName*与*mtdName*的具体内容. 然而，许多情况下，Java程序的字符串来自于外部输入(如命令行、配置文件、网络等)，或经过一系列字符串操作(如截取、拼接等)，因此静态无法得到准确的字符串分析结果，使得反射的副作用难以分析.

```
1 String clsName = …, mtdName = …;
2 Class cls = Class.forName(clsName);
3 Object o = cls.newInstance( );
4 Method mtd = cls.getMethod(mtdName, X.class);
5 X x = new X( );
6 mtd.invoke(o, x);
```

Fig. 3 A typical usage of Java reflection

图3 一个典型的Java反射用例

Livshits等人[67]提出借助指针分析解析反射关键API(如Class.*forName*()、Class.*getMethod*())的字符串参数进而分析出反射调用的副作用. 具体而言，该反射分析与指针分析同时运行并互相依赖，在此过程中，反射关键API字符串参数(如图3中的*clsName*与*mtdName*)的指针集发生变化时，指针分析会触发反射分析，*mtdName*根据指针集中的字符串常量来分析反射调用的行为，并将其相应的副作用反馈给指针分析. 然而该分析只能处理反射参数指向字符串常量的情况，对于其它字符串非常量的情况均无法分析(唯一例外是Class.*newInstance*()的返回值被立即进行向下类型转换(downcasting)这一特定情形，该技术可以利用类型转换的信息对Class.*newInstance*()的副作用进行推导).

Li等人[68]研究了真实Java程序中的反射使用情况，发现在绝大部分情况下，尽管反射调用的参数并非字符串常量(静态无法分析)，但在反射的调用点有许多可利用的信息可以用于分析反射(比如反射调用参数的类型、返回值的向下类型转换等)，并系统性地将这些信息总结为自推导属性(self-inferencing property). 基于这一发现，Li等人[68]提出一套新的静态反射技术Elf，利用自推导属性并结合Java类型系统，在反射字符串参数并非常量的情况下也能推导反射调用的行为. 实验表明，Elf能够有效分析出非常多的、已有工作无法静态解析出的真实反射调用行为，显著提升了反射分析的可靠性(soundness)，并同时取得良好的分析精度.

在Elf的基础上，Li等人[69-70]提出集体推导(collective inference)与懒惰堆建模(lazy heap modeling)的技术，并形成新的反射分析Solar. 集体推导在Elf的基础上进一步增强了推导能力. 懒惰堆建模用于分析由反射调用Class.*newInstance*()或Constructor.*newInstance*()创建但具体类型在创建点未知的堆对象. 对于这类对象，Solar将其传播到程序中使用它们的位置，如向下类型转换，或Method.*invoke*()、Field.*get*()、Field.*set*()的反射调用点等，并更充分地利用程序中这些位置的类型信息以分析反射创建对象的具体类型. 基于更强的反射分析能力，Solar能够识别出程序中未能被完整解析的反射调用点，首次在理论上实现了对于反射分析可靠性边界的静态推导；此外，Solar也能识别出分析不准确的反射调用点，因此可以帮助用户添加轻量级注解就可以满足他们的在可靠性、分析效率等方面的需求.

4.2 本地代码

Java是一门运行在虚拟机(Java Virtual Machine)之上的语言，虚拟机封装了不同平台的功能并支持一套统一的Java语言规范，这使得Java程序具有良好的平台无关性因而可以跨平台执行. 然而，在一些

情况下，例如 Java 程序需要直接与底层平台交互或实现性能攸关的功能时，Java 程序也需要调用能直接运行在宿主平台上的代码，即本地代码（通常由 C/C++实现）. Java 提供了 native 关键字，用于在 Java 程序中声明本地方法（native method），例如 java.lang.System 中的 *arraycopy*()方法：

```
class System { …
public static native void arraycopy(
Object src, int srcPos,
Object dest, int destPos, int length);
… }
```

它的具体实现并非 Java 代码，而是由 C/C++代码按照 Java 本地接口（Java Native Interface, JNI）规范[71]完成的. 由于本地代码并非由 Java 实现，因此上文介绍的 Java 指针分析技术无法用于本地代码的分析. 但本地代码可以对 Java 程序进行诸多操作，如修改对象的字段、回调 Java 方法等，因此如果不能妥善分析本地代码，将会对指针分析结果的可靠性造成显著影响.

目前主流 Java 指针分析框架[38-39,60]处理本地代码的方式是手动地在框架内添加一些对指针分析影响较大的本地方法（如 System.*arraycopy*()）的分析逻辑，当指针分析遇到这些本地代码的调用时，会触发相应分析逻辑，模拟它们对程序中指针的副作用.

为了自动化地分析本地代码回调 Java 函数这一类行为，Fourtounis 等人[72]提出扫描本地代码中的字符串常量并推测其对应的 Java 方法. 具体来说，JNI[71]提供了一系列用于回调 Java 方法的 API，而 Fourtounis 等人[72]注意到这些 API 的参数往往是字符串常量，并且与具体回调的 Java 方法的签名紧密相关，进而在该工作中提出扫描本地代码中的字符串常量，记录 JNI 调用的参数与字符串常量的对应关系，结合传统 Java 分析工具对方法签名的分析，推断该 JNI 调用回调的是哪个 Java 方法. 实验结果表明，该工作可以快速有效地推测出 XCorpus 基准程序库和 Chrome 等真实应用中本地代码的回调方法.

4.3 异　常

异常（exception）是 Java 程序中重要而不可忽视的控制流[73]. 准确的异常分析需要得知程序的 throw 语句以及方法调用可能抛出异常的具体类型，这需要依赖于指针分析的结果，反之，异常分析的结果也能影响指针分析，下面用图 4 中的代码片段进行说明.

```
1  void exception(){
2    try {
3      o.m();
4      …
5      throw e;
6      …
7    } catch (IOException ioe) {
8      ioe.printStackTrace();
9    } catch (RuntimeException rte) {
10     …
11   }
12 }
```

Fig. 4　An example for illustrating exception

图 4　解释异常的例子

图 4 中行 3 与行 5 都有可能抛出异常，一方面，对于行 3 的调用语句，需要分析它的具体目标方法才可知它可能抛出的异常，对于行 5 的 throw 语句，则需要分析变量 *e* 指向的具体异常对象，而这些都需要依赖于指针分析. 另一方面，指针分析若要准确分析行 8 的调用，也需要异常分析对于行 7 catch 语句的分析结果（即分析出变量 *ioe* 会捕获的具体异常对象，该异常可能是由调用 *o.m*()抛出，也可能由 throw 语句抛出）. 由此可见，实际上指针分析与异常分析互相依赖于对方的结果.

针对指针分析与异常分析相互耦合的特点，Bravenboer 等人[74]提出将指针分析与异常分析结合进行的技术. 具体而言，异常分析处理 throw *e* 语句时使用指针分析对于变量 *e* 的结果；而异常分析处理 catch 语句时，会推断哪些异常对象被对应的 catch 语句捕获，并将捕获的异常对象注入到对应 catch 语句的异常形参变量的指针集中，反馈给指针分析. 此外，指针分析在构建调用图时，也会触发异常分析，使其能够沿着调用边向调用点传播目标方法内没有被捕获的异常. 通过文献 [74] 的方式，实现了指针分析与异常分析的协同式迭代计算. 实验显示，与以往不精确的异常分析[19]相比，该技术对于 try-catch 异常对象流的分析精度有显著提升.

在文献 [74] 提出的异常分析中，异常对象会大量流入或者流出程序中的各个方法，因此若将异常当作常规对象进行处理会产生大量开销，同时由于异常分析往往并不关注异常对象本身内部字段的信息，因此在文献 [74] 的基础上，Kastrinis 等人[75]提出了将相同类型的异常对象合并，并且使用这个类型合并后的代表对象（每个类型只有 1 个）表示对应的异常对象. 实验中，该方式使异常分析几乎在不损失精度的情况下能够显著提升分析性能.

4.4 invokedynamic 与 Lambda 表达式

Java 7 引入了新的字节码调用指令 invokedynamic，

它与已有的调用指令 invokestatic, invokespecial, invokevirtual, invokeinterface 最大的区别在于,已有调用指令都是依照固定的规则解析出目标方法,而 invokedynamic 允许用户编写代码指定目标方法的解析规则,在动态时依据用户代码来决定调用点与目标方法的链接,而这种动态性对静态分析造成了很大困难. Java 引入 invokedynamic 指令的主要原因是为了使 Java 虚拟机(JVM)能更方便地支持动态类型语言,但 invokedynamic 指令也有其他应用,其中最为重要的便是 Lambda 表达式. Java 8 引入 Lambda 表达式显著地提升了 Java 编程的便利性,这也是现代 Java 编程中被频繁使用的特性. 如果指针分析不能妥善处理 Lambda 表达式,将会缺失程序中的许多行为. 然而 Lambda 表达式是基于 invokedynamic 实现的,并结合了动态代码生成,因此对于 Lambda 表达式的分析也是一个挑战.

Soot[39] 是目前最流行的 Java 分析框架之一,它选择在前端生成中间代码的过程中将程序内与 Lambda 表达式相关的 invokedynamic 指令转换为非 invokedynamic 的调用语句,从而避免分析 invokedynamic 指令. 然而这种方式损失了一部分原程序中的语义,并且 Soot 的指针分析也不支持对于 invokedynamic 指令的分析.

目前在指针分析中对 Lambda 表达式与 invokedynamic 处理得最完善的是 Fourtounis 等人提出的对 invokedynamic 指令进行深度静态建模的技术[76]. 该技术在模型中按照 invokedynamic 的语义,识别并分析用户编写的目标方法解析逻辑. 然而,由于 invokedynamic 机制具有高度的动态性与开放性,该技术并不能处理所有 invokedynamic 的使用场景. 而且,与通用的 invokedynamic 使用不同,Java Lambda 表达式的实现逻辑虽然复杂,但其总体行为模式是固定且可预期的,因此该工作对 Lambda 表达式进行额外的针对性建模,它包含了 Lambda 表达式的识别、目标方法的解析、参数与返回值的传递、外部变量捕捉等一系列行为的分析逻辑,能够准确地在指针分析中处理 Lambda 表达式.

5 非全程序指针分析

现代 Java 程序规模复杂,要进行一次全程序指针分析往往需要不小的开销,使得全程序分析在一些时间、空间资源有限的应用场景下不能很好地满足用户需求. 因此研究人员也在探索如何在不运行全程序指针分析的前提下获取到相应指针的分析结果,而目前相关的技术主要有 2 类:需求驱动分析(demand-driven analysis)与增量分析(incremental analysis).

5.1 需求驱动分析

需求驱动分析[77] 的使用场景是指指针分析的一些应用(如编译优化、故障检测、程序理解等)只需要了解程序中特定指针的指向关系,此时应用就会向指针分析发起相应的查询(query),例如“A 类第 10 行处变量 x 指向哪些对象?”,需求驱动指针分析只返回应用所需指针的指向关系即可,而无需对全程序进行分析,从而节省分析的开销.

Sridharan 等人[78] 提出了一个基于 CFL 可达性的需求驱动指针分析算法. 该工作将程序转换为一种图表示,其中节点表示变量和对象,边表示指针分析相关的程序语句(如复制语句 $x = y$、字段存储语句 $y = x.f$). 在图表示的基础上,该工作将字段敏感的指针分析建模为 CFL 可达性问题,指针 x 指向对象 o 的指向关系被视为程序中存在一条从 x 到 o 的数据流路径,且该路径上的字段写入边(field write),对应左括号与字段读取边(field read),对应右括号连接成的括号序列可构成一个平衡的括号串. 该工作提出的算法在处理“变量 x 指向哪些对象”的查询时,从 x 对应的节点开始,在图上搜索 CFL 可达的对象节点. 为了提高算法的可扩展性,该工作将算法分为估算和精化 2 个阶段. 估算阶段会在 CFL 可达性搜索时跳过一些可能不可行的路径;若有足够的时间预算,则在随后精化阶段中,算法会通过检查在估算阶段被跳过的部分来提升分析的精度. 在后续工作中,Sridharan 等人[79] 进一步将该方法结合上下文敏感技术,提升了分析的精度.

Yan 等人[80] 提出了过程内可达性摘要技术,减少了分析时的重复计算,提升了传统的基于 CFL 可达性需求驱动别名分析的效率. 具体而言,该工作使用符号指向图(symbolic points-to graph, SPG)[81] 作为其程序表示,一个方法的 SPG 包含其内部的指向关系,并使用占位的符号节点来代表方法外创建的对象. 基于此程序表示,该工作设计了一个算法能够在线地分析出过程内可达性摘要,并将其应用到别名分析中,减少了对于频繁分析方法的重复计算. 实验结果表明,该工作相比于基于精化的需求驱动指针分析[79] 在相同的运行时间限制下拥有更好的精度,并且使用过程内摘要技术能提升一定的运行效率.

Shang 等人[82] 提出了一个使用 CFL 可达性摘要

技术的需求驱动指针分析，它无需大开销地全程序预处理分析. 具体而言，该工作提出在过程内进行字段敏感的部分指针分析，并将其作为过程内分析的指向关系摘要；之后该摘要可以被重复利用到相同甚至不同上下文中，减少过程内分析的重复计算. 实验结果表明，该工作相比于基于精化的需求驱动指针分析[79]，在类型转换检查、空指针检查和工厂方法分析这 3 种分析应用中均能取得一定的效率提升.

Späth 等人[62] 首次提出了一个能够同时提供指向关系和别名关系信息的需求驱动、流敏感、上下文敏感指针分析. 该工作在处理一次查询时，首先会在程序控制流图上从待查询的变量开始逆向搜索，找出其可能指向的对象创建点；然后，从这些对象创建点开始正向搜索，找出其它可能指向同一对象的变量. 在逆向和正向搜索 2 个阶段，该工作使用了 IFDS 框架[83] 实现流敏感和上下文敏感，并利用过程摘要提高算法的效率. 实验结果表明，该工作的性能和精度均优于已有的技术[79-80].

5.2 增量分析

实际软件开发过程中，软件是由开发人员通过一次次更新与修改累积而成，而每次更新的部分通常只占程序中很小的一部分. 针对软件开发的这一特点，增量分析应运而生. 具体而言，增量分析通常需要先针对程序指定版本运行一次全程序分析，在此基础上，当开发人员更新程序时，增量分析只针对程序变化的部分展开分析，分析变化部分自身指针集的变化以及它们对程序其他部分的影响，而对于未受影响的部分，可以直接复用之前指针分析的运行结果，从而节省了分析的开销.

Lu 等人[84] 提出了一个基于 CFL 可达性的路径依赖追踪的增量指针分析算法. CFL 可达性分析将指向关系表示为一种图可达性关系，该工作使用了一个辅助数据结构，在运行图可达性搜索的同时，记录路径上经过的变量. 在程序发生变化时，只会重新计算那些受到影响的路径，即包含被修改变量的路径，并更新 CFL 可达性结果和路径信息. 该技术因为记录所有路径追踪信息的内存，开销很大，因此文献[84] 提出了一种优化追踪策略，减少存储程序中不经常修改部分的追踪信息的开销.

Liu 等人[85] 提出了一个基于 Andersen 指针分析算法的并行增量指针分析算法 IPA. 相比于之前的技术，该工作通过充分挖掘 Andersen 指针分析算法的传递性质，使用强连通分量优化，减少了大量重复计算或者图可达性分析，显著提高了增量指针分析的性能. 具体而言，Andersen 算法的传递性质保证了对指针赋值图（pointer assignment graph, PAG）进行强连通分量缩点优化；并消除 PAG 图中的环之后，在删除一条语句时，只需要检查受影响节点的邻居节点即可，而无需遍历整个 PAG. 在传递性质的基础上，该工作继续证明了增量指针分析算法的修改幂等性，即添加或删除语句后，指向关系在 PAG 上的传播是一个幂等操作，从而设计了一个无同步（synchronization-free）的并行指针分析算法. 实验结果表明，该工作与之前基于图可达性分析的技术[84] 相比，有显著的性能提升. 在后续工作中，Liu 等人[86] 进一步结合上下文敏感技术，提出了一种支持调用点敏感和对象敏感的增量指针分析算法. 该工作解决了处理方法调用删除时的重复计算，保证了在有上下文敏感的前提下处理添加、删除语句的分析结果的可靠性.

6 总　　结

本文从指针分析的核心算法入手，分别在其基础上就调用栈抽象（上下文敏感）与堆抽象技术展开介绍与讨论. 此外，想要有效地分析真实世界的 Java 程序，指针分析势必要处理 Java 的复杂语言特性，因此本文也介绍了主流的相关技术. 除了传统的全程序指针分析，本文也介绍了另外 2 种重要的指针分析，即，需求驱动分析与增量分析.

本文相比于已有的 Java 指针分析综述工作，补充了自 2015 年之后的重要研究文献，尤其是对近年来该方向的研究热点——选择性上下文敏感技术进行了系统性的梳理与讨论. 我们期待该文可以为国内研究人员了解 Java 指针分析研究工作提供便利的参考.

作者贡献声明：谭添、李樾负责撰写论文；马晓星、许畅和马春燕负责修改论文.

参　考　文　献

[1] Smaragdakis Y, Balatsouras G. Pointer analysis[J]. Foundations and Trends in Programming Languages, 2015, 2(1): 1−69

[2] Zhang Jian, Zhang Chao, Xuan Jifeng, et al. Recent progress in program analysis[J]. Journal of Software, 2019, 30(1): 80−109 (in Chinese)
(张健, 张超, 玄跻峰, 等. 程序分析研究进展[J]. 软件学报, 2019, 30(1): 80−109)

[3] Sridharan M, Chandra S, Dolby J, et al. Alias analysis for object-

oriented programs［G］//LNCS 7850: Aliasing in Object-Oriented Programming. Types, Analysis and Verification. Berlin: Springer, 2013: 196−232

［4］ Chandra S, Fink S J, Sridharan M. Snugglebug: A powerful approach to weakest preconditions［C］//Proc of the 30th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2009: 363−374

［5］ Naik M, Aiken A, Whaley J. Effective static race detection for Java［C］//Proc of the 27th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2006: 308−319

［6］ Naik M, Park C S, Sen K, et al. Effective static deadlock detection［C］//Proc of the 31st Int Conf on Software Engineering. Piscataway, NJ: IEEE, 2009: 386−396

［7］ Wang Lei, He Dongjie, Li Lian, et al. Sparse framework based static taint analysis optimization[J]. Journal of Computer Research and Development, 2019, 56(3): 480−495 (in Chinese)
(王蕾, 何冬杰, 李炼, 等. 基于稀疏框架的静态污点分析优化技术[J]. 计算机研究与发展, 2019, 56(3): 480−495)

［8］ Arzt S, Rasthofer S, Fritz C, et al. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android Apps［C］//Proc of the 35th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2014: 259−269

［9］ Grech N, Smaragdakis Y. P/Taint: Unified points-to and taint analysis［J］. Proceedings of the ACM on Programming Languages, 2017, 1(OOPSLA): 1−28

［10］ Livshits V B, Lam M S. Finding security vulnerabilities in Java applications with static analysis［C］//Proc of the 14th USENIX Security Symp. Berkeley, CA: USENIX Association, 2005: 271−286

［11］ Avots D, Dalton M, Livshits V B, et al. Improving software security with a C pointer analysis［C］//Proc of the 27th Int Conf on Software Engineering. New York: ACM, 2005: 332−341

［12］ Li Yue, Tan Tian, Zhang Yifei, et al. Program tailoring: Slicing by sequential criteria［C］//Proc of the 30th European Conf on Object-Oriented Programming. Wadern, Saarland: Schloss Dagstuhl - Leibniz-Zentrum Fuer Informatik, 2016, 15: 1−15: 27

［13］ Sridharan M, Fink S J, Bodík R. Thin slicing［C］//Proc of the 28th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2007: 112−122

［14］ Fink S J, Yahav E, Dor N, et al. Effective typestate verification in the presence of aliasing［J］. ACM Transactions on Software Engineering and Methodology, 2008, 17(2): 1−34

［15］ Pradel M, Jaspan C, Aldrich J, et al. Statically checking API protocol conformance with mined multi-object specifications［C］//Proc of the 34th Int Conf on Software Engineering. Piscataway, NJ: IEEE, 2012: 925−935

［16］ Lhoták O, Smaragdakis Y, Sridharan M. Pointer analysis (Dagstuhl Seminar 13162)［R］. Wadern, Saarland: Schloss Dagstuhl - Leibniz-Zentrum Fuer Informatik, 2013

［17］ Weihl W E. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables［C］//Proc of the 7th ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages. New York: ACM, 1980: 83−94

［18］ Lhoták O, Hendren L J. Scaling Java points-to analysis using SPARK［G］//LNCS 2622: Proc of the 12th Int Conf on Compiler Construction. Berlin: Springer, 2003: 153−169

［19］ Bravenboer M, Smaragdakis Y. Strictly declarative specification of sophisticated points-to analyses［C］//Proc of the 24th Annual ACM SIGPLAN Conf on Object-Oriented Programming, Systems, Languages, and Applications. New York: ACM, 2009: 243−262

［20］ Li Yue, Tan Tian, Møller A, et al. A principled approach to selective context sensitivity for pointer analysis［J］. ACM Transactions on Programming Languages and Systems, 2020, 42(2): 1−40

［21］ Andersen L O. Program analysis and specialization for the C programming language［D］. Copenhagen: University of Copenhagen, DIKU, 1994

［22］ Hardekopf B, Lin C. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code［C］//Proc of the 28th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2007: 290−299

［23］ Hardekopf B, Lin C. Semi-sparse flow-sensitive pointer analysis［C］//Proc of the 36th Annual ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages. New York: ACM, 2009: 226−238

［24］ Yu Hongtao, Xue Jingling, Huo Wei, et al. Level by level: Making flow- and context-sensitive pointer analysis scalable for millions of lines of code［C］//Proc of the 8th Annual IEEE/ACM Int Symp on Code Generation and Optimization. New York: ACM, 2010: 218−229

［25］ Sui Yulei, Xue Jingling. SVF: Interprocedural static value-flow analysis in LLVM［C］//Proc of the 25th Int Conf on Compiler Construction. New York: ACM, 2016: 265−266

［26］ Hind M. Pointer analysis: Haven't we solved this problem yet?［C］//Proc of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering. New York: ACM, 2001: 54−61

［27］ Gosling J, Joy B, Steele G, et al. The Java® Language Specification, Java SE 17 Edition［S/OL］. Austin, Texas: Oracle America, Inc. , 2021.［2022-10-20］.https://docs.oracle.com/javase/specs/jls/se17/html/index.html

［28］ Fähndrich M, Foster J S, Su Zhendong, et al. Partial online cycle elimination in inclusion constraint graphs［C］//Proc of the ACM SIGPLAN 1998 Conf on Programming Language Design and Implementation. New York: ACM, 1998: 85−96

［29］ Pearce D J, Kelly P H J, Hankin C. Online cycle detection and difference propagation for pointer analysis［C］//Proc of the 3rd IEEE Int Workshop on Source Code Analysis and Manipulation. Piscataway, NJ: IEEE, 2003: 3−12

[30] Sharir M, Pnueli A. Two approaches to interprocedural data flow analysis [R]. New York: New York University, 1978

[31] Lhoták O, Hendren L J. Context-sensitive points-to analysis: Is it worth it? [G] //LNCS 3923: Proc of the 15th Int Conf on Compiler Construction. Berlin: Springer, 2006: 47−64

[32] Whaley J, Lam M S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams[J]. ACM SIGPLAN Notices, 2004, 39(6): 131−144

[33] Xu Guoqing, Rountev A. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis [C] //Proc of the 2008 Int Symp on Software Testing and Analysis. New York: ACM, 2008: 225−236

[34] Tan Tian, Li Yue, Ma Xiaoxing, et al. Making pointer analysis more precise by unleashing the power of selective context sensitivity [J]. Proceedings of the ACM on Programming Languages, 2021, 5(OOPSLA): 1−27

[35] Tan Tian, Li Yue. Tai-e: A static analysis framework for Java by harnessing the best designs of classics [J]. arXiv preprint, arXiv: 2208,00337

[36] Tan Tian, Li Yue. Tai-e: An easy-to-learn/use static analysis framework for Java [CP/OL]. [2022-10-22].https://github.com/pascal-lab/Tai-e

[37] He Dongjie, Lu Jingbo, Xue Jingling. Qilin: A new framework for supporting fine-grained context-sensitivity in Java pointer analysis [C] //Proc of the 36th European Conf on Object-Oriented Programming. Wadern, Saarland: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2022, 30: 1−29

[38] Smaragdakis Y. Doop: Framework for Java pointer and taint analysis (using P/Taint) [CP/OL]. [2022-10-22].https://bitbucket.org/yanniss/doop

[39] Bodden E. Soot: A framework for analyzing and transforming Java and Android applications [CP/OL]. [2022-10-22]. http://soot-oss.github.io/soot/

[40] Shivers O G. Control-flow analysis of higher-order languages [D]. Pittsburgh, PA: Carnegie Mellon University, 1991

[41] Might M, Smaragdakis Y, Horn D V. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis [C] //Proc of the 31st ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2010: 305−315

[42] Milanova A, Rountev A, Ryder B G. Parameterized object sensitivity for points-to and side-effect analyses for Java [C] //Proc of the 2002 Int Symp on Software Testing and Analysis. New York: ACM, 2002: 1−11

[43] Milanova A, Rountev A, Ryder B G. Parameterized object sensitivity for points-to analysis for Java[J]. ACM Transactions on Software Engineering and Methodology, 2002, 14(1): 1−41

[44] Smaragdakis Y, Bravenboer M, Lhoták O. Pick your contexts well: Understanding object-sensitivity [C] //Proc of the 38th Annual ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages. New York: ACM, 2011: 17−30

[45] Kastrinis G, Smaragdakis Y. Hybrid context-sensitivity for points-to analysis [C] //Proc of the 34th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2013: 423−434

[46] Tan Tian, Li Yue, Xue Jingling. Making k-object-sensitive pointer analysis more precise with still k-limiting [G] //LNCS 9837: Proc of the 2016 Int Static Analysis Symp. Berlin: Springer, 2016: 489−510

[47] Jeon M, Jeong S, Oh H. Precise and scalable points-to analysis via data-driven context tunneling [J]. Proceedings of the ACM on Programming Languages, 2018, 2(OOPSLA): 1−29

[48] Jeong S, Jeon M, Cha S, et al. Data-driven context-sensitivity for points-to analysis [J]. Proceedings of the ACM on Programming Languages, 2017, 1(OOPSLA): 1−28

[49] Jeon M, Oh H. Return of CFA: Call-site sensitivity can be superior to object sensitivity even for object-oriented programs [J]. Proceedings of the ACM on Programming Languages, 2022, 6(POPL): 1−29

[50] Smaragdakis Y, Kastrinis G, Balatsouras G. Introspective analysis: Context-sensitivity, across the board [C] //Proc of the 35th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2014: 485−495

[51] Li Yue, Tan Tian, Møller A, et al. Precision-guided context sensitivity for pointer analysis [J]. Proceedings of the ACM on Programming Languages, 2018, 2(OOPSLA): 1−29

[52] Lu Jingbo, Xue Jingling. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity [J]. Proceedings of the ACM on Programming Languages, 2019, 3(OOPSLA): 1−29

[53] Lu Jingbo, He Dongjie, Xue Jingling. Eagle: CFL-reachability-based precision-preserving acceleration of object-sensitive pointer analysis with partial context sensitivity [J]. ACM Transactions on Software Engineering and Methodology, 2021, 30(4): 1−46

[54] He Dongjie, Lu Jingbo, Gao Yaoqing, et al. Accelerating object-sensitive pointer analysis by exploiting object containment and reachability [C] //Proc of the 35th European Conf on Object-Oriented Programming. Wadern, Saarland: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2021, 16: 1−31

[55] Li Yue, Tan Tian, Møller A, et al. Scalability-first pointer analysis with self-tuning context-sensitivity [C] //Proc of the 2018 26th ACM Joint Meeting on European Software Engineering Conf and Symp on the Foundations of Software Engineering. New York: ACM, 2018: 129−140

[56] Tan Tian, Li Yue, Xue Jingling. Efficient and precise points-to analysis: Modeling the heap by merging equivalent automata [C] //Proc of the 38th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2017: 278−291

[57] Jeon M, Lee M, Oh H. Learning graph-based heuristics for pointer analysis without handcrafting application-specific features [J]. Proceedings of the ACM on Programming Languages, 2020, 4(OOPSLA): 1−30

[58] Kanvar V, Khedker U P. Heap Abstractions for Static Analysis [J]. ACM Computing Surveys, 2017, 49(2): 1−47

[59] Chen Yifan, Yang Chenyang, Zhang Xin, et al. Accelerating program

analyses in Datalog by merging library facts [G] //LNCS 12913: Proc of the 2021 Int Static Analysis Symp. Berlin: Springer, 2021: 77−101

[60] IBM T. J. Watson Research Center. WALA: T. J. Watson Libraries for Analysis [CP/OL]. [2022-10-22].https://github.com/wala/WALA

[61] Hopcroft J E, Karp R M. A linear algorithm for testing equivalence of finite automata [R]. Ithaca, NY: Cornell University, 1971

[62] Späth J, Do L N Q, Ali K, et al. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for Java [C] //Proc of the 30th European Conf on Object-Oriented Programming. Wadern, Saarland: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 22: 1−26

[63] Livshits B, Sridharan M, Smaragdakis Y, et al. In defense of soundiness: A manifesto[J]. Communications of the ACM, 2015, 58(2): 44−46

[64] Rastogi V, Chen Yan, Jiang Xuxian. DroidChameleon: Evaluating Android anti-malware against transformation attacks [C] //Proc of the 8th ACM SIGSAC Symp on Information, Computer and Communications Security. New York: ACM, 2013: 329−334

[65] Landman D, Serebrenik A, Vinju J J. Challenges for static analysis of Java reflection: Literature review and empirical study [C] //Proc of the IEEE/ACM 39th Int Conf on Software Engineering. Piscataway, NJ: IEEE, 2017: 507−518

[66] Barros P, Just R, Millstein S, et al. Static analysis of implicit control flow: Resolving Java reflection and Android intents (T) [C] //Proc of the 30th IEEE/ACM Int Conf on Automated Software Engineering. Piscataway, NJ: IEEE, 2015: 669−679

[67] Livshits B, Whaley J, Lam M S. Reflection analysis for Java [G] //LNCS 3780: Proc of the 2005 Asian Symp on Programming Languages and Systems. Berlin: Springer, 2005: 139−160

[68] Li Yue, Tan Tian, Sui Yulei, et al. Self-inferencing reflection resolution for Java [G] //LNCS 8586: Proc of the 2014 European Conf on Object-Oriented Programming. Berlin: Springer, 2014: 27−53

[69] Li Yue, Tan Tian, Xue Jingling. Effective soundness-guided reflection analysis [G] //LNCS 9291: Proc of the 2015 Int Static Analysis Symp. Berlin: Springer, 2015: 162−180

[70] Li Yue, Tan Tian, Xue Jingling. Understanding and analyzing Java reflection [J]. ACM Transactions on Software Engineering and Methodology, 2019, 28(2): 1−50

[71] Oracle America, Inc. . Java Native Interface Specification Contents [S/OL]. Austin, Texas: Oracle America, Inc., 2021 [2022-10-22]. https://docs.oracle.com/en/java/javase/17/docs/specs/jni/index.html

[72] Fourtounis G, Triantafyllou L, Smaragdakis Y. Identifying Java calls in native code via binary scanning [C] //Proc of the 29th ACM SIGSOFT Int Symp on Software Testing and Analysis. New York: ACM, 2020: 388−400

[73] Christakis M, Bird C. What developers want and need from program analysis: An empirical study [C] //Proc of the 31st IEEE/ACM Int Conf on Automated Software Engineering. New York: ACM, 2016: 332−343

[74] Bravenboer M, Smaragdakis Y. Exception analysis and points-to analysis: Better together [C] //Proc of the 18th Int Symp on Software Testing and Analysis. New York: ACM, 2009: 1−12

[75] Kastrinis G, Smaragdakis Y. Efficient and effective handling of exceptions in Java points-to analysis [G] //LNCS 7791: Proc of the 2013 Int Conf on Compiler Construction. Berlin: Springer, 2013: 41−60

[76] Fourtounis G, Smaragdakis Y. Deep static modeling of invoke dynamic [C] //Proc of the 33rd European Conf on Object-Oriented Programming. Wadern, Saarland: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019, 15: 1−28

[77] Heintze N, Tardieu O. Demand-driven pointer analysis [C] //Proc of the 2001 ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2001: 24−34

[78] Sridharan M, Gopan D, Shan L, et al. Demand-driven points-to analysis for Java [C] //Proc of the 20th Annual ACM SIGPLAN Conf on Object-Oriented Programming, Systems, Languages, and Applications. New York: ACM, 2005: 59−76

[79] Sridharan M, Bodík R. Refinement-based context-sensitive points-to analysis for Java [C] //Proc of the 27th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2006: 387−400

[80] Yan Dacong, Xu Guoqing, Rountev A. Demand-driven context-sensitive alias analysis for Java [C] //Proc of the 2011 Int Symp on Software Testing and Analysis. New York: ACM, 2011: 155−165

[81] Xu Guoqing, Rountev A, Sridharan M. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis [G] //LNCS 5653: Proc of the 2009 European Conf on Object-Oriented Programming. Berlin: Springer, 2009: 98−122

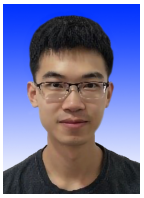
[82] Shang Lei, Xie Xinwei, Xue Jingling. On-demand dynamic summary-based points-to analysis [C] //Proc of the 10th Int Symp on Code Generation and Optimization. New York: ACM, 2012: 264−274

[83] Reps T, Horwitz S, Sagiv M. Precise interprocedural dataflow analysis via graph reachability [C] //Proc of the 22nd ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages. New York: ACM, 1995: 49−61

[84] Lu Yi, Shang Lei, Xie Xinwei, et al. An incremental points-to analysis with CFL-reachability [G] //LNCS 7791: Proc of the 2013 Int Conf on Compiler Construction. Berlin: Springer, 2013: 61−81

[85] Liu Bozhen, Huang J, Rauchwerger L. Rethinking incremental and parallel pointer analysis [J]. ACM Transactions on Programming Languages and Systems, 2019, 41(1): 1−31

[86] Liu Bozhen, Huang J. SHARP: Fast incremental context-sensitive pointer analysis for Java [J]. Proceedings of the ACM on Programming Languages, 2022, 6(OOPSLA1): 1−28



Tan Tian, born in 1991. PhD, associate researcher. Member of CCF. His main research interests include program analysis and programming languages.

谭　添, 1991 年生. 博士, 副研究员. CCF 会员. 主要研究方向为程序分析和程序设计语言.

Ma Xiaoxing, born in 1975. PhD, professor. Member of CCF. His main research interests include adaptive software systems, software architectures and middleware systems, and assurance of software qualities.

马晓星, 1975 年生. 博士, 教授. CCF 会员. 主要研究方向为自适应软件系统、软件架构与中间件系统、软件质量保障.

Xu Chang, born in 1977. PhD, professor. Senior Member of CCF. His main research interests include big data software engineering, intelligent software testing and analysis, and adaptive and autonomous software systems.

许　畅, 1977 年生. 博士, 教授. CCF 高级会员. 主要研究方向为大数据软件工程、智能软件测试与分析、自适应与自主软件系统.

Ma Chunyan, born in 1978. PhD, associate professor. Member of CCF. Her main research interests include embedded software system modeling and verification, software automatic testing and fault location.

马春燕, 1978 年生. 博士, 副教授. CCF 会员. 主要研究方向为嵌入式软件系统建模与验证、软件自动化测试与故障定位.

Li Yue, born in 1988. PhD, associate professor. Member of CCF. His main research interests include program analysis and programming languages.

李　樾, 1988 年生. 博士, 副教授. CCF 会员. 主要研究方向为程序分析与程序设计语言.