软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2022,33(1):78−101 [doi: 10.13328/j.cnki.jos.006345]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向对象程序的上下文敏感指针分析研究*

李昊峰[1,2], 孟海宁[1,2], 郑恒杰[1,2], 曹立庆[1,2], 李 炼[1,2]

[1](计算机体系结构国家重点实验室(中国科学院 计算技术研究所), 北京 100190)

[2](中国科学院大学, 北京 100049)

通信作者: 李炼, E-mail: lianli@ict.ac.cn

摘　要: 指针分析是编译优化、程序静态分析中的基础, 很多应用都需要基于指针分析, 低精度的指针分析会给这些应用带来高误报率和漏报率, 通过添加上下文敏感信息是提高指针分析的精度的一个重要手段. 自从面向对象的概念被提出来之后, 该概念得到了广泛的应用, Java、C++、.NET、C#等主流语言都支持面向对象的特性, 面向对象程序的指针分析越来越受关注. 做了一个系统文献综述(SLR), 通过对索引到的相关文献进行分析和归类, 总结了面向对象程序的上下文敏感指针分析研究的5个主要问题, 并对这5个问题中用到的方法进行了分析讨论.

关键词: 指针分析; 上下文敏感; 面向对象语言; 系统文献综述

中图法分类号: TP311

中文引用格式: 李昊峰, 孟海宁, 郑恒杰, 曹立庆, 李炼. 面向对象程序的上下文敏感指针分析研究. 软件学报, 2022, 33(1): 78–101. http://www.jos.org.cn/1000-9825/6345.htm

英文引用格式: Li HF, Meng HN, Zheng HJ, Cao LQ, Li L. Context-sensitive Pointer Analysis for Object-oriented Programs: A Systematic Literature Review. Ruan Jian Xue Bao/Journal of Software, 2022, 33(1): 78–101 (in Chinese). http://www.jos.org.cn/1000-9825/6345.htm

Context-sensitive Pointer Analysis for Object-oriented Programs: A Systematic Literature Review

LI Hao-Feng[1,2], MENG Hai-Ning[1,2], ZHENG Heng-Jie[1,2], CAO Li-Qing[1,2], LI Lian[1,2]

[1](State Key Laboratory of Computer Architecture (Institute of Computing Technology, Chinese Academy of Sciences), Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: Pointer analysis is the basis of compiler optimization and static analysis, and a lot of applications are based on pointer analysis. Low-precision pointer analysis brings high false positive rate and false negative rate to these applications, and adding context sensitive information is an important means to improve accuracy. Since the object-oriented concept was put forward, it has been widely used. Some mainstream languages, such as Java, C++, .NET and C#, support object-oriented features. Therefore, pointer analysis for object-oriented language is getting more and more attention. This study investigates context-sensitive pointer analysis for object-oriented language by using systematic literature review (SLR) method. After analyzing and categorizing the relevant literature, five questions are summarized about context-sensitive pointer analysis for object-oriented language.

Key words: pointer analysis; context-sensitive; object-oriented language; systematic literature review (SLR)

1 引　言

指针分析/别名分析是一种通过追踪对象在程序中的传播来计算变量的指向集或者变量间的别名关系的静态程序分析技术. 在静态分析中, 指针分析/别名分析是很多分析技术的基础, 如: 构建函数调用图(call graph)、逃逸分析(escape analysis)、副作用分析(side-effect analysis)等. 但是, 通过静态分析来确定运行时变量的指向集是不可判定的[1−4], 通过平衡精度和效率来求解近似解的方法在指针分析中得到广泛的应用.

* 基金项目: 国家自然科学基金(61872043)
收稿时间: 2019-11-27; 修改时间: 2020-09-12, 2021-02-24; 采用时间: 2021-04-07; jos在线出版时间: 2021-05-20

© 中国科学院软件研究所 http://www.jos.org.cn

自从 Smalltalk-80 出现以来, 面向对象技术逐渐成为了备受欢迎的技术, 面向对象语言使用继承、多态等机制对代码进行封装和重用, 从而模块化的构建应用. 由于面向对象语言的这些特性, 在处理面向对象程序的指针分析的时候, 添加上下文信息会对精度带来很大提升.

为了系统地分析面向对象程序的上下文敏感指针分析, 本文采用系统文献综述 (SLR) 的方法. 我们对索引到的所有相关的文献进行系统的分类, 然后逐个类别针对不同的技术进行分析.

本文第 2 节做了背景介绍, 第 3 节介绍系统文献综述 (SLR) 具体步骤, 第 4 节介绍对文献的分类与分析.

2 背景介绍

在指针分析中以下几个是主要关心的问题: 流敏感、上下文敏感、域敏感、堆模型、调用图构建方式 (在预处理中使用低精度构建函数调用图还是指针分析过程中构建函数调用图)、别名关系表示 (显示表示变量间别名关系还是使用指向集隐式表示)、集合模型 (是否区分集合中元素)、引用表示 (引用表示是否全局唯一)、对象流向 (基于合并 (unification-based) 还是基于包含 (inclusion-based))[5,6]. 基于合并 (unification-based) 的 Steensgaard 算法[7]和基于包含 (inclusion-based) 的 Andersen 算法[8]是两个经典的流不敏感上下文不敏感的指针分析算法. 其中, Andersen 算法是流不敏感上下文不敏感指针分析中精度最高的算法, 已经有很多工作在 Java 上实现了 Andersen 指针分析[9−12], 但是, 对一些基于指针分析的应用, 如: 污点分析、虚函数解析、类型安全转换等, Andersen 指针分析的精度不能满足需求, 添加上下文敏感可以带来很大的精度提升. 如图 1 所示, (a) 是源程序, 这里 id 函数简单地返回参数, 不做其他处理. 如果上下文不敏感, p 在整个分析过程中只有一份, 且指向{A, B}, 这样 $v1$ 和 $v2$ 就会出现不精确的指向集 ((b) 中加粗标出), 这时指针分析会产生误报, 误认为这里的类型转换不安全, 在上下文敏感的情况下则可以消除这种误报, 这里拿调用点敏感 (1-call-site-sensitive) 举例, 如 (c) 所示, 在整个分析过程中 p 根据不同的调用点会有两份, 在 L1 上下文下 p 会指向 A, 在 L2 上下文下 p 会指向 B, 这样 $v1$ 和 $v2$ 的指向集就不会出现不精确问题, 从而在 L3 和 L4 的类型转换就不会产生误报.

源程序	上下文不敏感	调用点敏感 (1-call-site-sensitive)

public void foo(){
A *o1* = new A();//A
B *o2* = new B();//B
Object *v1* = id(*o1*);//L1
Object *v2* = id(*o2*);//L2
A *a* = (A)*v1*;//L3
A *b* = (B)v2;//L4
}
public Object id(Object *p*){
return *p*;
}
	o1 ⟶ {A}
	o2 ⟶ {B}
	p ⟶ {A,B}
	v1 ⟶ {A,**B**}
	v2 ⟶ {**A**,B}
		<[], *o1*> ⟶ {A}
		<[], *o2*> ⟶ {B}
		<[L1], *p*> ⟶ {A}
		<[L2], *p*> ⟶ {B}
		<[], *v1*> ⟶ {A}
		<[], *v2*> ⟶ {B}

	L3: cast may fail L4: cast may fail	L3: cast safe L4: cast safe
(a)	(b)	(c)

图 1 上下文敏感对 Cast Safety 应用带来的精度提升

一些研究工作[10,13−15]表明, 在面向对象程序的指针分析中添加流敏感信息对精度提升有限, 而且代价会很大, 添加上下文信息带来的收益更大.

3 相关文献索引

本文采用系统文献综述方法[16−18]全面地分析面向对象程序的上下文敏感指针分析. 通过对表 1 中的关键字进行组合, 在 5 个主流数据库 (ACM Digital Library、IEEE Xplore Digital Library、SpringerLink、Web of Knowledge、ScienceDirect, 搜索时间为 2019-09-27) 中进行搜索, 这些数据库囊括了程序语言、形式化、软件工程等方向的顶级会议和期刊, 如: PLDI、OOPSLA、POPL、ASPLOS、ECOOP、SAS、CGO、FSE、ICSE 等. 然后人工筛选并通过引文进行补充, 最后得到 85 篇相关文献.

© 中国科学院软件研究所 http://www.jos.org.cn

表 1 搜索关键字

行号	关键字
1	Java、“object-oriented languages”
2	Points-to、“pointer analysis”“alias analysis”
3	context-sensitive

4 分 析

本节会通过阅读和分析索引到的文献, 对面向对象程序的上下文敏感指针分析所面临的问题和使用到的技术进行探讨.

4.1 上下文敏感指针分析主要研究的问题

本文对收集到的文献进行了分类, 将面向对象程序的上下文敏感指针分析研究的问题分成以下几类.

① 上下文敏感指针分析中上下文的表示方法

指针分析是一类特殊的数据流问题[19], 在进行过程间数据流分析时主要有两种方法维护上下文敏感[20], 一种是基于摘要的方式, 通过对函数的 input-output 关系做摘要来实现上下文敏感, 另一种是基于标签的方式, 通过给函数和变量贴上上下文敏感信息的标签来区分上下文.

② 上下文敏感指针分析的实现方法

本文对上下文敏感指针分析的实现方法做了统计和归类, 主要有两种实现方式 (这里只描述了基于包含的算法, 对于基于合并的算法只需要在计算指向集约束的时候采取合并的方式).

第 1 种是类 Andersen 的实现方法. 通过把每条指令表示成指向集中的对象的流向, 通过迭代的方式来计算一个闭包.

第 2 种是上下文无关语言图可达 (CFL-reachability). 通过计算某个对象到某个变量是否在图中可达, 并且满足上下文无关语言, 从而得出该变量是否指向该对象.

③ 上下文敏感指针分析的优化

上下文敏感指针分析在带来精度提升的同时通常会带来效率下降的问题. 为了更好地平衡精度和效率, 有很多优化方法被提出, 如: 对数据结构进行优化、部分的使用上下文敏感、按需计算指针分析 (demand-driven)、针对特定应用 (client-driven) 计算指针分析、增量分析等.

④ 别名分析

有些应用在分析过程中不需要得到某个变量具体指向集, 只需要得到两个变量的别名关系, 在面向对象程序的别名分析过程中, 通常使用 AccessPath 来表示变量.

⑤ 上下文敏感指针分析的评估指标

上下文敏感指针分析的效率很好评估, 用时间和内存可以很好地衡量. 对于精度的评估则没有统一的标准, 通过一些基于指针分析的应用的精度来衡量指针分析的精度是常用的一种方式.

4.2 上下文敏感指针分析研究各个问题所使用的方法

第 4.1 节已经对上下文敏感指针分析研究的问题进行了分类, 对于每个类别都有很多工作通过使用各种不同的方法来解决这些问题, 本小节对各个分类中所涉及的方法进行描述总结.

4.2.1 上下文敏感指针分析中上下文的表示方法

不同的上下文表示方法适应的场景不同, 不同的上下文表示方法对精度和效率的影响也不同, 本小节对基于标签和基于摘要两种上下文表示策略进行描述.

4.2.1.1 基于标签的方式

由于递归调用的存在, 基于标签的方式表示上下文的数量是无限的, 常用 k-limit 技术来处理上下文数量无限

© 中国科学院软件研究所 http://www.jos.org.cn

的问题, 即: 对每个上下文的标签数量做限制, 使用最近的 k 个标签来表示上下文.

基于标签的方式是通过标签的内容来区分上下文, 根据标签的内容类型的不同会有不同的上下文表示方式. 这里对标签的内容类型进行分类.

① 调用点敏感 (k-call-site-sensitive, kCFA)

一个函数可能对应多个不同的调用点, 在不同的调用点, 该函数的上下文信息是不同的. 调用点敏感是最常见的一种基于标签的上下文敏感指针分析的方式. 调用点敏感指针分析会使用前 k 个调用点作为上下文信息[21,22], 随着 k 的取值越大, 精度会有所提升, 通常效率会下降.

如图 2 所示, 这里对 k=1 的情况进行描述. (a) 是源代码, 首先使用 new 语句初始化两个对象 O1 和 O2, 分别赋值给变量 $o1$ 和 $o2$, 然后分别作为实参传入 id 函数, id 函数只是简单地返回形参, 将 id 函数的返回值分别赋值给 $v1$ 和 $v2$. (b) 是上下文不敏感下的指向关系, →左边是变量, 右边是对应的指向集, 这里使用 new 语句所在的位置表示对象 O1 和 O2, 这种表示在后面通用. 当上下文不敏感的时候 p 在分析过程中只有一份, $o1$ 和 $o2$ 的指向集会传递给形参 p 进行合并, 然后返回给 $v1$ 和 $v2$, 这样 $v1$ 和 $v2$ 的指向集就会出现冗余. 如 (c) 是调用点敏感指针分析, 当使用调用点敏感的时候, id 函数有两个调用点 L1 和 L2, p 在分析过程中使用调用点 L1 和 L2 来区分, $o1$ 的指向集传给上下文 L1 下的 p, 最后通过返回值传给 $v1$, $o2$ 的指向集会传给上下文 L2 下的 p, 最后通过返回值传给 $v2$, 这样 $v1$ 和 $v2$ 就有不存在冗余指向关系, 从而带来了精度提升. 当 k=2 的时候使用当前函数的调用点和该调用点所在的函数的调用点共同表示上下文信息. 当 k>2 的时候以此类推.

源程序	上下文不敏感	调用点敏感 (1-call-site-sensitive)

public void foo(){
Object *o*1 = new Object();//O1
Object *o*2 = new Object();//O2
Object *v*1 = id(*o*1);//L1
Object *v*2 = id(*o*2);//L2
}
public Object id(Object *p*){
return *p*;
}
(a)
	*o*1 ⟶ {O1}
	*o*2 ⟶ {O2}
	p ⟶ {O1,O2}
	*v*1 ⟶ {O1,O2}
	*v*2 ⟶ {O1,O2}
	(b)
		<[], *o*1> ⟶ {O1}
		<[], *o*2> ⟶ {O2}
		<[L1], *p*> ⟶ {O1}
		<[L2], *p*> ⟶ {O2}
		<[], *v*1> ⟶ {O1}
		<[], *v*2> ⟶ {O2}
		(c)

图 2　调用点敏感

② 对象敏感 (k-object-sensitive)

对象敏感是面向对象程序上下文敏感指针分析特有的上下文表示方式, 使用前 k 个函数的接收对象 (receiver object) 作为上下文信息[23,24], 随着 k 的取值越大, 往往精度会有所提升, 效率有所下降. 如图 3(a), 首先实例化一个对象 A1 赋值给 $a1$, 然后把 $a1$ 赋值给 $a2$, 然后实例化一个对象 A2 赋值给 $a2$, 接着实例化一个对象 A3 赋值给 $a3$, 然后实例化对象 O1、O2、O3 分别赋值给 $o1$、$o2$、$o3$, 然后分别作为参数传给 $a1$、$a2$、$a3$ 调用的函数 id, 将返回结果分别赋值给 $v1$、$v2$、$v3$. id 函数只是简单地返回参数. 图 3(b)–图 3(f) 省略了部分变量的指向集, 这些变量在不同的上下文下指向集一样, $a1$→{A1}, $a2$→{A1, A2}, $a3$→{A3}, $o1$→{O1}, $o2$→{O2}, $o3$→{O3}. 图 3(d) 是 k=1 的情况下的对象敏感的指向关系. 这里使用 id 函数的调用点的接收对象, 也就是 A1、A2、A3 作为上下文信息, 如图 3(d) 所示, 变量 p 在 3 个上下文下指向集不同, 从而 $v1$、$v2$、$v3$ 的指向集相比较上下文不敏感精度有所提升. 当 k>1 的时候如何表示上下文出现了分歧, 不同的实现表示方式不同, 文献 [25] 根据不同的实现定义了两种类型 Full-Object-Sensitivity 和 Plain-Object-Sensitivity, 其中, Full-Object-Sensitivity 是文献 [23,24] 所定义的对象敏感, 当 k=2 时, 该方法把某个函数的接收对象和该对象所在的函数的接收对象共同作为上下文信息, 当 k>2 的时候以此类推. Plain-Object-Sensitivity 是工具 Paddle[26,27]所实现的对象敏感, 当 k=2 时, 该方法把某个函数的接收对象和这个函数的调用函数 (caller) 的接收对象作为上下文信息, 当 k>2 时以此类推. 如图 4 所示是两种方式的对

© 中国科学院软件研究所 http://www.jos.org.cn

比, 如图 4(a) 所示, test 函数中实例化两个对象 O1 和 O2 然后分别赋值给 *o*1 和 *o*2, 然后作为参数依次传给 foo、bar、id, id 只是简单的对参数 *p*3 进行返回. 对于 Full-Object-Sensitivity 方法, *p*3 所在的函数 id 的接收对象是 B, B 是在函数 foo 中实例化的, foo 有两个接收函数, 分别是 T1 和 T2, 因此 *p*3 有两个上下文 [T1, B] 和 [T2, B], 具体的指向关系如图 4(b) 所示. 对于 Plain-Object-Sensitivity 方法, *p*3 所在的函数 id 的接收对象是 B, id 的调用函数 (caller) 是 bar, bar 的接收对象只有 A, 因此 *p*3 只有一个上下文 [A, B], 具体指向关系如图 4(c) 所示. 当使用相同的 *k* 的时候, Full-Object-Sensitivity 的精度往往会比 Plain-Object-Sensitivity 的精度更高, 图 4 是其中一个例子.

<table>
<tr><th>源程序</th><th>上下文不敏感</th><th>调用点敏感
(1-call-site-sensitive)</th></tr>
<tr><td rowspan="5">public void foo(String[] args){
 A a1 = new A();//A1
 A a2 = a1;
 if (args.length() > 0)
  a2 = new A();//A2
 A a3 = new A();//A3
 Object o1 = new Object();//O1
 v1 = a1.id(o1);//L1
 Object o2 = new Object();//O2
 v2 = a2.id(o2);//L2
 Object o3 = new Object();//O3
 v3 = a3.id(o3);//L3
}
Class A{
 public Object id(Object p){
  return p;
 }
}
(a)</td><td>p → {O1,O2,O3}
v1 → {O1,O2,O3}
v2 → {O1,O2,O3}
v3 → {O1,O2,O3}
(b)</td><td><[L1], p> → {O1}
<[L2], p> → {O2}
<[L3], p> → {O3}
<[], v1> → {O1}
<[], v2> → {O2}
<[], v3> → {O3}
(c)</td></tr>
<tr><td>对象敏感
(1-object-sensitive)</td><td>This 敏感
(1-this-sensitive)</td></tr>
<tr><td><[A1], p> → {O1,O2}
<[A2], p> → {O2}
<[A3], p> → {O3}
<[], v1> → {O1,O2}
<[], v2> → {O1,O2}
<[], v3> → {O3}
(d)</td><td><[A1], p> → {O1}
<[A1,A2], p> → {O2}
<[A3], p> → {O3}
<[], v1> → {O1}
<[], v2> → {O2}
<[], v3> → {O3}
(e)</td></tr>
<tr><td colspan="2">类型敏感 (1-type-sensitive)</td></tr>
<tr><td colspan="2"><[A], p> → {O1,O2,O3} <[], v1> → {O1,O2,O3}
<[], v2> → {O1,O2,O3} <[], v3> → {O1,O2,O3}
(f)</td></tr>
</table>

图 3 多种上下文表示方式

③ 类型敏感 (k-type-sensitive)

类型敏感和对象敏感有些类似, 类型敏感是把某函数的接收对象的类型作为上下文[25], 如图 3(a) 所示, id 函数有 3 个接收对象 A1、A2、A3, 他们类型都是 A, *p* 的上下文只有 A. 具体指向关系如图 3(f) 所示. 从定义可以看出, 对象敏感比类型敏感更加细粒度, 因此对象敏感总是比类型敏感精度要高. 在实验中显示[25], 类型敏感可以在接近对象敏感精度的同时可以有很好的可扩展性 (scalability).

④ this 敏感 (k-this-sensitive)

this 敏感和对象敏感也有些类似, 对象敏感使用函数的接收对象作为上下文信息, 而 this 敏感使用函数的接收变量的指向集作为上下文信息[28], 如图 3(a) 所示, id 函数有 3 个接收变量, 分别是 *a*1、*a*2、*a*3, id 的上下文就是这 3 个变量的指向集, 分别是: [A1], [A1, A2], [A3]. 图 3(e) 给出了具体指向集信息.

调用点敏感、对象敏感、this 敏感的精度孰高孰低理论上无法比较. 对象敏感比 this 敏感的上下文敏感信息更加细粒度, 很容易找到对象敏感比 this 敏感更精确的例子, 但是如图 3 所示, this 敏感又比对象敏感更加精确. 对于一些应用来说调用点敏感会比对象敏感的精度更高[29], 在图 3 示例中也证明有些情况下调用点敏感比对象敏感更加精确, 但是在图 5 中的例子中, L3 所在的调用点会对 L1 和 L2 上下文下的 *p*1 的指向集做合并操作, 从而带来精度损失, 该示例中对象敏感比调用点敏感更精确. 在真实例子测试中表明, 对象敏感和 this 敏感往往比调用点敏感精度和效率都会更高[30−32]. 文献 [32] 使用 4 个指标对对象敏感和 this 敏感在真实测试集中进行测试, 结果显示在其中 3 个指标中两种上下文敏感算法精度相差无几, this 敏感的效率会更高, 其中有 1 个指标 this 敏感的精度会低很多.

© 中国科学院软件研究所 http://www.jos.org.cn

源程序	2-Full-Object-Sensitivity	2-Plain-Object-Sensitivity

Class Test{
  public void test(){
    Test *t*1 = new Test()//T1
    Test *t*2 = new Test()//T2
    Object *o*1 = new Object();//O1
    Object *o*2 = new Object();//O2
    Object *v*1 = *t*1.foo(*o*1);//L1
    Object *v*2 = *t*2.foo(*o*2);//L2
  }
  public Object foo(Object *p*1){
    A *a* = new A();//A
    B *b* = new B();//B
    return *a*.bar(*b*, *p*1);//L3
  }
}
class A{
  public Object bar(B *pb*,Object *p*2){
    return pb.id(*p*2);//L4
  }
}
class B{
  public Object id(Object *p*3){
    return *p*3;
  }
}
(a)
	<[,], *t*1>⟶{T1}
	<[,], *t*2>⟶{T2}
	<[,], *o*1>⟶{O1}
	<[,], *o*2>⟶{O2}
	<[,T1], *p*1>⟶{O1}
	<[,T2], *p*1>⟶{O2}
	<[,T1], *a*>⟶{A}
	<[,T2], *a*>⟶{A}
	<[,T1], *b*>⟶{B}
	<[,T2], *b*>⟶{B}
	<[T1,A], *p*2>⟶{O1}
	<[T2,A], *p*2>⟶{O2}
	<[T1,B], *p*3>⟶{O1}
	<[T2,B], *p*3>⟶{O2}
	*v*1⟶{O1}
	*v*2⟶{O2}
	(b)
		<[,], *t*1>⟶{T1}
		<[,], *t*2>⟶{T2}
		<[,], *o*1>⟶{O1}
		<[,], *o*2>⟶{O2}
		<[,T1], *p*1>⟶{O1}
		<[,T2], *p*1>⟶{O2}
		<[,T1], *a*>⟶{A}
		<[,T2], *a*>⟶{A}
		<[,T1], *b*>⟶{B}
		<[,T2], *b*>⟶{B}
		<[T1,A], *p*2>⟶{O1}
		<[T2,A], *p*2>⟶{O2}
		<[A,B], *p*3>⟶{O1,O2}
		*v*1⟶{O1,O2}
		*v*2⟶{O1,O2}
		(c)

图 4　Full-Object-Sensitivity vs. Plain-Object-Sensitivity

源程序	对象敏感 (1-object-sensitive)	调用点敏感 (1-call-site-sensitive)

public void foo(){
  A *a*1 = new A();//A1
  A *a*2 = new A();//A2
  Object *o*1 = new Object();//O1
  Object *o*2 = new Object();//O2
  Object *v*1 = *a*1.id(*o*1);//L1
  Object *v*2 = *a*2.id(*o*2);//L2
}
Class A{
  public Object id(Object *p*1){
    return id2(*p*1);//L3
  }
  public Object id2(Object *p*2){
    return *p*2;
  }
}
(a)
	<[], *a*1>⟶{A1}
	<[], *a*2>⟶{A2}
	<[], *o*1>⟶{O1}
	<[], *o*2>⟶{O2}
	<[A1], *p*1>⟶{O1}
	<[A2], *p*1>⟶{O2}
	<[A1], *p*2>⟶{O1}
	<[A2], *p*2>⟶{O2}
	<[], *v*1>⟶{O1}
	<[], *v*2>⟶{O2}
	(b)
		<[], *a*1>⟶{A1}
		<[], *a*2>⟶{A2}
		<[], *o*1>⟶{O1}
		<[], *o*2>⟶{O2}
		<[L1], *p*1>⟶{O1}
		<[L2], *p*1>⟶{O2}
		<[L3], *p*2>⟶{O1,O2}
		<[], *v*1>⟶{O1,O2}
		<[], *v*2>⟶{O1,O2}
		(c)

图 5　对象敏感 vs.调用点敏感

上面是 4 种常用的上下文信息表示方法, 除了通过类型的不同进行归类, 还可以使用上下文信息选取的范围进行归类. 上面 4 种只是使用了函数调用点和函数的接收变量的相关信息来定义上下文, 函数的参数的信息也可以描述上下文信息.

图 6 是参数作为上下文信息的示例程序, 类 B_child1 和类 B_child2 继承自类 B, 在 main 函数中, *a* 的指向集是{A1, A2}, 类型都是 A, *b* 的指向集是{B1, B2}, 一个类型是 B_child1, 另一个类型是 B_child2, *c* 的指向集是{C1, C2}, 类型都是 C.

① this 敏感和参数指向集结合 (k-thisArgs-sensitive)

该方法使用前 *k* 个调用点的实参的指向集 (包含接收变量的指向集) 作为上下文信息[32]. 对于图 6 中的示例, 当 *k*=1 时, 第 12 行 foo 函数的上下文用 *a* 的指向集、*b* 的指向集和 *c* 的指向集共同表示, 即: [{A1, A2}, {B1, B2}, {C1, C2}].

© 中国科学院软件研究所 http://www.jos.org.cn

```
1. public void main(String[] args){          14. Class A{
2.   A a; B b; C c;                           15.   public void foo(B p1, C p2){}
3.   if (args.length() > 0){                  16. }
4.     a = new A();//A1                       17. Class B{}
5.     b = new B_child1();//B1                18. Class B_child1 extends B{}
6.     c = new C();//C1                       19. Class B_child2 extends B{}
7.   }else{                                   20. Class C{}
8.     a = new A();//A2
9.     b = new B_child2();//B2
10.    c = new C();//C2
11.  }
12.  a.foo(b, c);
13.}
```

图 6 参数信息作为上下文信息示例程序

② 对象敏感和参数指向集结合 (k-objectArgs-sensitive)

该方法使用前 *k* 个函数的接收对象和实参的指向集 (不包含接收变量) 共同作为上下文信息[32]. 对于图 6 中的示例, 当 *k*=1 时, 第 12 行 foo 函数的上下文用 foo 函数的接收对象、参数 *b* 的指向集和 *c* 的指向集共同表示, foo 函数的接收对象有两个: A1 和 A2, foo 函数的上下文有两个, 即: [A1, {B1, B2}, {C1, C2}] 和 [A2, {B1, B2}, {C1, C2}].

③ 参数指向集笛卡尔乘积 (value-context-sensitive)

该方法使用前 *k* 个调用点的实参的指向集 (包含接收变量的指向集) 的笛卡尔积作为上下文信息[33]. 对于图 6 中的示例, 当 *k*=1 时, 第 12 行 foo 函数的上下文是 *a* 的指向集、*b* 的指向集和 *c* 的指向集的笛卡尔乘积表示, 这里 foo 函数有 8 个上下文, 即: [A1, B1, C1]、[A1, B1, C2]、[A1, B2, C1]、[A1, B2, C2]、[A2, B1, C1]、[A2, B1, C2]、[A2, B2, C1]、[A1, B2, C2].

④ 参数指向集的类型笛卡尔乘积 (k-typeArgs-sensitive)

该方法和上面③中的方法类似, 该方法使用前 *k* 个调用点的实参的指向集 (包含接收变量的指向集) 的类型的笛卡尔积作为上下文信息[34]. 对于图 6 中的示例, 当 *k*=1 时, 第 12 行 foo 函数的上下文是 *a* 的指向集的类型、*b* 的指向集的类型和 *c* 的指向集的类型的笛卡尔乘积, 这里 foo 函数有两个上下文, 即: [A, B_child1, C] 和 [A, B_child2, C].

文献 [32] 指出对于 k-thisArgs-sensitive 和 k-objectArgs-sensitive 来说, 分别和 this 敏感和对象敏感相比较, 当 *k* 保持一样时, 带来的精度提升有限.

不同类型的上下文敏感对于上下文的表示方式不同, 几种不同的上下文表示方法可以一起使用, 共同表示某个上下文, 也可以对静态函数和非静态函数使用不同的上下文表示方法. 文献 [32,35] 提出了几种上下文结合使用的方法.

① this 敏感+调用点敏感 (m-this+nCFA)

把前 *m* 个函数的接收变量的指向集和前 *n* 个函数的调用点共同作为函数的上下文信息[32].

② 对象敏感+调用点敏感 (m-object+nCFA 或 U-m-object+nCFA)

把前 *m* 个函数的接收对象和前 *n* 个函数的调用点共同作为函数的上下文信息[32,35].

③ 类型敏感+调用点敏感 (U-m-type+nCFA)

把前 *m* 个函数的接收对象的类型和前 *n* 个函数的调用点共同作为函数的上下文信息[35].

④ Selective k-object-sensitive hybrid A

对于静态函数使用函数的调用点作为上下文信息, 对于非静态函数使用前 *k* 个函数的接收对象作为上下文信息[35].

⑤ Selective k-object-sensitive hybrid B

对于非静态函数使用前 *k* 个函数的接收对象作为上下文信息, 对于静态函数使用前 *k* 个函数的接收对象 (如

© 中国科学院软件研究所 http://www.jos.org.cn

果是静态函数就找 caller 的接收对象, 直到找到 k 个或者找到了入口函数) 和函数的调用点共同作为上下文信息[35].

⑥ Selective k-type-sensitive

对于非静态函数使用前 k 个函数的接收对象的类型作为上下文信息, 对于静态函数使用前 k 个函数的接收对象的类型 (如果是静态函数就找 caller 的接收对象的类型, 直到找到 k 个或者找到了入口函数) 和函数的调用点共同作为上下文信息[35].

和 this 敏感相比, this 敏感和调用点敏感结合使用带来的精度提升很小[32]. 对于所有的函数都使用对象敏感+调用点敏感相比较对象敏感来说精度会有所提升, 但是时间和内存都会成倍数增长, 使用类型敏感+调用点敏感也是如此[35]. 和 1-object-sensitive 相比, SA-1-obj (selective 1-object-sensitive hybrid A) 会更快, 精度相近, SB-1-obj (selective 1-object-sensitive hybrid B) 的精度总是更高, 开销也只是稍微有所增加. 和 2-type-sensitive 相比, S-2-type (selective 2-type-sensitive) 精度会高一些, 时间消耗相差无几.

上面描述了一些对于函数或变量使用上下文敏感来提高精度, 对于堆 (heap) 也可以使用上下文敏感[35]. 图 7(a) 是堆敏感示例, 当堆上下文不敏感的时候 B1 不存在上下文, 这时候通过 B1 访问域 g 就会发生合并, <B1>. g 指向 {O1, O2}, 最后 $v1$ 和 $v2$ 的指向集就会出现精度损失, 图 7(b) 是堆上下文不敏感时变量的指向集. 当堆上下文敏感的时候, 这里设置堆上下文深度为 1, B1 就会有两个堆上下文 [A1] 和 [A2], 在上下文 A1 下<[A1], B1>. g 指向 {O1}, 在上下文 A2 下<[A2], B1>. g 指向 {O2}, 最后 $v1$ 和 $v2$ 的指向集不会出现冗余, 图 7(c) 是堆上下文敏感时的指向关系.

源程序

```
public void foo(){
  A a1 = new A();//A1
  A a2 = new A();//A2
  Object o1 = new Object();//O1
  Object o2 = new Object();//O2
  a1.set(o1);
  a2.set(o2);
  Object v1 = a1.get();
  Object v2 = a2.get();
}
```

```
class A{
  B f = new B();//B1
  public void set(Object p){
    f.g = p;
  }
  public Object get(){
    Object ret = f.g;
    return ret;
  }
}
class B{ public Object g; }
```

(a)

对象敏感堆不敏感 (1-object-sensitive)

```
<[], a1>──→{A1}
<[], a2>──→{A2}
<[], o1>──→{O1}
<[], o2>──→{O2}
<[A1], p>──→{O1}
<[A2], p>──→{O2}
<<B1>.g>──→{O1,O2}
<[A1], ret>──→{O1,O2}
<[A2], ret>──→{O1,O2}
<[], v1>──→{O1,O2}
<[], v2>──→{O1,O2}
```

(b)

对象敏感堆敏感 (1-object-sensitive+1 -heap-sensitive)

```
<[], a1>──→{<[],A1>}
<[], a2>──→{<[],A2>}
<[], o1>──→{<[],O1>}
<[], o2>──→{<[],O2>}
<[A1], p>──→{<[],O1>}
<[A2], p>──→{<[],O2>}
<<[A1],B1>.g>──→{O1}
<<[A2],B1>.g>──→{O2}
<[A1], ret>──→{O1}
<[A2], ret>──→{O2}
<[], v1>──→{O1}
<[], v2>──→{O2}
```

(c)

图 7 堆敏感

堆上下文敏感可以和上面提到的任意类型的上下文敏感结合使用. 对于对象敏感和 this 敏感来说, 相比较 k=1, 增加 k 带来的效果提升不是很大[31,32], 反而通过添加堆的上下文会带来更大的收益[31].

4.2.1.2 基于摘要的方式

第 4.2.1.1 节描述的指针分析算法都是自顶向下 (top-down) 的算法, 基于摘要的方式实现上下文敏感指针分析通常使用自底向上 (bottom-up)[36−38]的方式. 通过预处理构建函数调用图, 然后去除环 (将环映射成一个节点, 当分析到环节点的时候再映射回来, 分析环中的每个节点), 然后从图中的叶子节点开始, 对该函数做过程内分析, 并

© 中国科学院软件研究所 http://www.jos.org.cn

对参数 (对于非静态函数, 参数包括 this 变量) 做摘要, 根据摘要信息的不同会有不同的实现方式.

如图 8 是一个简单的代码片段, 这里对函数 id 做摘要信息可以得到 id 的返回值的指向集就是参数 *p* 的指向集, 第 3 行实参 *o*1 指向集是{O1}, 根据摘要信息, *v*1 指向集是 *o*1 的指向集, 即 *v*1 指向集是{O1}, 同理, *v*2 的指向集是{O2}, 这样通摘要的方法对函数 id 实现了上下文敏感.

```
1. main() {
2.   Object o1 = new Object();\\O1
3.   v1 = id(o1);
4.   Object o2 = new Object();\\O2
5.   v2 = id(o2);
6. }
7. id(p) {
8.   return p;
9. }
```

图 8　摘要方式实现上下文敏感

① 指向关系做摘要信息

通过过程内分析可以计算出形参和返回值的指向关系, 把形参和返回值的指向关系作为摘要信息, 然后从预处理的函数调用图的叶子节点开始自底向上遍历, 对函数进行过程间处理, 当遇到调用语句的时候把实参和形参进行替换, 从而实现上下文敏感. 如图 9 所示, (a) 是源代码, 在 foo 函数中实例化对象 A 并赋值给变量 *a*, 然后调用 bar 函数, 在 bar 函数中将参数赋值给接收对象的域 *g*, 接着 foo 函数中会调用 id 函数, 并将返回值赋给接收对象的域 *f*, id 函数只是简单地返回参数. (b) 是以指向关系作为摘要信息, bar 函数的摘要中的 *this*.ε 表示 bar 函数的接收对象, 该接收对象的域 *g* 指向参数 *p*2 的指向的对象, 这里用 *p*2.ε 表示. id 函数的摘要中, *ret* 表示 id 函数的返回变量, *ret* 指向参数 *p*3 的指向对象, 这里用 *p*3.ε 表示. 函数 foo 中, 根据函数 id 的摘要可以得到 *this*.*f* 指向参数 *p*1 指向的对象, 因此函数 foo 的摘要是 foo 函数的接收对象的域 *f* 指向参数 *p*1 的指向对象. 为了提高精度, 文献 [39] 在摘要的指向关系中添加类型信息.

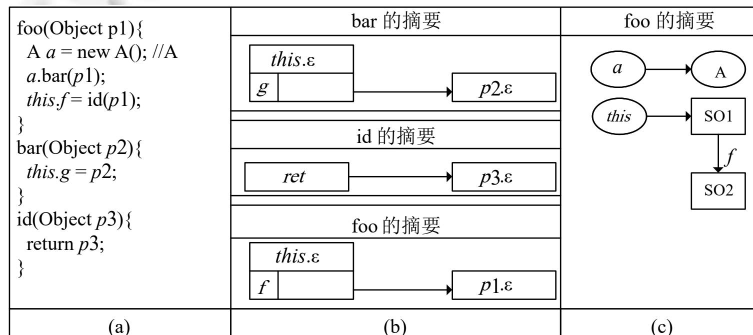

图 9　基于摘要的上下文表示

② 指向图作为摘要信息

除了使用指向关系作为摘要信息还可以使用指向图作为摘要信息, 文献 [40] 对每个函数进行过程内分析, 构建符号指向图 (SPG), SPG 是对标准指向图 (points-to graph) 的一个扩展, 对函数内不可见的对象符号化, 然后从预处理的函数调用图的叶子节点开始自底向上遍历, 对函数进行过程间处理, 当遇到调用语句的时候把对应的 SPG 摘要进行克隆, 并且把实参和对应的符号对象做替换, 从而维护上下文敏感. 图 9(c) 是 foo 函数以 SPG 作为摘要信息的示例, 椭圆表示程序中的变量或者函数中实例化的对象, 长方形表示符号对象, 变量 *a* 指向对象 A, *this* 指向 foo 函数的接收对象, 这里用 SO1 表示, SO1 的域 *f* 会指向函数 id 的返回值, 这里用 SO2 表示, 在使用摘要信息的时候会将对应的符号对象替换成真实的对象.

指向关系作为摘要和指向图作为摘要是不同的, 这里以 foo 函数的摘要信息为例. 以指向关系作为摘要, 图 9(b) 记录了 *this*.*f* 指向 *p*1 的指向集. 以指向图作为摘要, 对于那些在当前函数未知的对象都生成一个符号对象, 如

SO1 和 SO2, 图 9(c) 记录了 *this* 指向一个 SO1 对象, SO1 的域 *f* 指向对象 SO2.

基于摘要的方式除了自底向上 (bottom-up) 的方式外还可以使用自顶向下 (top-down) 的方式, 文献 [41,42] 通过过程内分析对每个函数做指向图 (points-to graph) 摘要, 然后从入口函数开始自顶向下 (top-down) 处理, 构建完整的指向图, 然后在指向图上求解闭包.

文献 [43] 对基于标签的调用点敏感和基于摘要两种上下文敏感指针分析的方法进行理论上和真实例子测试上做了对比, 对于有限的可分配的问题来说, 基于摘要的精度近似 *k* 取无穷大情况下的调用点敏感的精度, 同时基于摘要的方式的扩展性也会更好.

4.2.1.3 对比分析

自顶向下和自底向上两种构建指针分析的方式各有优势, 两种方式没有某一种有绝对的优势, 只是应对不同的应用场景有优势.

自底向上需要预先构造一个函数调图, 这就带来了额外的开销, 同时, 调用图的精度对后续的分析精度有一定的影响. 当摘要是基于 store-based 模型[44]的时候, 如果具体指向的对象未知, 会使用符号对象临时替代, 直到在处理 caller 的时候找到具体指向的对象, 才会将具体对象替换符号对象, 因此, 在这种情况下, 只有对整个程序全部分析完才能去掉所有的符号对象, 得到精确的指针分析结果. 当摘要是基于 storeless 模型[44]的时候, 因为不会对对象进行抽象, 所以在对函数做摘要的时候可以明确变量间的别名关系, 不用等到分析 caller 时才能确定, 因此很适合按需分析, 文献 [45−47] 就是采用这种方式实现上下文敏感.

自顶向下的指针分析可以同时构建函数调用图, 不需要预处理来构建. 通常用在构建整个工程的指针分析, 我们索引的文献中, 大多都是采用这种方式来构建指针分析.

4.2.2 上下文敏感指针分析的实现

面向对象程序的上下文敏感指针分析的研究有很多, 其具体的实现也有很多, 本节对各种实现方式进行了归纳总结, 分为以下几种实现方式.

4.2.2.1 类 Andersen 的实现

如图 10 所示, 这里对上下文敏感的类 Andersen 实现进行描述[48], 第 1 列是 6 个标准的语句, 第 2 列是对应的指针求解约束条件. pt(<*x*, c>) 表示在上下文 c 下的变量 *x* 的指向集, contexts(m) 表示函数 m 所有可达的上下文, selector 表示根据给定的信息对应的目标函数的上下文, heapSelector 是对应的堆上下文, m_{this} 表示函数 m 的 *this* 变量, m_{pk} 表示函数 m 的参数 (*k* 是参数的个数).

这里的 new、ASSIGN、LOAD、STORE 语句和标准的 Andersen 实现一样, 只是对变量、函数、堆添加了上下文信息, 在处理 new 语句的时候会根据该语句所在的函数的上下文来计算对应的堆上下文. ASSIGN、LOAD、STORE 语句在指向集传递的时候需要保证 *x* 和 *y* 的上下文一致. 对于 INVOKE 语句, 这里首先会找到函数 foo 的接收对象 (receiver object, 这里就是 *a*0 指向集中的对象), 然后根据该对象和函数的签名找到对应的被调用函数 *m*, 然后根据第 4.2.1.1 节的描述, 使用调用语句和参数指向集的信息来计算用户指定的上下文, 在计算上下文之后, 实参到形参做赋值处理. RETURE 语句会伪造 m_{ret} 变量表示函数 m 的返回值, 然后变量 *x* 到变量 $m_{\rm ret}$ 做赋值处理.

一些开源工具是使用这种方式来实现上下文敏感指针分析, 如 Paddle[27,27]、WALA[12].

类 Andersen 的实现可以在指针分析的同时构建高精度的函数调用图, 不需要预处理来构建函数调用图, 因此类 Andersen 的实现被广泛应用在基础分析, 如: 构建函数调用图, 框架 WALA、SOOT 都使用了类 Andersen 的实现方法来构建函数调用图, 当目标程序太大的时候, 如果添加高精度的上下文信息会带来很大的内存和时间开销. 同时, 应用程序的规模在不断增长, 根据文献 [49] 的统计, 在过去 5 年, Google Play 上的应用软件的大小平均增长了 3~4 倍. 这给全局指针分析带来了极大的挑战.

4.2.2.2 上下文无关语言图可达

对于数据流分析可以转换成图可达问题[50,51], 在处理指针分析时, 通常会转成上下文无关语言图可达 (CFL-

© 中国科学院软件研究所 http://www.jos.org.cn

reachability) 问题[52,53], 这种方法常常用于按需 (demand-driven) 指针分析的应用中.

如图 11(a) 是图 1(a) 的简化版, 图 11(b) 是对应的图表示, 图 11(c) 是 Dyck 语言, 一种经典的上下文无关语言, 用来处理上下文敏感指针分析. 对于语句 A *a*=new A(); 会有一条 A→*a* 的边, 其中 A 是第 2 行堆的抽象. 对于语句 Object *v*1=id(*a*); 会有一条带有 $(_{L1}$ 标签的 *a*→*p* 边, $(_{L1}$ 表示在 L1 进入函数 id. 对于语句 return *p*; 会有一条带有$)_{L1}$ 标签的 *p*→*v*1 边, $)_{L1}$ 表示在 L1 从函数 id 出来. 在路径 A→*a*→*p*→*v*1 上的标签 $(_{L1},)_{L1}$ 是 Dyck 文法匹配的, 是一条合法的路径. 对于路径 A→*a*→*p*→*v*2 则是一个非法路径. 通过上下文无关语言可达性可以得出 *v*1→{A}, *v*2→{B}, 实现了上下文敏感.

函数 m 中的语句	约束
NEW: x = new A()	$\dfrac{c \in \text{contexts}(m)}{\langle o, \text{heapSelector}(c)\rangle \quad pt(\langle x, c\rangle)}$
ASSIGN: $x=y$	$\dfrac{c \in \text{contexts}(m)}{pt(\langle y, c\rangle) \subseteq pt(\langle x, c\rangle)}$
LOAD: $x=y.f$	$\dfrac{c \in \text{contexts}(m), \langle o, c'\rangle \in pt(\langle y, c\rangle)}{pt(\langle o, c'\rangle.f) \subseteq pt(\langle x, c\rangle)}$
STORE: $x.f=y$	$\dfrac{c \in \text{contexts}(m), \langle o, c'\rangle \in pt(\langle x, c\rangle)}{pt(\langle y, c\rangle) \subseteq pt(\langle o, c'\rangle.f)}$
INVOKE: x = $a0$.foo($a1$,$a2$...an)	$\dfrac{\begin{array}{c} c \in \text{contexts}(m), \langle o, c'\rangle \in pt(\langle a0, c\rangle) \\ m' = \text{dispatch}(\langle o, c'\rangle, \text{foo}) \\ args = [\{o, c'\}, pt(\langle a1, c\rangle)...pt(\langle an, c\rangle)] \\ c'' \in \text{selector}(m', c, \text{INVOKE}, args) \end{array}}{\begin{array}{c} c'' \in \text{selector}(m') \\ \langle o, c'\rangle \in pt(\langle m_{this}', c''\rangle) \\ pt(\langle a_k, c\rangle) \subseteq pt(\langle m_{pk}', c''\rangle), 1 <= k <= n \\ pt(\langle m_{ret}', c''\rangle) \subseteq pt(\langle x, c\rangle) \end{array}}$
RETURN: return x	$\dfrac{c \in \text{contexts}(m)}{pt(\langle x, c\rangle) \subseteq pt(\langle m_{ret}, c\rangle)}$

图 10 上下文敏感指针分析类 Andersen 的实现

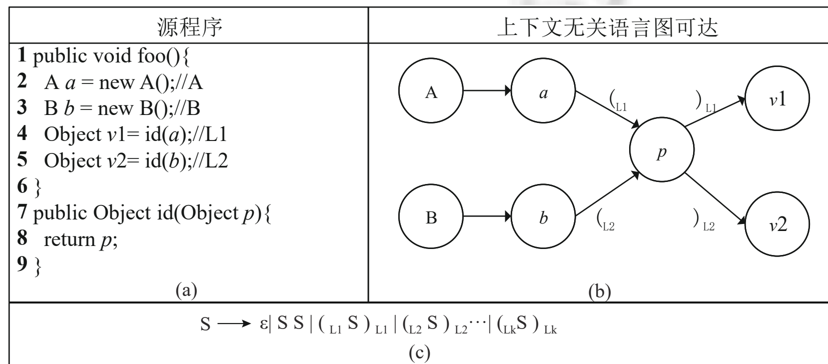

图 11 上下文无关语言图可达

© 中国科学院软件研究所 http://www.jos.org.cn

和类 Andersen 的实现方式不同, 上下文无关语言图可达的方法通常应用在按需的指针分析中, 当需要计算某个变量的指向集的时候, 该方法会在图中搜索出符合文法的路径, 但是, 当需要查询的变量过多的时候, 在极端的情况下, 所有的变量都需要计算指向集, 复杂度会迅速上升, 这时采用类 Andersen 的方法计算指向关系会更好. 在使用上下文无关语言图可达计算指向关系的时候通常需要使用低精度的指针分析构建一个初始函数调用图, 这时候通常会使用低敏感度类 Andersen 方式实现. 两种实现方法相辅相成.

4.2.2.3 主流分析框架内置指针分析技术

很多静态分析框架内置了指针分析的实现, 本小节对几个常用的分析框架的内置指针分析所使用的上下文敏感技术进行了分析.

Doop 是一个基于 Datalog 的静态程序分析框架. 使用逻辑语言来实现指针分析有很多的优势, 首先, 对于传统程序需要几百行到几千行才能实现的功能通过逻辑语言只需要几行的规则就可以完成. 然后, 所有的分析信息使用统一的模式表示, 可以很方便地使用分析结果进行组合分析. 最后, 对 Datalog 的优化可以对所有基于 Datalog 的应用都起作用. 越来越多的工具会使用 Datalog 来做指针分析[23–25,54–57]. 类 Andersen 实现方法和上下文无关语言图可达问题都可以用 Datalog 来实现.

图 12(a) 描述了基于 Datalog 上下文敏感指针分析的作用域 (如: 变量集、堆集等), 程序的输入关系、中间关系、输出关系, 还有 Record、Merge、MergeStatic 这 3 个构造上下文和堆上下文的函数[35]. ALLOC 根据 new 语句生成的谓词关系, MOVE 是根据赋值语句生成的谓词关系, LOAD 是根据加载域的语句生成的谓词关系, STORE 是根据存储域的语句生成的谓词关系, VCALL 是根据虚函数调用语句生成的谓词关系, SCALL 是根据静态调用语句生成的谓词关系. 形参和实参分别使用 FOMALARG 和 ACTUALARG 来表示, 返回值则用 FORMALRETURN 和 ACTUALRETURN 表示, THISVAR 表示函数的 *this* 变量, LOOKUP 根据信息找到对应的函数. VARPOINTSTO 和 FLDPOINTSTO 描述了变量指向关系, CALLGRAPH 描述了函数调用关系, INTERPROCASSIGN 是实参到形参和返回值间的赋值, REACHABLE 记录了函数的可达上下文. 图 12(b) 描述了指针分析和构建函数调用图的规则, 左箭头 (⟵) 表示右边的关系可以推导出左边的关系, 和第 3.3.2.1 节描述的约束条件类似, 这里使用逻辑关系来描述这些约束条件, 以赋值语句为例, MOVE (*to*, *from*) 表示将变量 *to* 赋值给变量 *from*, VARPOINTSTO (*from*, ctx, *heap*, hctx) 表示在上下文 ctx 下变量 *from* 指向堆上下文 hctx 下的对象 *heap*, 通过分析规则可以得到关系 VARPOINTSTO (*to*, ctx, *heap*, hctx) 表示堆上下文 hctx 下的对象 *heap* 添加到上下文 ctx 下的变量 *to* 的指向集中.

不同类型的上下文敏感可以通过 Record、Merge、MergeStatic 来描述, 使用 Datalog 语言处理指针分析进行不同的上下文敏感方式切换非常方便, 图 13 是文献 [31] 中描述的基于 Datalog 的各种不同的上下文敏感方式, 图中涵盖了第 4.2.1.1 节提到的各种上下文敏感. 图 13 中的 invo 是指调用点的唯一表示, T(*heap*) 函数是用于计算 *heap* 对象的类型, first(ctx) 函数用来取 ctx 的第 1 个值, second(ctx) 用于取 ctx 第 2 个值, pair 和 triple 的用法和实际中一致, 形成二元组和三元组.

Doop 内置的指针分析通过基于标签的方式实现上下文敏感, 包括: call-site-sensitive、object-sensitive、type-sensitive 以及一些对这 3 种的优化和变种, Doop 的指针分析非常容易扩展上下文的深度. 还有一些静态分析工具是基于 Datalog 实现的上下文敏感指针分析, 如: Chord[58]、Petablox[59].

Soot 内置了两种指针分析的实现, Spark 和 Paddle, 其中 Spark 是上下文不敏感的. Paddle 支持多种基于标签的上下文敏感, 包括: call-site-sensitive、object-sensitive. 为了解决添加上下文敏感带来的内存增加的问题, Paddle 使用 BDD 来表示指向关系, 从而减少内存消耗, BDD 的方式虽然可以减少内存, 但是会增加时间上的开销.

Soot 还内置实现了 IFDS/IDE 分析框架, 该框架通过摘要的方式实现上下文敏感. 基于 Soot 的 IFDS/IDE 框架, 文献 [45−47] 实现了上下文敏感的别名分析.

WALA 同样内置了多种基于标签的上下文敏感, 包括: call-site-sensitive、object-sensitive. 通过修改 WALA 提供的上下文选择器可以很容易地实现不同深度的上下文敏感. 和 Soot 一样, WALA 内置实现了 IFDS/IDE 分析框架, 但是没有对应的基于摘要的上下文敏感指针分析的实现.

© 中国科学院软件研究所 http://www.jos.org.cn

(a)

```
V 程序变量集
H 堆的抽象集 (i.e., allocation sites)
M 函数集
S 函数签名
F 域集
I 指令集
T 类的类型集
N 自然数集
C 上下文
HC 堆上下文

ALLOC (var : V, heap : H, inMeth : M) # var = new ...
MOVE (to : V, from : V) # to = from
LOAD (to : V, base : V, fld : F) # to = base.fld
STORE (base : V, fld : F, from : V) # base.fld = from
VCALL (base : V, sig : S, invo : I, inMeth : M) # base.sig(..)
SCALL (meth : M, invo : I, inMeth : M) # Class.meth(..)
FORMALARG (meth : M, i : N, arg : V)
ACTUALARG (invo : I, i : N, arg : V)
FORMALRETURN (meth : M, ret : V)
ACTUALRETURN (invo : I, var : V)
THISVAR (meth : M, this : V)
HEAPTYPE (heap : H, type : T)
LOOKUP (type : T, sig : S, meth : M)

VARPOINTSTO(var : V, ctx : C, heap : H, hctx : HC)
CALLGRAPH(invo : I, callerCtx : C, meth : M, calleeCtx : C)
FLDPOINTSTO(baseH: H, baseHCtx: HC, fld: F, heap: H, hctx: HC)
INTERPROCASSIGN(to : V, toCtx : C, from : V, fromCtx : C)
REACHABLE(meth : M, ctx : C)

RECORD(heap : H, ctx : C) = newHCtx : HC
MERGE (heap : H, hctx : HC, invo : I, ctx : C) = newCtx : C
MERGESTATIC(invo : I, ctx : C) = newCtx : C
```

(b)

```
INTERPROCASSIGN (to, calleeCtx, from, callerCtx) ←
        CALLGRAPH (invo, callerCtx, meth, calleeCtx),
        FORMALARG (meth, i, to), ACTUALARG(invo, i, from).

INTERPROCASSIGN (to, callerCtx, from, calleeCtx) ←
        CALLGRAPH (invo, callerCtx, meth, calleeCtx),
        FORMALRETURN (meth, from), ACTUALRETURN(invo, to).

RECORD (heap, ctx) = hctx,
        VARPOINTSTO (var, ctx, heap, hctx) ←
        REACHABLE (meth, ctx), ALLOC(var, heap, meth).

VARPOINTSTO (to, ctx, heap, hctx) ←
        MOVE(to, from),
        VARPOINTSTO(from, ctx, heap, hctx).

VARPOINTSTO (to, toCtx, heap, hctx) ←
        INTERPROCASSIGN (to, toCtx, from, fromCtx),
        VARPOINTSTO(from, fromCtx, heap, hctx).

VARPOINTSTO (to, ctx, heap, hctx) ←
        LOAD (to, base, fld),
        VARPOINTSTO (base, ctx, baseH, baseHCtx),
        FLDPOINTSTO (baseH, baseHCtx, fld, heap, hctx).

FLDPOINTSTO (baseH, baseHCtx, fld, heap, hctx) ←
        STORE (base, fld, from),
        VARPOINTSTO (from, ctx, heap, hctx),
        VARPOINTSTO (base, ctx, baseH, baseHCtx).

MERGE (heap, hctx, invo, callerCtx) = calleeCtx,
REACHABLE (toMeth, calleeCtx),
VARPOINTSTO (this, calleeCtx, heap, hctx),
CALLGRAPH (invo, callerCtx, toMeth, calleeCtx) ←
        VCALL (base, sig, invo, inMeth),
        REACHABLE (inMeth, callerCtx),
        VARPOINTSTO (base, callerCtx, heap, hctx),
        HEAPTYPE (heap, heapT),
        LOOKUP (heapT, sig, toMeth),
        THISVAR (toMeth, this).

MERGESTATIC(invo, callerCtx) = calleeCtx,
REACHABLE(toMeth, calleeCtx),
CALLGRAPH(invo, callerCtx, toMeth, calleeCtx) ←
        SCALL(toMeth, invo, inMeth),
        REACHABLE(inMeth, callerCtx).
```

图 12 基于 Datalog 的上下文敏感指针分析

4.2.3 上下文敏感指针分析的优化

4.2.3.1 优化数据结构

随着上下文深度的增加上下文的数量急剧增长, 传统的数据结构来表示指针指向信息已经无法满足需求, 需要新的数据结构来解决内存问题. BDD 是一种新型数据结构, 最早使用在硬件验证和模型检查, 随后被用在面向 C/C++程序的上下文敏感指针分析[60,61]. 对于面向对象程序来说, 使用基于克隆 (clone-based) 的方法来实现上下文敏感同样面临上下文数量爆炸的问题, 文献 [62] 使用 BDD 数据结构来表示指向关系, 从而减少内存. 在 BDD 中同样存在冗余的节点, ROBDD、XBDD 和 ZBDD[63,64]通过删除冗余的节点对 BDD 进行优化, 从而减少 BDD 的大小, 在节省内存的同时可以提高搜索速度. 工具 bddbddb[65]通过将 Datalog 语言中的关系转成布尔函数, 然后使用 BDD 技术来表示这些布尔函数, 降低内存消耗. 使用 BDD 数据结构进行优化的方法虽然可以很大程度上压缩内存, 但是会很耗时, 因此在很多静态分析框架中都没有使用 BDD 作为默认指向关系存储方式.

在使用 Datalog 处理上下文无关语言图可达问题的时候需要进行大量的对 Datalog 中的关系进行连接、并等操作, 和邻接矩阵和稠密矩阵相比较, 通过使用四叉树可以降低内存消耗同时提高效率[66].

对于基于克隆 (clone-based) 的上下文敏感指针分析, 处理一条 new 语句: A *a*=new A(); 或者一条赋值语句

© 中国科学院软件研究所 http://www.jos.org.cn

b=*a*;, 有多少上下文就至少需要分析该语句多少次. 如果对于某条 new 语句或者赋值语句上下文有 10 个, 如果可以将这 10 个上下文表示成连续的数字 [1:11), 那么对这 10 个上下文下的该语句可以一次性处理, 基于这个发现文献 [66] 提出了几何编码 (geometric encoding) 对指向关系进行表示, 可以减少冗余的计算.

1-call-site-sensitive	1-call-site-sensitive+1-H
RECORD (*heap*, ctx) = * MERGE (*heap*, hctx, invo, ctx) = invo MERGESTATIC (invo, ctx) = invo	RECORD (*heap*, ctx) = ctx MERGE (*heap*, hctx, invo, ctx) = invo MERGESTATIC (invo, ctx) = invo
1-object-sensitive	2-object-sensitive+1-H
RECORD (*heap*, ctx) = * MERGE (*heap*, hctx, invo, ctx) = heap MERGESTATIC (invo, ctx) = ctx	RECORD (*heap*, ctx) = first (ctx) MERGE (*heap*, hctx, invo, ctx) = pair (*heap*, hctx) MERGESTATIC (invo, ctx) = ctx
2-type-sensitive+1-H	1-object+1-call-site-sensitive
RECORD (*heap*, ctx) = first (ctx) MERGE (*heap*, hctx, invo, ctx) = pair (T(*heap*),hctx) MERGESTATIC (invo, ctx) = ctx	RECORD (*heap*, ctx) = * MERGE (*heap*, hctx, invo, ctx) = pair (*heap*,invo) MERGESTATIC (invo, ctx) = pair (first(ctx),invo)
2-object+1-call-site-sensitive+1-H	2-type+1-call-site-sensitive+1-H
RECORD (*heap*, ctx) = first (ctx) MERGE (*heap*, hctx, invo, ctx) = triple (*heap*, hctx, invo) MERGESTATIC (invo, ctx) = triple (first (ctx), second (ctx), invo)	RECORD (*heap*, ctx) = first(ctx) MERGE (*heap*, hctx, invo, ctx) = triple (T(*heap*), hctx, invo) MERGESTATIC (invo, ctx) = triple (first(ctx), second (ctx), invo)
Selective 1-object-sensitive hybrid A	Selective 1-object-sensitive hybrid B
RECORD (*heap*, ctx) = * MERGE (*heap*, hctx, invo, ctx) = *heap* MERGESTATIC (invo, ctx) = invo	RECORD (*heap*, ctx) = * MERGE (*heap*, hctx, invo, ctx) = pair (*heap*,*) MERGESTATIC (invo, ctx) = pair (first(ctx),invo)
Selective 2-object+1-H hybrid	Selective 2-type+1-H hybrid
RECORD (*heap*, ctx) = first (ctx) MERGE (*heap*, hctx, invo, ctx) = triple (*heap*, hctx, *) MERGESTATIC (invo, ctx) = triple (first (ctx), invo, second (ctx))	RECORD (*heap*, ctx) = first (ctx) MERGE (*heap*, hctx, invo, ctx) = triple (T(*heap*), hctx, *) MERGESTATIC (invo, ctx) = triple (first(ctx), invo, second (ctx))

图 13 基于 Datalog 不同方式的上下文敏感

4.2.3.2 基于摘要的优化

以图的方式作为摘要信息的时候摘要信息会很大, 在传递闭包的时候计算量就会很大, 如果采取随机的顺序计算闭包, 时间复杂度最坏是 $O(n^3)$, 为了减少不必要的传播, 文献 [41] 会构建一个拓扑序, 然后按照拓扑序进行传播, 由于图中可能会存在环的情况, 该文认为环中的节点拥有相同的指向集, 因此可以将环简化成一个节点, 从而可以减少分析时间并降低存储空间.

对于使用指向集作为摘要信息来说, 只有被调用函数的逃逸 (escape) 对象会对调用函数有影响, 在做摘要的时候只需要对逃逸对象做摘要即可, 文献 [67] 先对每个函数做过程内分析, 找到逃逸对象并对其做摘要, 然后进行自顶向下分析实现流敏感、上下文敏感指针分析.

在开发程序的时候会用到很多第三方库, 在做指针分析的时候第三方库和应用程序会被一起分析, 分析代码量会增大很多, 会消耗很大的内存和计算资源. 对第三方库函数生成摘要, 然后在处理应用程序时遇到了库函数调用, 直接使用摘要信息可以提高分析效率[68−72]. 文献 [72] 通过给摘要添加约束条件来提高分析精度.

4.2.3.3 选择性使用上下文敏感

对于上下文敏感指针分析来说, 整个程序中不是所有的函数都需要上下文敏感, 或者不同的函数需要不同类型的上下文敏感, 对不同的函数进行区别对待可以很好地平衡精度和效率, 如何准确快速的区分这些函数是关键.

通过定义一些启发式规则可以对函数进行区分. 对于静态函数而言是没有接收对象的, 所以在使用对象敏感或者类型敏感时非静态函数是继承调用函数的上下文, 如果对静态函数和非静态函数进行区分, 对于静态函数, 添加调用点到上下文信息中, 则可以在精度和效率进行平衡[35].

© 中国科学院软件研究所 http://www.jos.org.cn

文献 [73] 提出了 6 个启发式规则: 某函数在某调用点的实参指向集之和、某函数的本地变量的指向集之和、某个对象的所有的域的指向集之和、函数所有的本地变量指向集中的每个对象的所有的域的指向集之和、某个对象被多少个变量指向、某个对象被多少个对象的域指向. 通过设置阈值, 使用这 6 个启发式规则的组合来区分某个函数该使用哪种类型的上下文.

文献 [74] 定义了两个启发式规则: *InFlow*(*o*) 表示对于某个对象 *o* 有多少个对象可以通过 STORE 指令添加到对象 *o* 的域中, *OutFlow*(*v*,*o*,oc) 表示通过 LOAD 指令从在堆上下文 oc 下的对象 *o* 中加载并赋值给变量 *v* 的对象的个数, 利用低精度的指针分析的结果来计算上述两个规则, 然后通过这两个启发式规则找到需要更高精度的对象, 然后对这些对象所在的函数增加上下文的深度, 不断的迭代, 直到满足客户端需求.

机器学习也是一种区分不同函数的方法. 文献 [75] 使用统计的方法提出了两种策略: *STATREFINE* 先使用低精度的上下文敏感做指针分析, 根据客户端指定的查询信息去训练模型, 根据训练的模型得出哪些函数或对象需要更高的精度. *ACTIVECOARSEN* 和 *STATREFINE* 正好相反, 从最高精度的上下文敏感开始做指针分析, 然后根据学习出来的模型不断对部分函数或对象降低精度.

文献 [76,77] 选取语言底层的一些特征 (“java”“sun”“void”等描述符特征和赋值语句、返回语句等语句特征), 通过机器学习算法, 学习不同的函数对应的上下文深度, 在保证精度接近对所有函数使用深度为 k 的精度时, 保证代价最小.

上面提到的都是基于启发式规则或者机器学习的方法来选取上下文深度, 这两种方法不能够洞察需要高敏感度的这些函数的本质. 文献 [78] 定义了两种函数和 3 种流: *InMethod* 表示有参数的函数; *OutMethod* 表示有返回值的函数; *DirectFlow* 表示某对象从某个类的 *InMethod* 的参数流入, 从同一个类的 *OutMethod* 的返回值流出; *WrappedFlow* 表示某对象流入了某个类的 *InMethod* 的参数的域中, 然后从同一个类的 *OutMethod* 的返回值流出; *UnwrappedFlow* 表示对象流入了某个类的 *InMethod* 的参数, 然后通过 LOAD 指令取出该对象的域, 然后该域从同一个类的 *OutMethod* 的返回值流出. 如果对于某个对象来说, 从某个类的 *InMethod* 通过上述的 3 种数据流流到了该类的 *OutMethod*, 则该对象流过的路径上的函数都需要高精度.

和文献 [78] 旨在以精度为导向不同, 文献 [79] 则把视角立足于可扩展性, 在给定的时间和内存下尽可能对每个函数选择高精度的上下文, 其中主要问题是如何快速评估每个函数在不同上下文下需要的内存的大小, 该文使用本地变量指向集大小和某种类型上下文下可能的上下文个数乘积来表示该函数需要的内存大小, 对于本地变量指向集通过上下文不敏感指针分析作为预处理得到, 对于某类型上下文下某函数的上下文的数量, 该文通过构建对象分配图 (OAG), 对于某个函数的接收对象, 在 OAG 中到达该对象的 *k* 深度的路径个数表示 *k* 值下该函数可能的上下文数量.

对于对象敏感指针分析来说, 上下文表示成一条长度为 *k* 的对象分配点的链, 某个函数的多条上下文路径可能包含同一个对象, 当使用该对象作为上下文时, 起不到区分上下文的作用, 会出现冗余, 文献 [80] 通过构建对象分配图 (OAG) 找到每个函数对应的这些冗余的对象, 在进行对象敏感指针分析的过程中遇到这些对象的时候不添加到对应的函数的上下文信息中, 在使用相同的 *k* 的时候可以得到更高的精度.

4.2.3.4 按需指针分析

上下文无关语言图可达常用来做按需指针分析, 上下文无关语言的求解复杂度过高, 文献 [81] 通过在上下文无关语言可达图中添加匹配 (match) 边 (对于同一个域的 LOAD 和 STORE 边认为是匹配的), 将上下文无关语言图可达问题转化成正则语言图可达问题, 从而可以快速实现一个上下文不敏感域敏感的指针分析, 然后通过提炼来解决正则语言带来的精度损失. 文献 [82] 同样使用了上面提到的匹配 (match) 边的手段, 并且做了更多的精简: ① 忽略赋值语句; ② 不允许部分上下文敏感, 最后针对指定客户端需求来提炼精度.

使用上下文无关语言图可达来处理上下文敏感指针分析的时候, 对于少量的需求来说效果会很好, 但是当需要分析的变量曾多的时候扩展性就会变差, 文献 [83] 通过使用 SPG 快速有效计算出 must-not-alias 关系, 然后在使用上下文无关语言图可达做上下文敏感指针分析的时候对 must-not-alias 的路径进行删减, 从而减少计算量. 文献 [84] 允许用户设置参数, 在可达性计算过程中如果搜索节点数量或者时间超过一定阈值, 就直接转成上下文不

© 中国科学院软件研究所 http://www.jos.org.cn

敏感分析. 在按需指针分析中, 使用基于摘要的技术可以减少冗余计算[85].

IFDS[86]是处理特殊的数据流问题的算法, 常用于污点传播分析, 在处理 IFDS 问题的时候同样需要计算别名关系, 在使用 IFDS 做污点分析常用按需的后向传播的算法寻找别名[45,46]. 文献 [47] 通过扩展 IDE[87]问题来做上下文敏感别名分析.

4.2.3.5 客户端驱动的指针分析

指针分析一般会伴随着其他应用场景存在, 如: 构建函数调用图、污点分析等. 按需指针分析是在某具体应用分析过程中, 如果需要得到某个变量的指向集或者别名信息, 这时启动一个指针分析去计算相应的信息. 而客户端驱动的指针分析不同, 该分析的目标是为了满足客户端的需求, 只针对客户端关注的部分代码进行分析, 而不在乎与客户端无关的代码. 按需指针分析和客户端驱动的指针分析的本质相似, 都是只分析部分代码来减少分析的代码量, 从而可以实现高精度的同时保证良好的扩展性.

客户端驱动的指针分析[88]可以根据指定的客户端需求对部分代码使用高精度的上下文敏感指针分析.

基于克隆是处理上下文敏感指针分析的一种主要方式, 但是需要克隆的函数数量过大会带来扩展性差的问题, 有选择的克隆是有效的解决方法, 文献 [89] 从上下文不敏感开始, 判断是否满足指定客户端的全部需求, 如果不满足则记录不满足的路径集合, 采用最小克隆原则, 找到影响路径最多的调用点, 对该调用点进行克隆, 实现上下文敏感, 不断迭代, 直到满足指定客户端的需求.

程序切片是程序分析中一个重要的技术手段, 通过程序切片可以减少需要分析的代码, 便可以对部分代码片段进行高精度的上下文敏感进行指针分析. 文献 [90] 根据特定的客户端做切片分析, 选取客户端相关的代码片段, 然后使用上下文不敏感的指针分析对代码片段做指针分析, 对切片进行提炼, 进一步压缩代码, 最后在压缩后的代码上进行高敏感度的指针分析. 文献 [91,92] 先用上下文不敏感的指针分析做预处理, 然后根据特定的客户端需求进行查找, 对于查找失败的客户端需求来说找到相关的对象或者变量, 然后对这些对象和变量对应的函数做更高敏感度的分析, 这个过程可以迭代多次, 直到满足客户端需求.

文献 [93] 则把关注点放在堆抽象上, 先用粗粒度的抽象做程序进行指针分析, 然后根据客户端需求删除客户端无关的部分, 然后在剩下的程序上对堆做细粒度的抽象, 不断的迭代, 直到满足客户端的需求.

有些基于指针分析的应用是类型依赖的 (如: 函数调用图、虚函数解析和类型安全转换), 对于函数调用图分析来说, 如果对于两个对象 *o*1 和 *o*2, *o*1 和 *o*2 的类型一样, 并且 *o*1 和 *o*2 的每个域所指向的对象的类型一样且唯一, 如: *o*1.*f* 和 *o*2.*f* 指向的对象类型的类型一样且唯一, *o*1.*f*.*g* 和 *o*2.*f*.*g* 指向的对象的类型一样且唯一, 这样对象 *o*1 和 *o*2 可以使用同一个堆抽象来表示, 这样可以减少大量的堆抽象. 文献 [94] 对每个对象构建一个域指向图 (FPG), 然后将 FPG 转化成自动机, 通过求解自动机等价类来找到上面描述的对象.

4.2.3.6 增量分析

上下文敏感指针分析代价是很大的, 尤其对于大的工程来说, 完整的分析一次非常浪费时间和资源, 在开发过程中每次代码有变动都对整个工程重新分析显然不是很好的选择, 增量分析可以快速处理小范围代码变动的情况. 当代码出现变动的时候, 部分变量会出现增加和删除的情况, 通过数据流分析可以找到受影响的变量和对应的指令, 对这些变量的指向集清空然后对这些指令重新分析可以快速计算出新的指向关系[95]. 使用上面方法的时候需要对分析过的变量的指向集进行保存, 对于图可达问题来说则不需要. 文献 [96] 对函数内部的可达性做摘要, 当部分程序修改时候, 只需要修改程序内部的可达性摘要即可. 文献 [97] 则是客户端驱动的, 对客户端驱动的每个查找路径进行保存, 当代码改变的时候, 如果改变部分对这条路径上没有影响, 对于客户端来说就不需要重新分析. 文献 [98] 对文献 [99] 进行扩展, 在进行逃逸分析和指针分析的时候对每个函数独立进行不完全分析, 跳过所有的调用指令, 对于特定的需求进行增量分析, 不断对丢掉的调用语句进行补充, 从而提高精度.

4.2.3.7 解决 k-limit 带来的精度损失

对于基于标签的上下文敏感来说, 上下文的深度是无限的, 为了有限的表示上下文, 常用的技术是 k-limit, 即: 截取最近的 *k* 个上下文, 这种做法必然会带来精度的损失. 加权下推系统 (weighted pushdown system, WPDS) 常被用在模型检查中, WPDS 有栈结构, 可以表示无限的状态, 非常符合指针分析中上下文无限、域无限的问题, 使用

© 中国科学院软件研究所 http://www.jos.org.cn

WPDS 可以解决 k-limit 的问题[100−102].

由于 WPDS 表达能力限制, WPDS 只可以单独处理域敏感或者上下文敏感, 在单独处理上下文敏感的时候自然会有一定的精度损失, 为了提高基于 WPDS 上下文敏感指针分析的精度, 文献 [103] 提出了条件加权下推系统 (conditional weighted pushdown systems, CWPDS), 通过给传递规则 (transition rule) 添加约束条件来提高精度.

为了实现同时满足上下文敏感和域敏感指针分析, 使用 WPDS 分别做域敏感指针分析和上下文敏感指针分析, 然后对结果进行合并, 同时满足上述两个条件的指向关系视为合法[36]. 这样分开处理然后对结果合并会带来精度损失, 可能同时存在一条域敏感合法的路径和一条上下文合法的路径使得某变量指向某对象, 但是在域敏感合法的路径上上下文敏感不合法, 在上下文敏感合法的路径上域敏感不合法, 这样会带来精度损失, 在实践中这种简单的合并精度损失微乎其微[102].

在使用上下文无关语言图可达来同时处理域敏感和上下文敏感时也会存在上下文无关语言表达能力有限制的问题, 如果想同时满足域敏感和上下文敏感同样只能分开使用上下文无关语言图可达来分析然后对结果合并, 这样会带来精度的损失, 文献 [104] 提出了线性合取语言 (linear conjunctive language), 该语言的表达能力可以满足同时处理 field-sensitive 和 context-sensitive 的指针分析.

4.2.3.8　基于外设的图计算系统

对于基于指针分析的应用来说, 为了降低误报率和漏报率, 需要使用高精度的上下文敏感指针分析, 高精度的上下文敏感指针分析的扩展性一直以来都是一个问题, 随着程序增大上下文的数量成指数增加, 因此实现可用的高精度的上下文敏感指针分析复杂度很高, 这就使得高精度上下文敏感的指针分析很难应用到大的系统中. 文献 [105] 采用上下文无关语言图可达的方式实现上下文敏感指针分析, 当图很大的时候, 该文将图保存在磁盘中, 然后按需将图的部分数据加载到内存中并做图可达计算, 同时将结果保存到磁盘中, 这样就解决了内存不足的问题, 同时还可以利用已有的图计算系统进行优化, 解决了实现复杂度高的问题, 同时, 相比较传统使用动态规划线性地解决图可达问题, 该文使用了并行化处理图可达问题, 从而提高速度. 该方法是分析 C 语言的, 目前还没发现有面向对象程序的上下文敏感指针分析使用该方法.

4.2.3.9　其　他

在使用基于克隆 (clone-based) 的技术实现上下文敏感的时候不是所有的变量都需要克隆多份分别放到不同的调用函数中, 于某个函数中的某个变量的指向集在不同的上下文下都一样, 如果对每个上下文都生成唯一拷贝就会带来大量冗余, 在对变量进行克隆的时候, 如果该克隆和已有的版本指向集一致, 则只保留一份对精度没有任何影响, 从而可以去掉很多冗余的计算和存储[40].

Dyck 语言图可达 (Dyck-CFL-Reachability) 问题表示成的图是单向边, 在真实应用中会存在双向边的情况, 即: 对于图中每对相连的节点 a 和 b, 这两个节点存在 a→b 的边上有标签 a, 那么必定存在 b→a 的边带有标签 $\bar{a}$, 反之同样满足. 在指针分析中, 基于合并 (unification-based) 的算法就可以表示成双向 Dyck 语言图可达 (bidirected Dyck-CFL-reachability) 问题, 在处理该问题的时, 对于两个节点, 如果入边和出边都一样, 那么这两个节点就可以进行合并, 删除冗余的表示[106].

对于基于标签的上下文表示来说, 如果使用深度为 k 的上下文, 传统的做法会枚举所有的上下文, 当 k 比较大的时候上下文的数量会非常大, 这样不但会出现记录信息过大的问题, 也会增加程序计算复杂度, 文献 [107] 提出了上下文传递的概念, 通过使用把上下文表示成函数的方式, 只需要记录最近上下文, 完整的上下文信息可以通过最近上下文的上下文和最近上下文拼接得到, 从而减少了上下文的表示数量.

在使用 Datalog 来求解上下文敏感指针问题时, 文献 [108] 提出 semi-naïve 算法来加速求解, 在迭代过程中, 不是每次迭代都需要对所有的关系实例带入规则中求解, 只需要对新生成的关系实例进行迭代即可.

对于使用 value context 作为上下文来说, 对于面向对象程序可能会存在很深的域的情况, 这样上下文的数量就会很多, 通过预处理来判断对于某个参数所需要分析域的深度可以减少不必要的上下文[109].

逃逸分析可以判断某个函数的内部指向关系是否会影响其调用函数的指向关系. 文献 [109] 通过逃逸分析找到这些函数, 在分析调用函数的时候先跳过这些函数, 等到迭代结束后分析这些跳过的函数, 这样可以减少不必要

© 中国科学院软件研究所 http://www.jos.org.cn

的迭代. 文献 [99] 则通过逃逸分析和指针分析相结合, 只对部分需要逃逸的对象进行分析即可, 从而提高效率.

文献 [110] 则实现了轻量级的对象敏感指针分析, 该文提出 ObjectGraph(Ag), Ag 描述了一种对象访问关系, 对于对象 *o*1 和 *o*2, 如果说 *o*1 可访问 *o*2, 那么需要满足两个条件: ① *o*1 可通过域访问 *o*2 (即 *o*1.*f*=*o*2); ② *o*1 是函数 m 的接收对象 (receiver object), m 中有某个本地变量指向 *o*2. Ag(*o*) 定义了对象 *o* 可访问的对象集合. 然后通过结合 ObjectGraph 和指向图 (point-to graph) 来实现对象敏感. 对于一个变量 *r*, *r* 指向对象 *o*2(可以通过指向图得到), *r* 所在的函数 m 的接收对象 (receiver object) 是 *o*1, 如果 *o*2∈Ag(*o*1), 那么表示在上下文 *o*1 下函数 m 中的变量 *r* 指向 *o*2.

对于非面向对象程序的指针分析来说, 也存在平衡精度和效率的问题. 文献 [111,112] 在多核 CPU 上实现了 Andersen 算法, 文献 [113,114] 将基于包含的指针分析算法实现在了 GPU 上, 文献 [115] 将指针分析实现在了异构 CPU-GPU 系统上.

传统的指针分析通常是对每个程序中的变量单独区分 (value-based), 计算每个变量的指向集, 文献 [116] 发现, 对于某些应用场景来说, 某些变量可以放在一个集合中 (set-based) 来计算指向关系, 提出了 set-based 的预处理, 从而可以简化指针分析的输入, 提高分析效率.

文献 [117] 对基于包含的指针分析的一些在线优化和离线优化进行了分析, 对一些常用的优化技术进行了比较. 这些在非面向对象程序中常用的优化方法同样可以在面向对象程序中使用, 从而提高指针分析的效率和精度.

4.2.4 上下文敏感别名分析

和指向集 (point-to) 分析不同, 别名分析不需要知道变量具体的指向对象是什么, 只需要得到两个变量是不是别名关系即可. 对于面向对象程序, 每个对象中通常会包含一些域, 这些域可能是基础类型, 也可能是引用类型, 这样就会出现嵌套域的情况, 会存在某个变量 *a* 会和某个变量 *b* 的某个域 *f* 所指向的对象中的某个域 *g* 是别名, 即: *a* 和 *b*.*f*.*g* 是别名. 在针对面向对象程序的别名分析中, 常用 AccessPath 来表示变量, 由于嵌套域的存在, AccessPath 中域的深度会存在不可数的情况, 对于这种情况通常使用 k-limit 技术解决. 根据别名关系的确定性可以分为 must-alias 和 may-alias, 对于中等大小程序来说使用 must-alias 别名分析效果会很好, 对于大程序来说, may-alias 别名分析效果会更好[48]. 文献 [118] 基于 Datalog 实现了上下文敏感的 must-alias 关系, 在实现上下文敏感的时候采用了基于标签的方法. 文献 [45,46] 基于 IFDS 框架, 通过后向传播算法计算 may-alias, IFDS 框架是上下文敏感的数据流分析框架, 它是一种特殊的 CFL-reachability 问题. 文献 [47] 基于 IDE 框架实现了上下文敏感、流敏感的别名分析, IDE 框架对 IFDS 框架进行了抽象, 本身也是一种特殊的 CFL-reachability 问题. k-limit 技术会带来精度的损失, 为了避开 k-limit 带来的精度损失, AccessGraph[119−121]、AliasGraph[121]这种图结构被用来表示变量, 从而可以表示无限域的情况.

4.2.5 上下文敏感指针分析的评估指标

对于效率的评估是容易的, 通常使用整个指针分析的时间和内存来评估效率. 对于精度的评估相对不是很容易, 用来评估精度的指标也不尽相同, 最直接的评估方式是每个引用平均指向的对象的个数[122], 这种做法有 3 个重要的缺陷: 对象的个数和堆的抽象相关, 不同的堆抽象间无法评估; 指针分析通常是给一些客户端提供支持, 这种指标无法衡量指针分析对客户端的影响; 即使堆抽象一致, 该指标无法区分不同上下文敏感对不同的变量的影响[5,123]. 对于面向对象程序的上下文敏感指针分析来说, 在评估精度的时候大多数会使用一些客户端分析来比较精度, 如: 类型安全转换、虚函数解析、可达函数、函数调用图. 也有通过每个函数的上下文数量来衡量精度[30,31], 文献 [29] 根据每个对象作为接收对象的调用点的数量信息提出了引用变量精度 (precision reference value) 来比较精度. 文献 [32] 使用更细粒度的指标 (函数调用图边和节点数量、堆抽象的数量、进入某个函数的对象的数量) 来对比精度.

5 总 结

本文使用系统文献综述 (SLR) 的方法对面向对象程序的上下文敏感指针分析进行调研, 将索引到的文献归为

© 中国科学院软件研究所 http://www.jos.org.cn

以下 5 类: 上下文的表示方法、上下文敏感指针分析的实现方法、上下文敏感指针分析的优化、上下文敏感别名分析、上下文敏感指针分析的评估指标. 其中上下文的表示方法主要有基于标签和基于摘要的两种方式. 上下文敏感指针分析的实现方法主要有类 Andersen 的实现方法、上下文无关语言图可达、基于 Datalog 实现. 上下文敏感指针分析随着精度的提升效率通常会降低, 同时可扩展性也随之下降, 为了在精度和效率之间做平衡, 有很多优化手段被提出, 如: 按需分析、增量分析、部分使用上下文敏感、优化数据结构等. 如何更好的在高精度下提高扩展性是目前研究的一个热点. 对于某些应用 (如: 污点分析) 来说不需要得到变量的指向集, 只需要得到变量间的别名关系即可, 针对这样的应用在分析过程中不需要保存指向关系, 一些工作使用 AccessPath、AccessGraph、AliasGraph 等来表示变量, 通过数据流分析获取别名关系, 别名分析在处理上下文敏感时所使用的技术和指针分析相同, 都可以使用基于标签和基于摘要的方式实现上下文敏感. 针对上下文敏感指针分析的精度的评估也有很多指标, 比如: 使用指向集的大小来比较、使用特定的客户端来比较等. 面向对象程序的上下文敏感指针分析目前还存在不足, 如何更好地在精度和效率间做平衡还是一个需要解决的问题.

References:

[1] Horwitz S. Precise flow-insensitive may-alias analysis is NP-hard. ACM Trans. on Programming Languages and Systems, 1997, 19 (1): 1–6. [doi: 10.1145/239912.239913]

[2] Landi W, Ryder BG. A safe approximate algorithm for interprocedural aliasing. In: Proc. of the ACM SIGPLAN 1992 Conf. on Programming Language Design and Implementation. San Francisco: ACM, 1992. 235–248. [doi: 10.1145/143095.143137]

[3] Ramalingam G. The undecidability of aliasing. ACM Trans. on Programming Languages and Systems, 1994, 16(5): 1467–1471. [doi: 10.1145/186025.186041]

[4] Landi W. Undecidability of static analysis. ACM Letters on Programming Languages and Systems, 1992, 1(4): 323–337. [doi: 10.1145/161494.161501]

[5] Hind M. Pointer analysis: Haven’t we solved this problem yet? In: Proc. of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. Snowbird: ACM, 2001. 54–61. [doi: 10.1145/379605.379665]

[6] Ryder BG. Dimensions of precision in reference analysis of object-oriented programming languages. In: Proc. of the 12th Int’l Conf. on Compiler Construction. Warsaw: Springer, 2003. 126–137. [doi: 10.1007/3-540-36579-6_10]

[7] Steensgaard B. Points-to analysis in almost linear time. In: Proc. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. St. Petersburg Beach: ACM, 1996. 32–41. [doi: 10.1145/237721.237727]

[8] Andersen LO. Program analysis and specialization for the C programming language [Ph.D. Thesis]. Copenhagen: University of Copenhagen, 1994.

[9] Lhoták O, Hendren L. Scaling Java points-to analysis using spark. In: Proc. of the 12th Int’l Conf. on Compiler Construction. Warsaw: Springer, 2003. 153–169. [doi: 10.1007/3-540-36579-6_12]

[10] Rountev A, Milanova A, Ryder BG. Points-to analysis for Java using annotated constraints. ACM SIGPLAN Notices, 2001, 36(11): 43–55. [doi: 10.1145/504311.504286]

[11] Whaley J, Lam MS. An efficient inclusion-based points-to analysis for strictly-typed languages. In: Proc. of the 9th Int’l Static Analysis Symp. Madrid: Springer, 2002. 180–195. [doi: 10.1007/3-540-45789-5_15]

[12] Sridharan M, Fink SJ. The complexity of Andersen’s analysis in practice. In: Proc. of the 16th Int’l Static Analysis Symp. Los Angeles: Springer, 2009. 205–221. [doi: 10.1007/978-3-642-03237-0_15]

[13] Razafimahefa C. A study of side-effect analyses for Java [MS. Thesis]. Montreal: McGill University, 1999.

[14] Streckenbach M, Snelting G. Points-to for Java: A general framework and an empirical comparison. Universität Passau, 2000.

[15] Liang DL, Pennings M, Harrold MJ. Extending and evaluating flow-insenstitive and context-insensitive points-to analyses for Java. In: Proc. of 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. Snowbird: ACM, 2001. 73–79. [doi: 10.1145/379605.379676]

[16] Kitchenham B. Procedures for performing systematic reviews. Technical Report, Australia: Keele University and National ICT, 2004.

[17] Nguyen P H, Kramer M, Klein J, Traon YL. An extensive systematic review on the Model-Driven Development of secure systems. Information and Software Technology, 2015, 68: 62–81. [doi: 10.1016/j.infsof.2015.08.006]

[18] Li L, Bissyandé TF, Papadakis M, Rasthofer S, Bartel A, Octeau D, Klein J, Traon L. Static analysis of android apps: A systematic literature review. Information and Software Technology, 2017, 88: 67–95. [doi: 10.1016/j.infsof.2017.04.001]

[19] Aho AV, Lam MS, Sethi R, Ullman JD. Compilers: Principles, Techniques, and Tools. 2nd ed., Boston: Addison Wesley, 2006.

© 中国科学院软件研究所 http://www.jos.org.cn

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

[21] Shivers O. Control-flow analysis of higher-order languages [Ph.D. Thesis]. Pittsburgh: Carnegie Mellon University, 1991.

[22] Grove D, DeFouw G, Dean J, Chambers C. Call graph construction in object-oriented languages. ACM SIGPLAN Notices, 1997, 32(10): 108–124. [doi: 10.1145/263700.264352]

[23] Milanova A, Rountev A, Ryder BG. Parameterized object sensitivity for points-to analysis for Java. ACM Trans. on Software Engineering and Methodology, 2005, 14(1): 1–41. [doi: 10.1145/1044834.1044835]

[24] Milanova A, Rountev A, Ryder BG. Parameterized object sensitivity for points-to and side-effect analyses for Java. ACM SIGSOFT Software Engineering Notes, 2002, 27(4): 1–11. [doi: 10.1145/566171.566174]

[25] Smaragdakis Y, Bravenboer M, Lhoták O. Pick your contexts well: Understanding object-sensitivity. ACM SIGPLAN Notices, 2011, 46 (1): 17–30. [doi: 10.1145/1925844.1926390]

[26] Lhoták O. Program analysis using binary decision diagrams [Ph.D. Thesis]. Montreal: McGill University, 2006.

[27] Lhoták O, Hendren L. Relations as an abstraction for BDD-based program analysis. ACM Trans. on Programming Languages and Systems, 2008, 30(4): 19. [doi: 10.1145/1377492.1377494]

[28] Lundberg J, Gutzmann T, Edvinsson M, Löwe W. Fast and precise points-to analysis. Information and Software Technology, 2009, 51(10): 1428–1439. [doi: 10.1016/j.infsof.2009.04.012]

[29] Liang DL, Pennings M, Harrold MJ. Evaluating the impact of context-sensitivity on Andersen's algorithm for Java programs. ACM SIGSOFT Software Engineering Notes, 2006, 31(1): 6–12. [doi: 10.1145/1108768.1108797]

[30] Lhoták O, Hendren L. Context-sensitive points-to analysis: Is it worth it? In: Proc. of the 15th Int'l Conf. on Compiler Construction. Vienna: Springer, 2006. 47–64. [doi: 10.1007/11688839_5]

[31] Lhoták O, Hendren L. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. ACM Trans. on Software Engineering and Methodology, 2008, 18(1): 3. [doi: 10.1145/1391984.1391987]

[32] Lundberg J, Löwe W. Points-to analysis: A fine-grained evaluation. Journal of Universal Computer Science, 2012, 18(20): 2851–2878. [doi: 10.3217/jucs-018-20-2851]

[33] Padhye R, Khedker UP. Interprocedural data flow analysis in soot using value contexts. In: Proc. of the 2nd ACM SIGPLAN Int'l Workshop on State of the Art in Java Program Analysis. Seattle: ACM, 2013. 31–36. [doi: 10.1145/2487568.2487569]

[34] Petrashko D, Ureche V, Lhoták O, Odersky M. Call graphs for languages with parametric polymorphism. ACM SIGPLAN Notices, 2016, 51(10): 394–409. [doi: 10.1145/3022671.2983991]

[35] Kastrinis G, Smaragdakis Y. Hybrid context-sensitivity for points-to analysis. ACM SIGPLAN Notices, 2013, 48(6): 423–434. [doi: 10.1145/2499370.2462191]

[36] Liang DL, Harrold MJ. Efficient points-to analysis for whole-program analysis. In: Proc. of the 7th ACM SIGSOFT Symp. on Foundations of Software Engineering. Toulouse: Springer, 1999. 199–215. [doi: 10.1007/3-540-48166-4_13]

[37] Lattner C, Lenharth A, Adve V. Making context-sensitive points-to analysis with heap cloning practical for the real world. ACM SIGPLAN Notices, 2007, 42(6): 278–289. [doi: 10.1145/1273442.1250766]

[38] O'Callahan R. Generalized aliasing as a basis for program analysis tools [Ph.D. Thesis]. Pittsburgh: Carnegie Mellon University, 2001.

[39] Feng Y, Wang XY, Dillig I, Dillig T. Bottom-up context-sensitive pointer analysis for Java. In: Proc. of the 13th Asian Symp. on Programming Languages and Systems. Springer, 2015. 465–484. [doi: 10.1007/978-3-319-26529-2_25]

[40] Xu GQ, Rountev A. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In: Proc. of the 2008 Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2008. 225–236. [doi: 10.1145/1390630.1390658]

[41] Li Q, Zhao JH, Li XD. Optimize context-sensitive Andersen-style points-To analysis by method summarization and cycle-elimination. In: Proc. of the 4th Int'l Symp. on Leveraging Applications of Formal Methods, Verification and Validation. Heraklion: Springer, 2010. 564–578. [doi: 10.1007/978-3-642-16558-0_46]

[42] Li Q, Tang E Y, Dai XF, Wang LZ, Zhao JH. Optimization of points-to analysis for Java. Ruan Jian Xue Bao/Journal of Software, 2011, 22(6): 1140−1154 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4025.htm [doi: 10.3724/SP.J.1001.2011.04025]

[43] Mangal R, Naik M, Yang H. A correspondence between two approaches to interprocedural analysis in the presence of join. In: Proc. of the 23rd European Symp. on Programming Languages and Systems. Grenoble: Springer, 2014. 513–533. [doi: 10.1007/978-3-642-54833-8_27]

[44] Kanvar V, Khedker UP. Heap abstractions for static analysis. ACM Computing Surveys, 2016, 49(2): 29. [doi: 10.1145/2931098]

[45] Lerch J, Hermann B, Bodden E, Mezini M. FlowTwist: Efficient context-sensitive inside-out taint analysis for large codebases. In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Hong Kong: ACM, 2014. 98–108. [doi: 10.1145/2635868.2635878]

© 中国科学院软件研究所 http://www.jos.org.cn

[46] Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Traon YL, Octeau D, McDaniel P. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. ACM SIGPLAN Notices, 2014, 49(6): 259–269. [doi: 10.1145/2666356.2594299]
[47] Späth J, Ali K, Bodden E. IDEal: Efficient and precise alias-aware dataflow analysis. Proc. of the ACM on Programming Languages, 2017, 1(OOPSLA): 99. [doi: 10.1145/3133923]
[48] Sridharan M, Chandra S, Dolby J, Fink SJ, Yahav E. Alias analysis for object-oriented programs. In: Clarke D, Noble J, Wrigstad T, eds. Aliasing in Object-oriented Programming. Types, Analysis and Verification. Berlin, Heidelberg: Springer, 2013. 196–232. [doi: 10.1007/978-3-642-36946-9_8]
[49] Wu DY, Gao DB, Deng RH, Chang RKC. When program analysis meets bytecode search: Targeted and efficient inter-procedural analysis of modern android apps in BackDroid. arXiv: 2005.11527, 2020.
[50] Reps T. Program analysis via graph reachability. Information and Software Technology, 1998, 40(11–12): 701–726. [doi: 10.1016/S0950-5849(98)00093-7]
[51] Reps T, Horwitz S, Sagiv M. Precise interprocedural dataflow analysis via graph reachability. In: Proc. of the 22nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. San Francisco: ACM, 1995. 49–61. [doi: 10.1145/199448.199462]
[52] Heintze N, Tardieu O. Demand-driven pointer analysis. ACM SIGPLAN Notices, 2001, 36(5): 24–34. [doi: 10.1145/381694.378802]
[53] Zheng X, Rugina R. Demand-driven alias analysis for C. ACM SIGPLAN Notices, 2008, 43(1): 197–208. [doi: 10.1145/1328897.1328464]
[54] Lam MS, Whaley J, Livshits VB, Martin MC, Avots D, Carbin C, Unkel C. Context-sensitive program analysis as database queries. In: Proc. of the 24th ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems. Baltimore: ACM, 2005. 1–12. [doi: 10.1145/1065167.1065169]
[55] Bravenboer M, Smaragdakis Y. Strictly declarative specification of sophisticated points-to analyses. ACM SIGPLAN Notices, 2009, 44(10): 243–262. [doi: 10.1145/1639949.1640108]
[56] Smaragdakis Y, Bravenboer M. Using datalog for fast and easy program analysis. In: Proc. of the 1st Int'l Datalog 2.0 Workshop. Oxford: Springer, 2010. 245–251. [doi: 10.1007/978-3-642-24206-9_14]
[57] Smaragdakis Y, Balatsouras G. Pointer analysis. Foundations and Trends in Programming Languages, 2015, 2(1): 1–69. [doi: 10.1561/2500000014]
[58] Naik M. Chord: A program analysis platform for Java. 2006.
[59] Petablox. https://github.com/petablox/petablox
[60] Zhu JW. Symbolic pointer analysis. In: Proc. of the 2002 IEEE/ACM Int'l Conf. on Computer-aided Design. San Jose: ACM, 2002. 150–157. [doi: 10.1145/774572.774594]
[61] Zhu JW, Calman S. Symbolic pointer analysis revisited. ACM SIGPLAN Notices, 2004, 39(6): 145–157. [doi: 10.1145/996893.996860]
[62] Whaley J, Lam M S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. ACM SIGPLAN Notices, 2004, 39(6): 131–144. [doi: 10.1145/996893.996859]
[63] Lhoták O, Curial S, Amaral JN. Using ZBDDs in points-to analysis. In: Proc. of the 20th Int'l Workshop on Languages and Compilers for Parallel Computing. Urbana: Springer, 2007. 338–352. [doi: 10.1007/978-3-540-85261-2_23]
[64] Lhoták O, Curial S, Amaral JN. Using XBDDs and ZBDDs in points-to analysis. Software: Practice and Experience, 2009, 39(2): 163–188. [doi: 10.1002/spe.895]
[65] Whaley J, Avots D, Carbin M, Lam MS. Using datalog with binary decision diagrams for program analysis. In: Proc. of the 3rd Asian Symp. on Programming Languages and Systems. Tsukuba: Springer, 2005. 97–118. [doi: 10.1007/11575467_8]
[66] Xiao X, Zhang C. Geometric encoding: Forging the high performance context sensitive points-to analysis for Java. In: Proc. of the 2011 Int'l Symp. on Software Testing and Analysis. Toronto: ACM, 2011. 188–198. [doi: 10.1145/2001420.2001443]
[67] Naeem NA, Lhoták O. Faster alias set analysis using summaries. In: Proc. of the 20th Int'l Conf. on Compiler Construction. Saarbrücken: Springer, 2011. 82–103. [doi: 10.1007/978-3-642-19861-8_6]
[68] Ali K, Lhoták O. Averroes: Whole-program analysis without the whole program. In: Proc. of the 27th European Conf. on Object-oriented Programming. Montpellier: Springer, 2013. 378–400. [doi: 10.1007/978-3-642-39038-8_16]
[69] Ali K, Lhoták O. Application-only call graph construction. In: Proc. of the 26th European Conf. on Object-Oriented Programming. Beijing: Springer, 2012. 688–712. [doi: 10.1007/978-3-642-31057-7_30]
[70] Kulkarni S, Mangal R, Zhang X, Naik M. Accelerating program analyses by cross-program training. ACM SIGPLAN Notices, 2016, 51(10): 359–377. [doi: 10.1145/3022671.2984023]
[71] Madhavan R, Ramalingam G, Vaswani K. Modular heap analysis for higher-order programs. In: Proc. of the 19th Int'l Static Analysis

Symp. Deauville: Springer, 2012. 370–387. [doi: 10.1007/978-3-642-33125-1_25]
[72] Tang H, Wang D, Xiong YF, Zhang LM, Wang XY, Zhang L. Conditional Dyck-CFL reachability analysis for complete and efficient library summarization. In: Proc. of the 26th European Symp. on Programming. Uppsala: Springer, 2017. 880–908. [doi: 10.1007/978-3-662-54434-1_33]
[73] Smaragdakis Y, Kastrinis G, Balatsouras G. Introspective analysis: Context-sensitivity, across the board. ACM SIGPLAN Notices, 2014, 49(6): 485–495. [doi: 10.1145/2666356.2594320]
[74] Hassanshahi B, Ramesh RK, Krishnan P, Scholz B, Lu Y. An efficient tunable selective points-to analysis for large codebases. In: Proc. of the 6th ACM SIGPLAN Int'l Workshop on State of the Art in Program Analysis. Barcelona: ACM, 2017. 13–18. [doi: 10.1145/3088515.3088519]
[75] Liang P, Tripp O, Naik M. Learning minimal abstractions. ACM SIGPLAN Notices, 2011, 46(1): 31–42. [doi: 10.1145/1925844.1926391]
[76] Jeong S, Jeon M, Cha S, Oh H. Data-driven context-sensitivity for points-to analysis. Proc. of the ACM on Programming Languages, 2017, 1(OOPSLA): 100. [doi: 10.1145/3133924]
[77] Jeon M, Jeong S, Oh H. Precise and scalable points-to analysis via data-driven context tunneling. Proc. of the ACM on Programming Languages, 2018, 2(OOPSLA): 140. [doi: 10.1145/3276510]
[78] Li Y, Tan T, Møller A, Smaragdakis Y. Precision-guided context sensitivity for pointer analysis. Proc. of the ACM on Programming Languages, 2018, 2(OOPSLA): 141. [doi: 10.1145/3276511]
[79] Li Y, Tan T, Møller A, Smaragdakis Y. Scalability-first pointer analysis with self-tuning context-sensitivity. In: Proc. of the 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Lake Buena Vista: ACM, 2018. 129–140. [doi: 10.1145/3236024.3236041]
[80] Tan T, Li Y, Xue JL. Making k-object-sensitive pointer analysis more precise with still k-limiting. In: Proc. of the 23rd Int'l Static Analysis Symp. Edinburgh: Springer, 2016. 489–510. [doi: 10.1007/978-3-662-53413-7_24]
[81] Sridharan M, Gopan D, Shan LX, Bodík R. Demand-driven points-to analysis for Java. ACM SIGPLAN Notices, 2005, 40(10): 59–76. [doi: 10.1145/1103845.1094817]
[82] Sridharan M, Bodík R. Refinement-based context-sensitive points-to analysis for Java. ACM SIGPLAN Notices, 2006, 41(6): 387–400. [doi: 10.1145/1133255.1134027]
[83] Xu GQ, Rountev A, Sridharan M. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In: Proc. of the 23rd European Conf. on Object-oriented Programming. Genoa: Springer, 2009. 98–122. [doi: 10.1007/978-3-642-03013-0_6]
[84] Yan DC, Xu GQ, Rountev A. Demand-driven context-sensitive alias analysis for Java. In: Proc. of the 2011 Int'l Symp. on Software Testing and Analysis. Toronto: ACM, 2011. 155–165. [doi: 10.1145/2001420.2001440]
[85] Shang L, Xie XW, Xue JL. On-demand dynamic summary-based points-to analysis. In: Proc. of the 10th Int'l Symp. on Code Generation and Optimization. San Jose: ACM, 2012. 264–274. [doi: 10.1145/2259016.2259050]
[86] Naeem NA, Lhoták O, Rodriguez J. Practical extensions to the IFDS algorithm. In: Proc. of the 19th Int'l Conf. on Compiler Construction. Paphos: Springer, 2010. 124–144. [doi: 10.1007/978-3-642-11970-5_8]
[87] Sagiv M, Reps T, Horwitz S. Precise interprocedural dataflow analysis with applications to constant propagation. Theoretical Computer Science, 1996, 167(1-2): 131–170. [doi: 10.1016/0304-3975(96)00072-2]
[88] Guyer SZ, Lin C. Client-driven pointer analysis. In: Proc. of the 10th Int'l Static Analysis Symp. San Diego: Springer, 2003. 214–236. [doi: 10.1007/3-540-44898-5_12]
[89] Zhang X, Mangal R, Grigore R, Naik M, Yang H. On abstraction refinement for program analyses in Datalog. ACM SIGPLAN Notices, 2014, 49(6): 239–248. [doi: 10.1145/2666356.2594327]
[90] Allen N, Scholz B, Krishnan P. Staged points-to analysis for large code bases. In: Proc. of the 24th Int'l Conf. on Compiler Construction. London: Springer, 2015. 131–150. [doi: 10.1007/978-3-662-46663-6_7]
[91] Rama GM, Komondoor R, Sharma H. Refinement in object-sensitivity points-to analysis via slicing. Proc. of the ACM on Programming Languages, 2018, 2(OOPSLA): 142. [doi: 10.1145/3276512]
[92] Sun CG, Midkiff S. Demand-driven refinement of points-to analysis. In: Proc. of the 41st Int'l Conf. on Software Engineering: Companion Proc. Montreal: IEEE, 2019. 264–265. [doi: 10.1109/ICSE-Companion.2019.00106]
[93] Liang P, Naik M. Scaling abstraction refinement via pruning. ACM SIGPLAN Notices, 2011, 46(6): 590–601. [doi: 10.1145/1993316.1993567]
[94] Tan T, Li Y, Xue JL. Efficient and precise points-to analysis: Modeling the heap by merging equivalent automata. ACM SIGPLAN

© 中国科学院软件研究所 http://www.jos.org.cn

Notices, 2017, 52(6): 278–291. [doi: 10.1145/3140587.3062360]

[95] Hirzel M, Von Dincklage D, Diwan A, Hind M. Fast online pointer analysis. ACM Trans. on Programming Languages and Systems, 2007, 29(2): 11. [doi: 10.1145/1216374.1216379]

[96] Shang L, Lu Y, Xue JL. Fast and precise points-to analysis with incremental CFL-reachability summarisation: Preliminary experience. In: Proc. of the 27th IEEE/ACM Int'l Conf. on Automated Software Engineering. Essen: ACM, 2012. 270–273. [doi: 10.1145/2351676.2351720]

[97] Lu Y, Shang L, Xie XW, Xue JL. An incremental points-to analysis with CFL-reachability. In: Proc. of the 22nd Int'l Conf. on Compiler Construction. Rome: Springer, 2013. 61–81. [doi: 10.1007/978-3-642-37051-9_4]

[98] Vivien F, Rinard M. Incrementalized pointer and escape analysis. In: Proc. of the ACM SIGPLAN 2001 Conf. on Programming Language Design and Implementation. Snowbird: ACM, 2001. 35–46. [doi: 10.1145/378795.378804]

[99] Whaley J, Rinard M. Compositional pointer and escape analysis for Java programs. ACM SIGPLAN Notices, 1999, 34(10): 187–206. [doi: 10.1145/320385.320400]

[100] Li X, Ogawa M. An ahead-of-time yet context-sensitive points-to analysis for Java. Electronic Notes in Theoretical Computer Science, 2009, 253(5): 31–46. [doi: 10.1016/j.entcs.2009.11.013]

[101] Li X, Ogawa M. Stacking-based context-sensitive points-to analysis for Java. In: Proc. of the 5th Int'l Haifa Verification Conf. on Hardware and Software: Verification and Testing. Haifa: Springer, 2009. 133–149.

[102] Späth J, Ali K, Bodden E. Context-, flow-, and field-sensitive data-flow analysis using synchronized Pushdown systems. Proc. of the ACM on Programming Languages, 2019, 3(POPL): 48. [doi: 10.1145/3290361]

[103] Li X, Ogawa M. Conditional weighted pushdown systems and applications. In: Proc. of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. Madrid: ACM, 2010. 141–150. [doi: 10.1145/1706356.1706382]

[104] Zhang QR, Su ZD. Context-sensitive data-dependence analysis via linear conjunctive language reachability. ACM SIGPLAN Notices, 2017, 52(1): 344–358. [doi: 10.1145/3093333.3009848]

[105] Wang K, Hussain A, Zuo ZQ, Xu GQ, Sani AA. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. ACM SIGOPS Operating Systems Review, 2017, 51(2): 389–404. [doi: 10.1145/3093315.3037744]

[106] Zhang QR, Lyu MR, Yuan H, Su ZD. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. ACM SIGPLAN Notices, 2013, 48(6): 435–446. [doi: 10.1145/2499370.2462159]

[107] Thiessen R, Lhoták O. Context transformations for pointer analysis. ACM SIGPLAN Notices, 2017, 52(6): 263–277. [doi: 10.1145/3140587.3062359]

[108] Hollingum N, Scholz B. Towards a scalable framework for context-free language reachability. In: Proc. of the 24th Int'l Conf. on Compiler Construction. London: Springer, 2015. 193–211. [doi: 10.1007/978-3-662-46663-6_10]

[109] Thakur M, Nandivada V K. Compare less, defer more: Scaling value-contexts based whole-program heap analyses. In: Proc. of the 28th Int'l Conf. on Compiler Construction. Washington: ACM, 2019. 135–146. [doi: 10.1145/3302516.3307359]

[110] Milanova A. Light context-sensitive points-to analysis for Java. In: Proc. of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. San Diego: ACM, 2007. 25–30. [doi: 10.1145/1251535.1251540]

[111] Méndez-Lojo M, Mathew A, Pingali K. Parallel inclusion-based points-to analysis. In: Proc. of the ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications. Reno: ACM, 2010. 428–443. [doi: 10.1145/1869459.1869495]

[112] Putta S, Nasre R. Parallel replication-based points-to analysis. In Proc. of the 21st Int'l Conf. on Compiler Construction. Tallinn: Springer, 2012. 61–80. [doi: 10.1007/978-3-642-28652-0_4]

[113] Su Y, Ye D, Xue JL, Liao XK. An efficient GPU implementation of inclusion-based pointer analysis. IEEE Trans. on Parallel and Distributed Systems, 2016, 27(2): 353–366. [doi: 10.1109/TPDS.2015.2397933]

[114] Mendez-Lojo M, Burtscher M, Pingali K. A GPU implementation of inclusion-based points-to analysis. In: Proc. of the 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. New Orleans: ACM, 2012. 107–116. [doi: 10.1145/2145816.2145831]

[115] Su Y, Ye D, Xue JL. Accelerating inclusion-based pointer analysis on heterogeneous CPU-GPU systems. In: Proc. of the 20th Int'l Conf. on High Performance Computing. Bengaluru: IEEE, 2013. 149–158. [doi: 10.1109/HiPC.2013.6799110]

[116] Smaragdakis Y, Balatsouras G, Kastrinis G. Set-based pre-processing for points-to analysis. In: Proc. of the 2013 ACM SIGPLAN Int'l Conf. on Object Oriented Programming Systems Languages & Applications. Indianapolis: Association for Computing Machinery, 2013. 253–270. [doi: 10.1145/2509136.2509524]

[117] Chen CM, Huo W, Yu HT, Feng XB. A survey of optimization technology of inclusion-based pointer analysis. Chinese Journal of Computers, 2011, 34(7): 1224–1238 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2011.01224]

© 中国科学院软件研究所 http://www.jos.org.cn

[118] Balatsouras G, Ferles K, Kastrinis G, Smaragdakis Y. A Datalog model of must-alias analysis. In: Proc. of the 6th ACM SIGPLAN Int'l Workshop on State of the Art in Program Analysis. Barcelona: ACM, 2017. 7–12. [doi: 10.1145/3088515.3088517]

[119] Deutsch A. Interprocedural may-alias analysis for pointers: Beyond *k*-limiting. ACM SIGPLAN Notices, 1994, 29(6): 230–241. [doi: 10.1145/773473.178263]

[120] Khedker U P, Sanyal A, Karkare A. Heap reference analysis using access graphs. ACM Trans. on Programming Languages and Systems, 2007, 30(1): 1–40. [doi: 10.1145/1290520.1290521]

[121] Kastrinis G, Balatsouras G, Ferles K, Prokopaki-Kostopoulou N, Smaragdakis Y. An efficient data structure for must-alias analysis. In: Proc. of the 27th Int'l Conf. on Compiler Construction. Vienna: ACM, 2018: 48–58. [doi: 10.1145/3178372.3179519]

[122] Landi W, Ryder BG, Zhang S. Interprocedural modification side effect analysis with pointer aliasing. In: Proc. of the ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation. Albuquerque: ACM, 1993. 56–67. [doi: 10.1145/155090.155096]

[123] Diwan A, McKinley KS, Moss JEB. Type-based alias analysis. ACM SIGPLAN Notices, 1998, 33(5): 106–117. [doi: 10.1145/277652.277670]

附中文参考文献:

[42] 李倩, 汤恩义, 戴雪峰, 王林章, 赵建华. Java指针指向分析优化. 软件学报, 2011, 22(6): 1140−1154. http://www.jos.org.cn/1000-9825/4025.htm [doi: 10.3724/SP.J.1001.2011.04025]

[117] 陈聪明, 霍玮, 于洪涛, 冯晓兵. 基于包含的指针分析优化技术综述. 计算机学报, 2011, 34(7): 1224–1238. [doi: 10.3724/SP.J.1016.2011.01224]

李昊峰(1994－), 男, 博士生, CCF 学生会员, 主要研究领域为程序分析.

曹立庆(1997－), 男, 博士生, CCF 学生会员, 主要研究领域为程序分析.

孟海宁(1995－), 女, 博士生, CCF 学生会员, 主要研究领域为程序分析.

李炼(1977－), 男, 博士, 研究员, 博士生导师, CCF 专业会员, 主要研究领域为程序分析, 软件安全.

郑恒杰(1996－), 男, 硕士, 主要研究领域为程序分析.