软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2011,22(6):1140−1154 [doi: 10.3724/SP.J.1001.2011.04025]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

Java 指针指向分析优化*

李 倩[1,2], 汤恩义[1,2], 戴雪峰[1,2], 王林章[1,2], 赵建华[1,2+]

[1](计算机软件新技术国家重点实验室(南京大学),江苏 南京 210093)

[2](南京大学 计算机科学与技术系,江苏 南京 210093)

Optimization of Points-to Analysis for Java

LI Qian[1,2], TANG En-Yi[1,2], DAI Xue-Feng[1,2], WANG Lin-Zhang[1,2], ZHAO Jian-Hua[1,2+]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210093, China)

[2](Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China)

+ Corresponding author: E-mail: zhaojh@nju.edu.cn

Li Q, Tang EY, Dai XF, Wang LZ, Zhao JH. Optimization of points-to analysis for Java. *Journal of Software*, 2011,22(6):1140−1154. http://www.jos.org.cn/1000-9825/4025.htm

Abstract: Points-to analysis mainly aims to attain the runtime points-to sets of program variables. This paper describes the design and implementation of an efficient Andersen-style, context-sensitive points-to analysis for Java code. The implementation supports language features such as inheritance, polymorphism, and field objects. The study tracks the fields of individual objects separately and makes the algorithm in field-sensitive style for aggregate objects. To improve the efficiency and scalability of the algorithm, this study employs two kinds of optimizations, nodes topology construction with concomitance on-line cycle detection and cycle elimination. Experiment results show that this algorithm can be used to compute precise points-to sets for large-scale Java programs.

Key words: points-to analysis; context-sensitive; field-sensitive; cycle elimination

摘 要: 指针指向分析的主要目的是静态地获取程序在运行时刻的指针指向信息.基于 Andersen 算法,设计了一种有效的、上下文敏感的指针指向分析算法,支持继承、字段对象等语言特性.不同对象的字段在算法中被分别处理,同时,算法对复合类型的对象实现了基于字段的处理.为了提高算法的效率和可扩展性,引入了两种优化方式:一种是结点间的拓扑排序以降低分析过程中的迭代次数;另一种是在线的环路侦测与消除,它与拓扑排序过程同步实现,有效地提高了处理效率.实验数据表明,该算法可以用来为较大规模的 Java 代码生成精确的指向关系集合.

关键词: 指针指向分析;上下文敏感;字段敏感;环路检测

中图法分类号: TP314 **文献标识码**: A

指针指向分析通过解析变量的读写信息,分析出变量或变量字段在程序运行时刻可能指向的数据对象集合(称为这些变量/字段的指向对象集合),得到被分析程序在运行时刻的指针指向信息.指向信息是编译器进行优化的基础,假设指向分析结果可以推导出某个变量 x 的精确类型信息,那么语句 *x.msg*()就可以使用静态过程

* 基金项目: 国家自然科学基金(90818022, 91018006, 61021062); 国家重点基础研究发展计划(973)(2009CB320702); 核高基项目(2009z01036-001-001-3)

收稿时间: 2010-07-10; 修改时间: 2011-03-29; 定稿时间: 2011-04-11

调用来实现,而不需要通过相对低效的动态消息分发过程来实现.很多软件工程相关的工具利用指向分析来提高工具的效率和精度.例如程序切片工具、基于数据流的测试等.程序分析工作如程序更改的波动分析[1]、变量的定值-使用分析等,可以利用指针指向信息来提高分析结果的精度.

然而,指针指向分析算法的设计是非常具有挑战性的.通过静态分析得到完整的指向信息已被证明是不可判定的[2,3],完整地对程序所有语言特性及程序结构进行描述及分析并不可行,因此,指针分析得到的结果只是某种程度上的近似结果.算法对程序结构的描述越精确,得到指向信息的精度就越高,但是算法的时间复杂性也相应提高.指针指向分析算法的精度一般可以从以下几个角度进行分类[4]:

- 流敏感度,即分析过程是否考虑控制流信息,如语句的顺序、分支结构等.流不敏感的分析算法忽略了语句的执行顺序,因此缺失了部分信息;流敏感的分析算法则根据程序的控制流图计算得到更加精确的指向关系.
- 上下文敏感度,即分析过程是否区分过程调用所处的上下文环境.在上下文不敏感的分析算法中,同一个过程在不同调用点的数据流信息是相同的;在上下文敏感的分析算法中,不同调用点上的数据流信息受到上下文环境的影响,通常是不同的.
- 对象敏感度,或者称为对象表示法.早期的分析算法[5,6]中,同一个类的所有对象被表示为单一结点.之后,对象的处理粒度细化了,许多算法使用对象实例化语句所在的行号来标识由这个语句实例化得到的全部对象.更进一步地,文献[7,8]为对象实例化语句也加入了上下文信息,被称为 Heap-clone 的分析算法;
- 字段敏感度,即分析过程是否对对象的不同字段进行描述,以区分对象字段的指向关系.字段不敏感的算法中,字段的指向关系与其所在的对象是统一处理的;而字段敏感算法中,对象的字段被看作是单独的数据结构,是独立的指向关系计算单位.文献[9]通过实验证明,field-sensitive 对于分析的精度与效率有非常大的影响.

一般来说,某个方面不敏感必然使得算法牺牲一部分精度,而某方面的敏感必然增加算法的复杂度,降低算法实现的效率.当前,指针分析研究工作面临的主要问题是,在保证算法精确度的同时平衡时间/空间上的消耗,构造可扩展的、高精度的指针指向分析算法.

1994 年,Andersen 在文献[10]中提出了一种基于包含的经典的 C 语言指针指向分析算法,这个算法被认为是最精确的流不敏感、上下文不敏感的指针指向分析算法[11].他将程序中的直接指向关系描述为变量与对象之间的约束关系集合,再通过求解约束关系集合的传递闭包,计算得到间接的指向关系,从而获得所有变量的、完整的指向关系集合.这种基于包含的思想被广泛应用在后续的指针指向分析工作中.

文献[12–16]提出的指针分析算法都是基于包含的算法,研究者从不同角度扩展了 Andersen 算法的分析精度与效率.比如,文献[12]将 Andersen 算法扩展到了 Java 语言中,文献[13]中提出的框架一定程度上解决了上下文敏感分析算法中带来的开销问题.文献[14]在文献[13]的基础上进一步优化,在过程内部加上了流敏感的分析精度.文献[15]使用 BDD 编码进行指针分析,不仅减少了存储指向集合的内存需求,也减少了了闭包计算的开销.而文献[16]提出的需求驱动的指针分析算法可以从用户关注的某一个程序点出发,逆向求解该程序点的指向关系,从另一个角度解决了分析的效率问题.

本文以 Andersen 算法为基础,设计了一种上下文敏感、流不敏感的 Java 指针指向分析算法.具体特点包括:

- 上下文敏感:变量在不同调用上下文中的指向关系是分别计算的.
- 对象字段敏感:为各个对象的字段变量分别计算指向关系.
- 数组以及集合类型指向关系的抽象处理:为通用类库中定义的集合类(collections)对象及数组建立统一的抽象模型来表示集合元素以及数组元素的指向关系.

基于包含的指针分析算法原始复杂度为 $O(n^3)$,其中,n 为结果指向图中的结点个数.这样的复杂度使得算法在应用上受到了较大的限制,为了解决效率上的缺陷,我们引入了一系列的优化技术.在指向关系集合的迭代计算过程中,我们通过拓扑排序来提高计算效率.在拓扑排序过程中同时进行环检测,以合并等价的结点,从而降

低指向图中的结点数目,提高算法效率.

本文第 1 节介绍 Java 指针指向分析算法的基本概念.第 2 节给出本算法的实现框架以及关键的优化技术.第 3 节是原型工具及实例分析.第 4 节是相关工作的比较.最后是对本文工作的总结.

1 Java 指针指向分析算法简介

早期的指针指向分析工作都是针对C语言进行设计的.C语言中存在的指针计算操作以及类型转换等语言特性,使得对 C 语言程序的指针分析工作相对困难.Java 语言中没有明确的指针表示,也不支持类似的指针运算操作,并且 Java 语言是类型安全的.这些特性使得在 Java 上的指针分析工作有更大的发展潜力.

Java 程序中的指针指向分析,实质就是计算各个变量在运行时刻可能指向的对象的集合(即指向对象集合).这项工作也可以从另一个角度进行描述:确定一个变量在运行时刻是否可能和其他变量共同指向同一个数据对象.因此,它也被称为指针别名分析.不管怎样描述这个问题,Java 语言的指针分析工作的任务是明确的,即计算变量与对象之间的指向关系.

C 和 Java 之间存在的区别,使得各自的分析算法在方法和细节上有所区别,但是基本思想是共通的.本文工作的基础 Andersen 算法就是为 C 语言设计的指针指向分析算法.我们采用了算法中提出的集合约束的基本理念,实现了对 Java 程序中指向关系的描述与计算.

接下来,将逐一介绍 Java 语言指针指向分析的一些基本概念.

1.1 Java语言指针指向分析简介

概括来说,Java 语言的指针指向分析旨在描述下面 3 个集合中的元素之间的关系:

(1) *R*,变量集合,包括程序中出现的全部变量.

(2) *O*,对象名字集合.很多算法都把程序中的对象实例化语句作为对象的名字,它代表了所有由这个语句生成的对象.

(3) *F*,对象字段的集合.指针分析的结果以指向图来描述,指向图的顶点就是上面描述的 3 个集合.程序中赋值语句产生的变量与对象直接指向关系用结点之间的指向边表示.如图 1 所示:

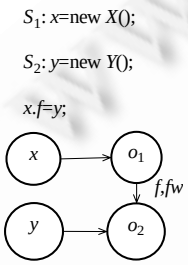


Fig.1 Example

图 1 指向图示例

- 图中的结点包括有变量结点 *x* 和 *y*,对象结点 o_1,o_2,分别用对象生成语句所在的行号来命名;
- 从变量结点 *x* 到对象结点 o_1 的指向边,表示变量 *x* 在运行时刻可能指向对象 o_1;
- 从对象字段结点 $o_1.f$ 到对象 o_2 的边,表示数据对象 o_1 的字段 *f* 可能指向对象 o_2.

假设 *r* 指向对象 *o*,且存在一个赋值语句 *w*=*r*,那么 *w* 与对象结点 *o* 之间存在间接指向关系.直接指向及间接指向构成了程序中完整的指向关系.通过计算指向集合的传递闭包,可以获得完整的指向关系集合.

1.2 不同精度下的指向分析

指向分析工作开始的基础是明确指向分析的精度,即指向分析中对各种程序结构的建模粒度.不同的分析精度决定了算法结果的准确性以及算法过程的效率,本文中的算法是字段敏感以及上下文敏感的.

字段敏感与否,区别在于对于类的成员变量的建模.字段不敏感的算法忽略了成员变量的指向关系;而字段敏感的算法则把每一个成员变量都看成一个独立的分析单位,所有对成员变量进行的读写操作反应为这个分析单位(在指向图中即为一个变量结点)的指向关系.表 1 说明了这两种精度对分析结果的影响.

从这个例子可以看出,字段不敏感算法的结果是不精确的,而字段敏感的算法通过在分析过程构造了更多的变量结点,得到了更高的分析精确度.

Table 1 Field-Sensitivity

表 1 字段敏感度

Code	Field-Insensitive	Field-Sensitive
s_1: $A\ a$=new A();	$a \supseteq \{o_1\}$	$a \supseteq \{o_1\}$
s_2: $a.f_1$=new F_1();	$a \supseteq \{o_2\}$	$a.f_1 \supseteq \{o_2\}$
s_3: $a.f_2$=new F_2();	$a \supseteq \{o_3\}$	$a.f_2 \supseteq \{o_3\}$
s_4: $b=a.f_1$	$b \supseteq a$	$b \supseteq a.f_1$
Analyze result	$b \supseteq \{o_1,o_2,o_3\}$	$b \supseteq \{o_2\}$

上下文敏感与否,在于是否区别不同的过程调用中由于不同调用序列而产生的当前全局指向信息.这些全局指向信息对过程内的各个局部变量的指向关系是有影响的,诸如实参、调用接收者以及全局变量等的指向关系都会影响局部变量指向关系的计算结果.上下文不敏感的分析将这些影响因素摒除在外,在指向图中,不同过程调用使用同样的指向信息,这样的处理过程无法满足高精度的需求.在上下文敏感的指针分析中,每一次的过程调用都会产生一个独立的过程副本,假设过程 *m* 中有局部变量 *v*,当过程 *m* 在某个点被调用时的上下文信息为 *con*,那么本次调用中局部变量 *v* 标记为 *con*:*v*,这个变量在结果指向图中拥有唯一的结点表示.同时,通过绑定过程的参数、返回值及 *this* 结点,过程内的数据流会接收过程调用时上下文信息带来的影响.我们的算法使用调用序列及接收对象来表示上下文.表 2 中的例子说明了这两种精度对分析结果的影响.

Table 2 Context-Sensitivity

表 2 上下文敏感度

Code	Context-Insensitive	Context-Sensitive
s_1: $A\ a_1$=new A();	$a_1 \supseteq \{o_1\}$	$a_1 \supseteq \{o_1\}$
s_2: $A\ a_2$=new A();	$a2 \supseteq \{o_2\}$	$a_2 \supseteq \{o_2\}$
s_3: $F\ f$=new F();	$f \supseteq \{o_3\}$	$f \supseteq \{o_3\}$
s_4: $a_1.setf(f)$;	$A.setf:arg \supseteq f$	$s_4\#setf:arg \supseteq f\ s_4\#setf:this \supseteq a_1$
class A {		
$F\ f$;		
$setf(F\ arg)\{this.f=f;\}$	$A.f \supseteq arg$	$this.f \supseteq arg$
}		
Analyze result:	$a_1.f \supseteq \{o_3\}, a_2.f \supseteq \{o_3\}$	$a_1.f \supseteq \{o_3\}$

不难发现,引入上下文敏感的指针分析得到的指向关系图描述的结果比上下文不敏感分析得到的指向图精确许多.然而,随着过程调用数量增多、层次深入,新增的上下文也会越来越多.由于同一过程在不同上下文中对应不同副本,指向图中的结点个数更是急剧增加.因此,求解上下文敏感的完全指向图时面临着更为严峻的效率问题,这也是本文工作旨在解决的最重要的问题.

2 算法描述

本文在 Andersen 算法的基础上设计了一种上下文敏感的 Java 指针指向分析算法.本节主要描述算法设计以及相关的优化技术.

算法整体分为 3 个阶段:首先,以 Eclipse 平台下的 JDT(Java development tooling)插件作为源码分析器,生成源文件相应的抽象语法树 AST;然后,以此为基础进行过程内的分析,生成每一个过程的摘要信息,包括过程内的局部指向图以及调用其他过程的调用点信息;最后,算法以主函数作为起始扩展过程进行过程间的分析,将待扩展过程的局部指向图复制入全局指向图中,通过绑定过程调用的接收者以及相应的形参和实参,完成此过程带有上下文信息的指向图.同时,以此过程内的调用点作为生成新上下文的起点.此时,根据当前的指针图可以确定调用点接收者的指向对象信息,进而确定实际调用的目标过程,不再需要保守的假设目标过程的接受者是声明类型及其所有子类,精化了扩展的过程,减少了不必要的操作.由于算法是流不敏感的,随着指向图的扩展,新产生的指向关系有可能使得已经处理过的调用点的接收者指向新的对象,从而增加新的待扩展目标过程.当不再有新的调用点需要扩展,也没有新的待扩展目标过程产生时,这个过程收敛.此时,算法可以获得最终完整的指向关系集合.

图 2 描述了算法的整体流程.

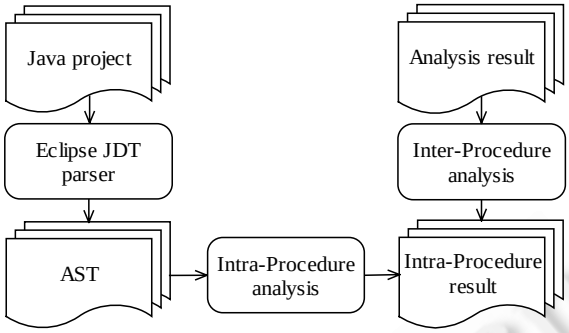


Fig.2 Skeleton of the algorithm

图 2 算法流程

上下文敏感的算法可以提高指向分析的精度,但同时也不可回避地带来了更庞大的指向图规模(同一个过程在不同的调用序列中都会产生一个单独的副本,从而使得局部变量在全局指向图中被多次复制与计算).计算各个结点的完全指向关系时进行的传递闭包运算的复杂度为 $O\{n^3\}$,结点个数的急剧增加使得我们将会面临更为严峻的效率问题以及算法的可扩展性问题.为此,我们在算法的基础上又提出了优化的措施,优化主要包括两个方面:

(1) 在算法进行指向关系扩展前,算法对过程内的指向集合进行预处理操作.过程内的指向关系被抽象为第 2.1 中定义的过程内的局部指向关系.每次扩展新的过程时,过程间的扩展算法调用这个预处理结果来生成此过程的全局副本.预处理操作中生成的局部指向图被复制到全局指向图中,联系当时的上下文环境,绑定相应的结点,计算当前上下文中的结点指向关系.引入预处理过程可以有效地避免在分析过程中反复分析过程内部的语句.

(2) 在扩展全局的指针指向关系时,我们通过拓扑排序的方式来提高迭代计算的效率.并且在拓扑排序的过程中同时进行环路检测,从而有效减少指向图中的变量结点数目.这项技术在第 2.3 中给出详细的说明.

2.1 过程内的指向关系预处理

过程内的指向关系预处理旨在为每个过程计算一个局部指向图,这个指向图描述了过程内局部变量、对象、参数、返回值以及全局变量之间的指向关系.

2.1.1 原子语句

在指针分析过程中,我们需要处理所有可能影响指针关系的语句.这些语句包括过程内所有的赋值语句、字段读/写语句以及过程调用语句(包括调用参数和返回值).为了方便分析,算法通过引入临时变量,将原程序中语句分解为一些形式简单的原子语句.这组原子语句包括:

- 赋值语句:$l=f$,即将 f 的值赋给 l;
- 字段写操作:$l.f=r$,即将 r 的值赋给 l 所指对象的 f 字段;
- 字段读操作:$l=r.f$,即将 r 所指对象的 f 字段中的值赋给 l;
- 对象实例化:$l=new\ C()$,即实例化类 C 的一个对象,并将对象句柄赋给 l;
- 方法调用语句:$l=r_0.m(r_1)$,即以 r_1 为实际参数向 r_0 所指向的对象发送消息 m,并把结果返回给 l.因为 Java 的多态特性,这个方法语句调用可能调用多个不同的过程;
- 返回值语句:$ret\ t$,将 t 作为过程的返回值返回.

通过一些近似处理,我们可以把所有的语句转换成为上面的原子语句的组合.比如:数组初始化可以被近似处理为对数组变量的赋值,强制类型转换语句可以被近似处理为赋值语句.

在把源代码分解成为这些原子语句之后,我们就可以将这些语句对指向关系的影响逐一表示在局部指向图中.

2.1.2 过程的摘要信息

过程内的指向分析旨在获得关于该过程的一个摘要信息.这个摘要信息包括两个部分:过程的调用点集合和局部指向关系.

过程的调用点集合对应于过程中的所有过程调用语句.调用点集合中包含了各个调用点的消息接收者、实际参数等信息.

局部指向关系图是过程中的所有原子语句语义的抽象表示.局部指向图的结点集合包括变量结点(形式参数和返回值等也被当作变量结点处理)和对象结点,结点之间有 4 种类型的有向边,包括变量之间的包含关系边、字段读指向边和字段写指向边以及变量到对象的指向边.

我们在对单个过程进行分析时,只考虑局部变量与这个过程内创建的对象之间的指向关系.在其他过程中创建的对象可能通过参数或者全局变量传递到过程内部,被局部变量指向.同时,过程内创建的对象也可能通过返回值或者全局变量被其他过程内定义的变量指向.我们在局部指向图中通过记录变量与变量之间的指向关系,间接地记录这种潜在的传递关系.算法在接下来的全局分析的过程中,通过绑定形参/实参、返回值/接收变量之间的关系,将局部变量与过程外的对象之间的指向关系传递进来.

直观地讲,生成局部指向图的过程就是对过程内所有的原子语句逐一分析的过程,具体的操作包括:

- 赋值语句 $l=r$:添加变量结点 l 到变量结点 r 的包含关系边,表示变量 l 的指向集合包含了变量 r 的指向集合;
- 字段写操作 $l.f=r$:添加变量结点 l 到变量结点 r 的字段写边,表示对于所有被变量 l 指向的对象 o,o 的字段 f 的指向集合应包含变量 r 的指向集合;
- 字段读操作 $l=r.f$:添加变量结点 l 到变量结点 r 的字段写边,表示变量 l 的指向集合应添加上所有被变量 r 指向的对象的字段 f 的指向集合;
- 对象创建 l=new C():添加变量结点 l 到对象 o 的指向边,表示变量 l 的指向集合中包含对象 o;
- 方法调用语句 $l=r_0.m(r_1,\ldots,r_n)$:生成过程 m 的一个调用点.同时,算法添加变量结点 l 到结点 *ret* 的包含关系边,以及从形式参数到相应的实际参数的包含关系边;
- 返回值语句 *ret t*:添加变量结点 t 与过程的抽象返回值 *ret* 结点之间的包含关系边,表示程序的返回值的指向集合中包含了 t 的指向集合.

我们使用函数 δ:$ptGraph\times stat\rightarrow ptGraph$ 形式化地表示处理各个原子语句的规则.$\delta(G,s)=G'$ 表示如果当前局部指向图为 G,再根据语句 s 的语义进行扩展后得到新的指向图 G'.传统的扩展过程中[17],每个原子语句的作用都直接反映为变量与对象之间的指向关系.在定义的局部指向关系预处理方法中,我们引入了变量与变量之间的指向关系.在全局的指针分析阶段,这些关系将被用于推导变量和对象之间的关系.图 3 描述了文献[17]与我们的处理方法之间的区别.($l.f,r$)和($l,r.f$)分别表示从 l 到 r 的字段写和字段读边.

图 4(b)描述了图 4(a)给出的示例程序中过程 *chaneYF*(*F* new*f*)的局部变量指向图,其结点包括抽象结点 *this* 和 *ret*,分别表示调用接收者和调用的返回值;变量结点 y_1,y_2 和 y_3;对象结点 o_3 以及调用点 cs_5 相关的 3 个抽象结点:表示消息接收者的 cs_5:*this*、表示调用返回值的 cs_5:*ret* 和调用形式化参数的 cs_5:*arg*.

我们按照上面描述的方法向局部指向图中添加不同的边,逐句说明局部指针指向图的计算过程.

语句 3 添加 y_1 到 o_3 的包含边,语句 4 添加 y_2 与 *this* 结点之间标号为 *fr* 的字段读边.与文献[14]中提到的一般的处理方式不同,我们此时并不考虑过程接收者 *this* 的指向关系是什么,以及它的指向集合有哪几个元素,只是单纯地把这种潜在的关系用字段读边记录下来,这使得局部处理过程可以比较迅速地完成.语句 5 对应于一个调用点 cs_5,同时添加了 y_3 与调用返回值之间的边以及 new *f* 与调用形参之间的包含关系边,语句 6 添加了 *this* 结点与 y_1 间的标号为 *fw* 的字段写边,语句 7 则添加了返回值与 l_2 之间的包含关系边.至此,所有语句处理完毕,我们得到了过程 *changyYF* 的局部指向图.

$\delta(G,l=\text{new } C)=G\cup\{(l,o)\}$
$\delta(G,l=r)=G\cup\{(l,o_i)|o_i\in Pt(G,r)\}$
$\delta(G,l.f=r)=G\cup\{(\langle o_i,f\rangle,o_j)|o_i\in Pt(G,l)\wedge o_j\in Pt(G,r)\}$
$\delta(G,l=r.f)=G\cup\{(l,o_i)|o_j\in Pt(G,r)\wedge o_i\in Pt(G,\langle o_j,f\rangle)\}$
$\delta(G,l=r_0.m(r_1,...,r_n))=G\cup\{resolve(G,m,o_i,r_1,...,r_n,l)|o_i\in Pt(G,r_0)\}$
$resolve(G,m,o_i,r_1,...,r_n,l)=$
let $m_j(p_0,p_1,...,p_n,ret_j)=dispatch(o_i,m)$ in
$\{(p_0,o_i)\}\cup\delta(G,l=ret_j)$

(a) Algorithm in Ref.[17]
(a) 文献[17]的算法

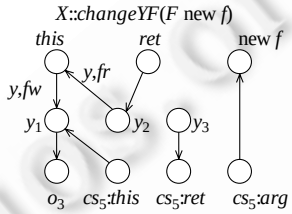
$\delta(G,l=\text{new } C)=G\cup\{(l,o)\}$
and $G.addcs(cs(C(\cdot)))$
$\delta(G,l=r)=G\cup\{(l,r)\}$
$\delta(G,l.f=r)=G\cup\{(l.f,r)\}$
$\delta(G,l=r.\delta)=G\cup\{(l,r.f)\}$
$\delta(G,l=r_0.m(r_1,...,r_n))=G\cup\{(l,ret)$
and $G.addcs(cs(m,l,r_0,r_1,...,r_n))$
$\delta(G,\text{return } t)=G\cup\{(t,ret)\}$

(b) Our algorithm
(b) 我们的算法

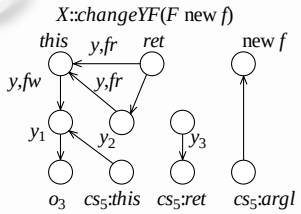
Fig.3 Points-to effective of program atomic statement
图 3 程序原子语句的指向效应

```
class X{
1    Y y;
2    Y changeYF(F new f){
3        Y y1=new Y();
4        Y y2=y;
5        Y y3=y1.setF(new f);
6        y=y1;
7        return y2;
8      }
9    ...
10 }
Class F{...}
class Y{
     F f;
     Y setF(F new f){
         f=new f;
         return this;}
     }
```

(a) A piece of program
(a) 代码示例



(b) Local graph of *changeYF*
(b) 过程 *changeYF* 的局部指向图



(c) Local graph after propagation
(c) 扩展后的局部指向图

Fig.4 Local point-to graph
图 4 局部指向图

2.1.3 局部指向图进一步处理

根据上述方法构造得到的图,抽象地表示了过程中的各个结点之间的关系.在全局指针分析算法中,每次扩展调用点就会把局部指向图拷贝到全局指向图中,并根据这些边来计算最终的指针指向关系.为了提高效率,我们对局部指向图进行进一步的预处理.比如,变量 *ret* 与变量 y_2 之前有包含关系,而 y_2 到 *this* 有标号为(*y*,*fr*)的字段读的关系,那么我们会添加一条从 *ret* 到 *this* 的标号为(*y*,*fr*)的字段读边.这个操作表示对于 *this* 所指向的任意对象 *o*,如果 *o*.*f* 指向某个对象 *o*′,那么 *ret* 也会指向对象 *o*′.这样做可以避免在构建全局指向图时多次处理 *u* 到 *w* 之间的关系,从而提高了效率.我们可以在图 4(c)中看到预处理的结果.

2.1.4 局部指向图在分析过程中的用法

指向分析的目标是计算各个变量在不同上下文中可能指向的对象集合,过程摘要描述了过程内与此过程相关的所有变量之间的指向关系,以及这些变量与此过程将会调用的其他过程的变量之间的联系.过程间分析使用摘要中描述的结点指向关系生成全局副本,同时将摘要中描述的跨过程的变量联系反映在全局指向图中.使用过程摘要进行全局扩展,不仅避免了在过程间分析时对同一个方法的代码进行多次分析,也使得扩展过程更为简洁.

图 5 描述了 m_1,m_2 的局部指向图在过程间分析中的使用.首先,算法在全局指向图中为局部指向图中的各个结点生成拷贝,这些拷贝结点之间的关系和他们在局部指向图中的关系相同.同时,还需要建立这些结点与过程

外结点之间可能的指向关系.为此,我们需要将过程的 *this* 结点和消息接收者、过程的形式参数和调用点的实际参数、过程的返回值和调用点接收返回值的变量分别联系起来即可.如图 5 所示,过程 m_1 中存在调用点 *cs*,相应的语句为 $v_4=v_4.m_2(v_3)$,我们为此语句生成一个调用点的描述,包括有调用方法的名称 m_2,调用的接收者结点 *rec* 指向结点 v_4,而结点 v_4 指向调用返回结点 *ret*.在扩展 m_2 的全局副本时,m_2 的 *this* 结点即为 m_1 中此调用的接收结点 *rec*,因此,m_2:*this* 指向 m_1:v_4,而 m_2 的返回值会返回给调用返回结点 *ret*.所以,m_1:v_4 又指向 m_2:*ret*.

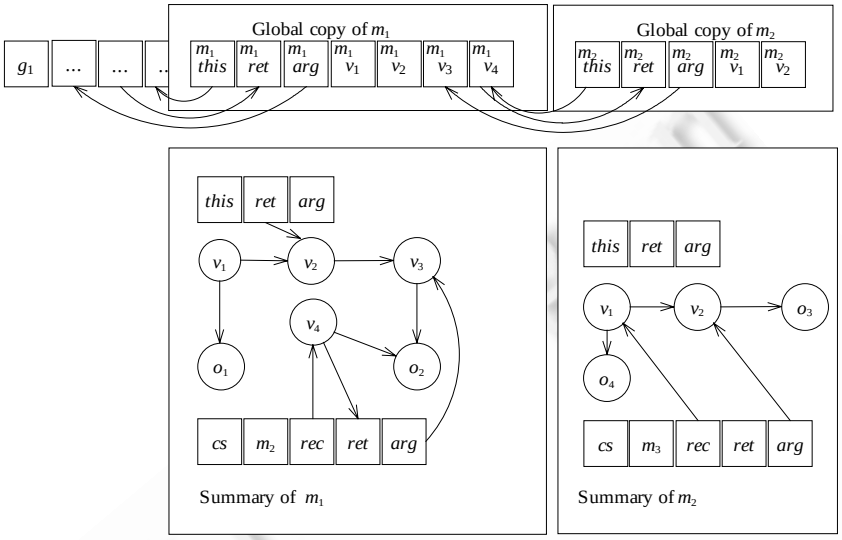


Fig.5 Use of method summary

图 5 过程摘要的使用

这里要注意的是,调用点定义中说明的 m_2 只是一个名字为 m_2 的虚调用,当我们在进行全局分析的时候,我们需要根据当前上下文中接收者变量 v_4 的指向对象集合来判断这个虚调用在运行过程中被调用的目标过程.因为 Java 的多态性,这个虚调用有可能有多个目标,每个目标过程都需要在全局图中进行扩展.

2.2 过程间指针分析

过程间指针分析以过程内指针分析获得的局部摘要信息为扩展的基础,以 *main* 函数为扩展起点,根据过程调用序列逐步迭代生成完整的过程间指针指向图.

2.2.1 过程间指针指向分析的基本流程

算法的目标是构造一个全局的指针指向关系图,使之反映了上下文敏感的指针指向关系.算法在总体上采用 *worklist* 的方式进行过程间指针分析工作.算法将程序运行的起始点(例如工程的 *main* 函数)作为顶层的上下文、以 *main* 函数作为初始的 *worklist*.随后,算法对于 *worklist* 中的每个调用点,根据已有的指向信息计算出可能被它调用的过程,然后向全局指向图中添加这些过程对应的上下文.算法将这个过程的局部指向关系图按照第 2.1.4 节中描述的方法复制到全局的指向关系结点图中,并进一步计算出新的指向关系.对于被加入的每个过程,该过程中的调用点(加上了上下文前缀之后)又被加入到 *worklist* 中.算法的基本过程如图 6 所示.

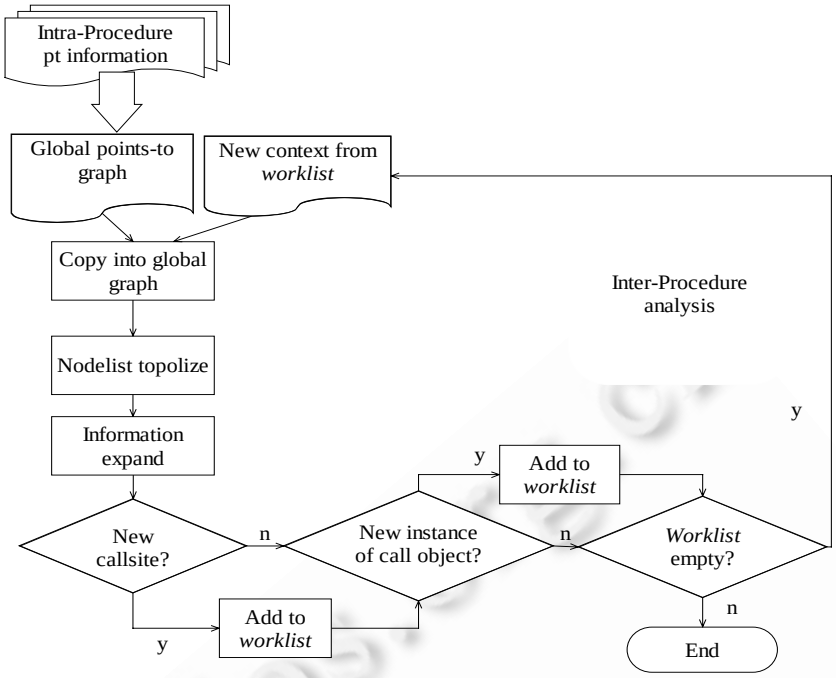


Fig.6 Inter procedure analysis

图 6 过程间分析

2.2.2 过程间指针指向分析的迭代过程

整个算法的处理过程是一个不断迭代的过程.算法不停地根据 *worklist* 中的调用点来扩展新的上下文;每一轮上下文扩展之后,算法通过一个内层迭代过程来更新全局指向图中结点之间的指向关系.在这个内层迭代过程的每一轮迭代执行如下的两个步骤,直到既没有新的边被加入到结点中,也没有结点的指向对象集合被修改为止.

(1) 对于字段读/写边的处理

(a) 每一条从结点 n 到 n'的标号为 f 的字段读边表示对于 n'所指向的每个对象 o,n 将指向 $o.f$ 所指向的每个对象.因此,算法对 n'的指向对象集合中的每个对象 o,添加一条从 n 到 $o.f$ 的包含关系边;

(b) 每一条从结点 n 到 n'的标号为 f 的字段写边表示对于 n 所指向的每个对象 o,$o.f$ 将指向 n'所指向的每个对象.因此,算法对于 n 的指向对象集合中的每个对象 o,添加一条从 $o.f$ 到 n'包含关系边.

(2) 对于指向边的处理

从结点 n 到结点 n'的包含关系边表示 n'的指向对象集合被 n 的指向对象集合所包含.因此,算法需要不断地处理每一条指向边,将其目标结点的指向对象集合添加到它的源结点的指向对象集合中.步骤 2 会一直执行到所有结点的指向对象集合都不再改变为止,同一指向边会被执行多次.

因为步骤 2 的处理会导致一个结点指向更多的对象,从而导致步骤 1 需要加入更多的包含关系边.因此,在每一轮增加扩展上下文后,我们需要反复执行步骤 1 和步骤 2,直到边和指向对象集合都不再增加为止.

2.3 算法优化

在第 2.2 节描述的算法中,大部分的 CPU 时间花费在内层迭代的步骤 2 中.因此,我们对步骤 2 的处理过程进行了优化.

优化的主要思想包括如下两点:

(1) 如果指向图中存在包含关系边形成的环路 $n_0 \rightarrow n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_0$,因为包含关系边表示了指向对象集合

之间的包含关系,因此这些结点的指向对象集合是相同的.即,这些结点在指向关系的范畴内是等价的.在我们的分析中,可以把这些结点合并成为一个代表结点,消除指向图中的环路.去环操作不仅降低了指向图的结点规模,而且减少了迭代的次数,有效地提高了算法的效率.如果把所有的环路都合并成一个结点,那么指向图的结点和包含关系边就形成了一个有向无环图.

(2) 在步骤 2 的迭代过程中,对各个结点的处理顺序会影响到迭代收敛的速度.假设指向图中的结点 n 和结点 n'代表变量或者对象字段,且从 n 到 n'存在一条指向边.如果我们在一次迭代中先处理 n 再处理 n',那么首先根据 n'当时的指向对象集合来计算 n 的指向对象集合;接下来处理结点 n'时可能有新的对象加入到 n'的指向对象集合中,导致我们必须在下一次迭代时对 n 的指向对象集合再次进行更新.但是如果我们先处理 n'再处理 n,那么我们在一次迭代中就可以处理完两个结点的指向对象集合.合并环路之后的指向图是一个有向无环图,如果把这些结点进行拓扑排序,算法只需要一次迭代就可以完成步骤 2.因此,通过拓扑排序,我们可以把第 2.2.2 节中的步骤 2 的迭代过程简化为对各个结点的一次遍历.

从第 3 节的实验结果中可以看出,指向图中这样的环是普遍存在的;同时,通过拓扑排序,使得我们在进行全局分析时,仅仅需要迭代一次就可以完成计算指向关系的传递闭包.

我们给出算法 1 来同时实现环路合并和拓扑排序,它是对拓扑排序算法的改进,在使用深度优先拓扑排序的过程中检测图中是否存在环路.如果发现环的存在,它就进行结点合并处理,并继续进行拓扑排序,最后得到一个合并了所有环的图以及图中结点的拓扑排序.

算法 1. TopoSortAndCircleCombine.

输入:图 G,这里我们只考虑包含关系边;

输出:合并环路之后的图以及图中结点的拓扑排序.

全局变量:

T:记录已经被拓扑排序的结点

trace:记录当前的搜索路径

```
算法:
将所有的结点标记为 unvisited;
while (图中存在 unvisited 结点)
    令 n0 为一个标记为 unvisited 的结点;
    将 n0 加入到 trace 中;
    while (trace 非空)
        n=trace 的最后一个结点
        if (n 的所有后继都已经标记为 visited)      //n 的所有后继或者已被合并,或者已经排序.
            将 n 加入到 T 的头部;                   //将 n 排序,加入 T
            将 n 标记为 visited;
            将 n 从 trace 中删除;                   //回溯
        else
            令 n′为 n 的某个标记为 unvisited 的后继;
            将 n′标记为 visited;
            if (n′是 trace 的第 k 个结点)           //形成环路
                n″=CircleCombined(k,n′);            //合并
                将 n″加入到 trace 的尾部;
            else
                将 n′加入到 trace 中;
            endif
        endif
    end while
end while
CircleCombined(k,n′)
{
    将 n′,也就是 trace 的第 k 个结点选择为这个环中各个结点的代表;
```

把环上的所有结点从 trace 中删除;
将环中的所有其他结点设置为 visited;
将这些结点上的所有边拷贝到 n'上;
return n';
}

假设图 $G=\langle N,E\rangle$描述了指向图中的包含关系,其中:N是指向图中变量结点和对象字段结点的集合;E是指向图中包含关系边的集合,集合 E 中的元素 $e_i=\langle n,n'\rangle$表示结点 n 和 n'之间的指向集合存在包含关系.

算法 1 的基本框架是一个深度优先搜索算法,它使用了两个全局变量 T 和 *trace* 分别记录已经排好序的结点和搜索过程中的当前路径.算法的拓扑排序过程是从后端开始的,也就是说,如果一个结点的所有后继都已经在 T 中了,算法就可以把这个结点加入 T 中.算法按照深度优先的方法遍历各个结点:

(1) 如果某个结点的所有后继都已经被遍历过,那么它的所有后继已经被加入到 T 中;

(2) 如果某个结点出现在 *trace* 中,算法就找到了一个环路.按照前面的讨论,这个环路可以被合并成为一个结点.我们从这个环路中选出一个结点作为这些结点的代表,然后把其他结点上的各种边都相应地加入到这个代表结点上.然后从这个代表结点继续搜索.

请注意,当我们合并环路的时候会选择一个代表结点,与环路上各个结点相关的各种类型的边都会被合并到这个代表结点上.

如果不考虑环路合并,这个算法的时间复杂性是线性的.因此,我们可以高效地完成拓扑排序.而这个拓扑排序可以把步骤 2 的迭代过程约简为对结点的单次遍历.因此,在步骤 2 之前先执行这个算法可以有效地提高算法的效率.算法中的环路合并所需要的时间代价相对较高,但是环路合并可以有效地降低指向图的结点数目.因此,环路合并付出的代价小于结点合并带来的效率提升.

2.4 集合对象的处理

Java 程序中大量用到了 Java 类库中的集合类(collections),比如 set,list,map 等,而对于集合类中元素的操作引起的指向关系是传统指针指向分析中的一个难点.由于静态分析无法明确定位进行操作的目标元素,进而无法确定指向关系发生的对象,从而使得单纯从源码进行分析来获取可能的指向关系变得相对困难;另一方面,虽然这些集合类在定义和操作方法上表现各异,但是包含的基本操作大体类似.

针对这个问题,我们对集合对象进行了抽象处理,将集合的所有元素抽象为集合对象的一个字段;所有对集合元素的操作都看作是对这个字段进行的操作.因为在使用集合类时,集合中的元素通常具有相同的指向关系,因此,这种抽象化仍在一定程度上保证了程序分析的精度.这个做法的好处是,指针分析程序可以避免处理这些集合类的复杂的内部数据结构,提高了效率.

我们的算法并没有直接去分析类库中集合类的源码,而是将集合类抽象为只有一个字段的类.所有对集合对象的赋值操作被抽象为对这个特定字段的操作.下面以 Set 类为例来说明这个抽象过程.其他集合类的抽象处理是相似的.

Set 类中与指向关系相关的基本操作包括 *add*(*Object obj*),*addAll*(*Collection c*),*remove*(*Object obj*)等.因为我们的分析是流不敏感的,*remove* 操作不会影响分析的结果,因此只需要考虑 *add* 和 *addAll* 两个操作.我们为 Set 类设置特殊字段 *ele*,对于声明为 Set 类型的变量 *s*,我们的算法在预处理过程中把语句 *s.add*(obj_1)和 *s.addAll*(*col*)分别转换为语句 $s.ele=obj_1$ 和 *s.ele=col.ele*.

这样的抽象使得我们比较容易处理这些集合类的子类.比如当处理 Set 类的用户自定义子类 MySet 时,MySet 类中将继承我们加入的特殊字段 *ele*;而 MySet 的例程中调用 *supper*::*add* 时,会被自动转化为对 *ele* 的赋值.

3 工具实现与实验结果

我们在 Eclipse 平台上实现了本文中描述的算法.这个实现是 Eclipse 平台下的一个插件工程,输入 Java 工程的源代码,给定相应的 *main* 入口,工具就可以分析得到相应的指针指向关系结果.

3.1 工具实现

工具的整体结构包括有 3 个部分:

- 基本信息收集.通过对源代码的扫描,收集保留了代码中的各种信息.例如类的信息包括有全局标识的名字、字段列表、过程声明列表、继承关系等.而过程的信息包括有全局标识的名字、参数类型、局部变量列表等.工具先使用 JDT 包对源程序进行分析得到抽象语法树,然后对抽象语法树进行分析得到这些信息.所有的信息保留在相应的数据结构中;
- 局部指向图生成.算法对 jdt 生成的抽象语法树进行扫描,将原来的程序语句分解为形如第 2.1.1 节中的原子语句.然后,工具按照第 2.1 节中的方法,根据这些原子语句构造出第 2.1.2 节中定义的局部指向图;
- 根据指定的 *main* 入口,工具根据第 2.2 节、第 2.3 节中描述的算法逐步构造出全局的指针指向图.

3.2 实验结果及分析

我们在一台 PC 机上进行实验,机器的基本配置为 2.20GHz,2.19GHz 的 Intel 双核,3.25Gb 内存.实验的环境是在 WindowsXP 系统下,Eclipse3.5 平台.我们使用的 jdk 是 1.5 版本.所有实验分配的内存都在 512MB 之内.

实验的对象是 SPECjvm2008 标准族.SPECjvm2008 是一个开源的标准族,所包括的十几个标准可以用来对 Java 虚拟机做相关的性能评测.SPECjvm 一共包括 32 个包,129 个 Java 文件,总共 19 578 行代码.其中有 10 个标准满足我们的实验需求,且过程调用层次至少 5 层(不包括对库函数的调用),我们以这 10 个标准为实验的对象进行了指针指向分析实验.表 3 中给出了这 10 个标准的基本描述.

Table 3 SPECjvm summary

表 3 SPECjvm 概况

Bemchmark	Call depth	Benchmark	Call depth
Compress	8+	scimark.lu	8+
crypto.aes	5+	scimark.Monte_carlo	8+
crypto.signverify	5+	scimark.sor	5+
MPEGaudio	7+	scimark.sparse	8+
scimark.fft	10+	xml.validation	6+

工具的第 1 步和第 2 步是在 SPECjvm 整体工程上进行的,工程中所有包和文件都进行源码级的扫描及信息收集,同时为所有的过程生成局部指向关系图.第 1 步和第 2 步的平均花费时间大约是 3s,这一部分工作可以离线操作,表 2 中的实验结果显示的时间不并包括这一部分.

从第 3 步开始,即是从每个标准的 *main* 过程开始,沿着过程调用的轨迹进行指针分析,结果见表 4.

Table 4 Analyze result

表 4 分析结果

Benchmark	Size			Time (ms)		Iteration (max)		Circle	
	Class	Method	LOC	Opt	No-Opt	Opt	No-Opt	Num	Avg. size
Compress	77	826	10 479	42	47	1	3	6	2.97
crypto.aes	66	760	9 867	142	156	1	3	90	3.0
crypto.signverify	66	760	9 841	120	125	1	3	115	3.02
MPEGaudio	66	757	9 708	40	47	1	2	9	3.02
scimark.fft	71	803	10 581	85	78	1	2	46	2.98
scimark.Monte_carlo	70	794	10 425	43	47	1	2	6	2.98
scimark.lu	70	804	10 651	60	63	1	3	43	2.92
scimark.sor	71	796	10 428	70	78	1	2	21	2.92
Sparse	70	796	10 473	57	63	1	3	24	2.96
Validation	67	770	9 910	55	62	1	2	17	3.0

表 4 的前 3 列概括说明了每个被分析的标准的规模,包括所涉及的类数量、方法数量以及代码行数.我们的算法只分析从入口可以到达的代码,但并不包括对库函数的调用,对集合类型相关操作也被抽象为相应的指向关系.

接下来的两列数字分别是经过优化的算法所使用的时间以及没有使用第 2.3 节中提出的优化技术时分析所使用的时间.再下来的两列数字分别是经过优化和未经过优化算法在计算传递闭包时所需要的迭代次数.经

过优化,迭代过程有效地减少到了一次,这也是算法减少时间的一个重要因素.最后两列记录的是优化过程中算法一共消除的环的个数以及平均环的大小.我们可以看到,指向图中的环是普遍存在的,平均大小为 3.如果环的个数为 n,环的平均大小为 m,那么可以减少结点个数为 $n\times(m-1)$.

有一点要特别说明,由于在环检测工作中需要对指向图进行深度优先遍历,同时在环消除的过程中涉及了环中结点的指向集合的合并操作,操作的开销与指向集合的大小有关.这一步的开销是制约优化的结果关键,如果环合并的开销不能抵销带来的效益,优化的表现有可能会不明显.例如在标准 scimark.fft 这个实验中,时间开销反而更多了.但是在大多数的情况下,本文提出的优化技术可以有效地提高指针指向分析的效率.

表 5 给出了工具在 specjvm98 标准包上的运行结果,表中数据为文献[14]中给出的实验结果.可以很明显地看到:在本文的算法框架下,我们的过程间分析时间只有不到 0.1s;而分析前可以进行离线的预处理过程,得到所有的过程摘要,这部分的时间也只要 2s 左右.即使两项时间消耗合并,也优于文献[14]的结果.本文的算法表现优异的原因有两个方面:

- Whaley 的算法中没有对方法摘要做进一步的处理,只反映了最代码中最原始的读写关系.全局扩展时,通过结点在指向关系图中的可达性计算来获取最终的指向关系集合,这个过程可能会执行多次.通过第 2.1.3 节的预处理,我们扩展了方法中各个结点的指向集合(即可达集合),在全局扩展中可以多次使用到这个结果,减少了冗余的操作;
- Whaley 的算法中,在检查结点的可达集合时进行环路的检测和消除.我们的算法中不仅进行了环路的消除,还可以获取结点的拓扑排序关系来指导指向关系的传递计算过程.使用这个排序来进行扩展,可以极大地减少扩展时所需要的迭代次数.

Table 5 Related work

表 5 相关工作比较

Benchmark	Our algorithm (s)			J. Whaley's (s)
	Method summary	Inter-procedure	Total	
Check	1.796	0.094	2.89	14
Compress	1.875	0.048	1.923	5
db	1.563	0.052	1.615	5
Raytrace	2.776	0.078	2.854	4

4 相关工作

指针指向分析工作是软件工程中许多开发工具的基础.指针指向分析工作基本可以分为两大流派,或者是基于 Andersen 提出的基于包含关系的算法进行扩展,或者是基于 Steensgaards[18]提出的 unification system 进行扩展,如文献[19,20].Anderson 和 Steensgaards 的算法都是流不敏感且上下文不敏感的算法,虽然 Steensgaards 算法中把存在赋值关系的两个指针直接做等价处理,使得算法在效率上比 Andersen 算法高出很多,但是在分析精度了也牺牲很多.

算法设计工作百家争鸣,针对不同算法的评估工作也是指针分析领域的一个发展重心.文献[21]把几种不同分析精度的代表性算法应用到客户分析中(即使用指针指向关系信息的其他代码分析工作,如定义可达性分析、变量活性分析等),评估了各个算法的性能.实验结果显示,Andersen 算法在精度和效率上都获得了较好的表现.文献[22]通过实验证明,对于依赖指针指向信息的应用,指向分析的结果会对应用本身的效率有举足轻重的影响.在这些应用中,上下文敏感的引入虽然降低了指向算法的效率,增加了开销,但是算法在精度上获得的优势更为明显.由于算法为应用提供了更为精确的指针指向信息,使得客户应用的效率显著增加,而开销方面如内存的使用反而减少了.除了上下文敏感,算法可以引入更多维度的信息.文献[4]指出,可扩展的维度包括有流敏感、对象敏感、字段敏感等.文献[11]通过实验指出,上下文敏感、字段敏感可以有效提高指针分析的精度.

因此,我们以 Andersen 的算法为基础,设计了一个上下文敏感和字段敏感的指针分析算法.

文献[12−16]都是 Inclusion-based 的指针分析算法的扩展,我们工作的基本框架与文献[13,14]比较接近.文献[13]中,Heintze 和 Tardieu 提出的 CLA 算法框架对 Anderson 算法进行了扩展.CLA 框架下,所有单个过程的指

向关系被提前分析,并写入到文件中.在进行全局指向关系分析时,算法对新遇到的扩展点,读出文件中的分析结果,将局部指向关系扩展到全局指向图中.这个框架可以有效地减少分析过程对内存的使用需求,Heintze 和 Tardieu 在这个框架下实现了流不敏感和上下文不敏感的指向关系分析算法.文献[14]中,Whaley 和 Lam 基于他们的框架进行了新的扩展,实现了一个上下文敏感的分析算法.由于我们在局部图中使用间接指向表示,因此在进行过程间分析时效率更高.同时,我们简化了过程内指向关系的表示,使得我们的算法即使没有把单个过程的指向关系写入文件,对内存的需求也在合理的范围内.

文献[23]通过线下的预分析,获取可以合并的变量集合(这些变量具有相同的指向关系),从而减少输出的结点个数.文献[24,25]中,通过在线的环侦测技术,合并相关结点,从而减少结点数量.这类优化思想的出发点在于减少迭代时结点的个数.环侦测的算法在每次新边插入的时候被触发,从而引入了相当大的计算开销.文献[26]中提出的 Lazy Cycle Detection(LCD)和 Hybrid Cycle Detection(HCD)算法改变了触发环侦测算法的时机,从而减少了优化带来的开销,但是算法不能保证检测所有的环,优化效果不彻底.我们的算法在新的方法指向图被扩展入全局指向图之时而不是新边插入之时被触发,这样的触发粒度可以在一定程度上减少优化过程带来的开销.另外,算法从变量之间指向关系的偏序关系出发,构造了变量结点之间的拓扑序列,同时侦测结点之间的环,合并相关结点.这样不仅可以检测到所有的环,减少结点个数,更可以减少迭代的次数.

文献[15]首次使用 BDD 实现 Java 程序的指向分析,实验证明,使用 BDD 可以有效地减少分析过程中对内存的使用.文献[26]中的实验证明,文献[15]使用 BDD 虽然可以有效减少对内存的使用,但是时间的消耗反而增加.另外,使用 BDD 对指向关系进行编码,掩盖了指向关系中潜在的语义,损失了对算法进行进一步优化的可能性.文献[15]中,算法是字段不敏感的,同时也无法处理间接方法调用.这些都是 BDD 带来的限制.

文献[27]论证了指向分析实现并行计算的可能性,并针对 HCD 实现了并行算法.如果结点之间是无关的,那么结点的扩展可以并行完成.我们将在未来的工作中把并行的思想加入到我们的算法中,进一步减少优化的开销,同时使得整体算法获得更好的效率表现.

5 总 结

指针指向分析是静态分析工作的一个重要课题,也是各项优化技术和程序分析工具的基础.为了能进一步提高算法的精确性和可扩展性,越来越多的新技术被融合到这项工作中.

我们基于 Andersen 算法实现了上下文敏感、字段敏感的指针指向分析算法,并且对算法进行了优化.通过对结点指向关系拓扑排序,进而发现并消除环,从而极大地减少迭代的工作量.实验数据表明,这个算法有效提高了指针分析工作的效率与精度.在未来的工作中,我们计划在现有工作的基础上进一步发现指向关系扩展时可以合并的地方,提高合并的粒度,从而进一步提高指针分析工作的效率.

References:

[1] Landi W, Ryder BG, Zhang S. Interprocedural modification side effect analysis with pointer aliasing. In: Proc. of the Conf. on Programming Language Design and Implementation (SIGPLAN'93). New York: ACM Press, 1993. 56−57. [doi: 10.1145/155090.155096]

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

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

[4] Ryder BG. Dimensions of precision in reference analysis of object-oriented programming languages. In: Hedin G, ed. Proc. of the CC 2003. LNCS 2622, Berlin, Heidelberg: Springer-Verlag, 2003. 126−137. [doi: 10.1007/3-540-36579-6_10]

[5] Diwan A, Moss J.E.B, McKinley K. Simple and effective analysis of statically typed object-oriented programs. In: Proc. of the Conf. on Object-Oriented Programming Systems, Languages, and Applications. 1996. 292−305. [doi: 10.1145/236337.236367]

[6] Bacon DF, Sweeney PF. Fast static analysis of C++ virtual function calls. In: Proc. of the Conf. on Object-Oriented Programming Systems, Languages, and Applications. New York: ACM Press, 1996. 324−341. [doi: 10.1145/236337.236371]

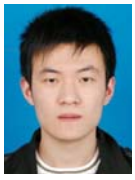
[7] Whaley J, Monica S. Lam. Cloning-Based context-sensitive pointer alias analysis using binary decision diagrams. PLDI, 2004,39(6):131−144. [doi: 10.1145/996841.996859]

[8] Lattner C, Lenharth A, Adve V. Making context-sensitive points-to analysis with heap cloning practical for the real world. In: Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation. 2007. [doi: 10.1145/1250734.1250766]

[9] Pearce DJ, Kelly PHJ, Hankin C. Efficient field-sensitive pointer analysis for C. In ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE), 2004. [doi: 10.1145/1290520.1290524]
[10] Andersen LO. Program analysis and specialization for the C programming language [Ph.D. Thesis]. DIEM: University of Copenhagen, 1994.
[11] Liang DL, Pennings M, Harrold MJ. Evaluating the impact of context-sensitivity on Andersen's algorithm for Java programs. In: Proc. of the PASTE 2005. 2005. [doi: 10.1145/1108792.1108797]
[12] Liang DL, Pennings M, Harrold MJ. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In: Proc. of the Workshop on Program Analysis for Software Tools and Engineering. 2001. 73−79. [doi: 10.1145/379605.379676]
[13] Heintze N, Tardieu O. Ultra-Fast aliasing analysis using CLA: A million lines of C code. In: Proc. of the Conf. on Programming Language Design and Implementation (PLDI). 2001. 146−161. [doi: 10.1145/378795.378855]
[14] Whaley J, Lam MS. An efficient inclusion-based points-to analysis for strictly-typed languages. In: Proc. of the Static Analysis Symp. 2002. 180−195. [doi: 10.1007/3-540-45789-5_15]
[15] Berndl M, Lhot´ak O, Qian F, Hendren L, Umanee N. Points to analysis using BDDs. In: Proc. of the Conf. on Programming Language Design and Implementation. New York: ACM Press, 2003. 103−114.
[16] Sridharan M, Gopan D, Shan LX, Bodik R. Demand-Driven points-to analysis for Java. In: Proc. of the 20th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications. 2005. [doi: 10.1145/1094811.1094817]
[17] Rountev A, Milanova A, Ryder BG. Points-to analysis for Java using annotated constraints. In: Proc. of the Conf. on Object-Oriented Programming Systems, Languages, and Applications. 2001. 43−55. [doi: 10.1145/504282.504286]
[18] Steensgaards B. Points-to analysis in almost linear time. In: Proc. of the ACM Symp. on Principles of Programming Languages (POPL). New York: ACM Press, 1996. 32−41. [doi: 10.1145/237721.237727]
[19] Das M. Unification-Based pointer analysis with directional assignments. In: Proc. of the ACM Conf. on Programming Language Design and Implementation (PLDI). New York: ACM Press, 2000. 35−46. [doi: 10.1145/349299.349309]
[20] Kahlon V. Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis. In: Proc. of the PLDI 2008. New York: ACM Press, 2008. [doi: 10.1145/1375581.1375613]
[21] Hind M, Pioli A. Which pointer analysis should I use? In: Proc. of the International Symposium on Software Testing and Analysis (ISSTA). 2000. 113−123. [doi: 10.1145/347324.348916]
[22] Lhoták O, Hendren L. Context-Sensitive points-to analysis: Is it worth it? In: Proc. of the Int'l Conf. on Compiler Construction. LNCS 3923, 2006. 47−64. [doi: 10.1007/11688839_5]
[23] Rountev A, Chandra S. Off-Line variable substitution for scaling points-to analysis. In: Proc. of the 2000 Conf. on Programming Language Design and Implementation. 2000. [doi: 10.1145/349299.349310]
[24] Fähndrich M, Foster JS, Su Z, Aiken A. Partial online cycle elimination in inclusion constraint graphs. In: Proc. of the Programming Language Design and Implementation (PLDI). 1998. 85−96. [doi: 10.1145/277650.277667]
[25] Su ZD, Fähndrich M, Aiken A. Projection merging: Reducing redundancies in inclusion constraint graphs. In: Proc. of the 27th Annual ACM SIGPLANSIGACT Symp. on Principles of Programming Languages (POPL). 2000. [doi: 10.1145/325694.325706]
[26] Hardekopf B, Lin C. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In: Proc. of the Programming Language Design and Implementation (PLDI). 2007. [doi: 10.1145/1250734.1250767]
[27] Mendez-Lojo M, Mathew A, Pingali K. Parallel inclusion-based points-to analysis. In: Proc. of the OOPSLA/SPLASH. 2010. [doi: 10.1145/1869459.1869495]

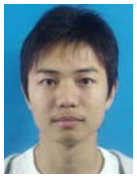
李倩(1982－),女,江苏苏州人,博士生,主要研究领域为程序分析.

王林章(1973－),男,博士,副教授,主要研究领域为模型驱动的软件测试与验证,软件测试自动化.



汤恩义(1982－),男,博士生,主要研究领域为程序分析,软件测试.

赵建华(1971－),男,博士,教授,博士生导师,主要研究领域为软件工程,形式化方法.



戴雪峰(1989－),男,学士,主要研究领域为程序分析.