

点间确定别名及其在 Java 程序 数据依赖分析中的应用

钱 巨 徐宝文 周毓明

(东南大学计算机科学与工程学院 南京 210096)

(江苏省软件质量研究所 南京 210096)

摘 要 堆内存的大量使用使得 Java 程序上数据依赖关系的精确提取仍存在许多困难. 对于堆空间上的依赖提取, 通常的做法是先对堆上空间进行命名, 再据此分析依赖关系. 然而该方法不能在多个定义间进行强更新, 故分析精度不够理想. 针对此问题, 该文首先提出了一种点间确定别名的概念, 然后用它生成强更新和相对更新来精化数据依赖分析. 实验表明, 与不进行强更新和相对更新的数据依赖分析方法相比, 新算法能够在相对较少的额外时间消耗内, 有效地提高堆空间上依赖分析的精度.

关键词 数据依赖; 指针分析; 别名分析; 确定别名; 强更新

中图法分类号 TP311

Interstatement Must Alias and Its Application in Dependence Analysis of Java Programs

QIAN Ju XU Bao-Wen ZHOU Yu-Ming

(School of Computer Science and Engineering, Southeast University, Nanjing 210096)

(Jiangsu Institution of Software Quality, Nanjing 210096)

Abstract Data dependence is widely used in software engineering activities. Due to the massive use of heap objects, it is still difficult to extract them from Java programs precisely. Several methods have been proposed to address this problem. One simple but scalable method is performing dependence analysis based on heap object naming. However, such a method, although effective, is not precise enough. This paper firstly introduces a notion of interstatement must alias, and then refines dependence analysis by generating strong updates and relative updates with these aliases. Empirical results show that the proposed algorithm can effectively improve the precision of dependence analysis for heap locations with very limited analysis time increment.

Keywords data dependence; pointer analysis; alias analysis; must alias; strong update

1 引 言

数据依赖关系在软件工程领域有着广泛应用.

在一个程序中, 语句 u 数据依赖于语句 d 当且仅当从 d 到 u 存在一条程序路径, d 定义了某存储空间 l , u 使用了 l 的值, 且该路径上没有其它对存储空间 l 的定义^[1]. 对于不含别名的程序, 数据依赖可由可

达定义分析^[2]获得;对于含指针所导致的别名的程序,数据依赖分析则必须首先借助指针分析^[3]来识别间接的变量访问.堆空间的引入给数据依赖的分析带来了更多困难.由于静态情况下难以对不同程序点上所访问的堆空间作准确区分,其分析精度很难得到保证.我们通过深入研究并发程序的依赖分析、基于依赖分析的度量等问题^[4-5]发现,对于这些应用,没有堆空间上的高精度依赖分析就难以获得很好的效果,因此提高堆空间上依赖分析的精度将非常有意义.

为进行堆空间上的依赖分析,首先要对堆空间进行识别区分,这可以通过数据结构形态分析或对象命名来实现^[8].前者因复杂度过高,难以适应大型程序分析的需要.后者相对简单,它结合指向分析目前已能用于数据依赖的提取,但该方法由于不能在堆空间定义之间进行强更新^[6],故精度并不理想,这对于 Java 这样指针和堆内存使用广泛的面向对象语言,显得尤为明显.

尽管当前亦有少量学者研究了堆空间上的强更新问题,但其解决方案均存在很大的局限性,效率低、通用性差,且不能全面系统地识别程序中的强更新.本文提出了一种利用点间确定别名来优化数据依赖分析的方法,该方法能利用不同程序点上访问路径之间的确定别名关系来识别更多的强更新.此外,本文还提出了相对更新的概念,它是对现有强更新概念的一个有益补充,采用这种更新能进一步提高依赖分析的精度.较之已有的方法,本文的方法性更佳、可配置性强、更系统,且能识别大量现有方法所不能识别的冗余依赖.我们实现了一个数据依赖分析系统的原型,实验结果表明,与不进行强更新和相对更新的算法相比,新算法能够在相对较少的额外时间消耗内,有效地提高堆空间上数据依赖分析的精度.这将为后继应用带来精度与效率上的双重收益.

本文以在程序切片^[7]等领域有较多应用的上下文不敏感依赖关系提取作为研究目标,详细阐述了我们的方法.在后续的章节中,文章首先介绍强更新问题,然后提出点间确定别名的概念并给出它们的求解方法.以此为基础,我们给出利用点间确定别名优化数据依赖分析的具体方法,并通过实验验证它的有效性.

2 强更新问题

象空间来描述一组运行时的物理空间.其中抽象堆空间描述了堆上分配的对象及其属性域,而特殊空间 *null* 描述了空指针状态.依赖关系仍由可达定义分析来识别.该分析中,对某抽象空间的强更新指用新的可达定义覆盖前面关于该空间的可达定义,而那些只在可达定义集中添加新定义的更新称为弱更新.强更新越多,依赖分析越精确.

在不使用堆空间时,只要一条语句仅表示对唯一一个抽象空间的定义,该定义就能够覆盖前面关于此空间的所有定义.文献[8]即用类似思想对局部变量的指向集进行强更新.然而,在有堆空间使用时,上述方法却并不适用.为此,Altucher 和 Landi 提出了一种利用扩展确定别名进行强更新的方法(AL 方法)^[9],该方法在语句仅定义唯一一个抽象堆空间的情况下保留了部分强更新能力.他们认为,若语句 *a* 和 *b* 均唯一地定义了抽象堆空间 *l*,且从 *a* 到 *b* 的所有不包含 *a*, *b* 的路径上 *l* 未被重新分配,则语句 *b* 的定义能覆盖最近一次语句 *a* 的定义.例如,在图 1 的 *foo()* 方法中,由于 *F2* 和 *F3* 上的 *s* 均指向同一抽象空间,并且它表示的物理空间在 *F2* 和 *F3* 之间并未改变,因此 AL 方法可断定 *F3* 的定义能够完全覆盖 *F2* 的定义.

```

class Node{int x;...}
void foo(Node[] v, Node e){
F1  Node s=new Node();
F2  s.x=0;
F3  s.x++;
    for (int i=0; i<v.length; i++){
F4      s.x+=i;
        v[i]=s;
F5      s=new Node();
F6      s.x=v[i].x;
    }
F7    q=s;
F8    s=null;
F9    q.x=e.x;
}

void bar(){
    Node p=create();
    Node q=create();
B1    p.x=1;
B2    q.x=2;
}

Node create(){
    return new Node();
}

void reset(Iterator it, int d){
    while(it.hasNext()){
        Node p=(Node)it.next();
R1    p.x=d;
R2    Node q=p;
R3    System.out.print(q.x);
    }
}

```

图 1 几段有趣的 Java 程序

与多数指针分析方法相同,本文仍用静态的抽

AL 方法的主要缺点是: 首先, 它仅在唯一地定义了同一个抽象空间 (null 也统计在内) 的语句间才可能有强更新, 而这种唯一指向性在多数高性能指向分析中难以得到保证; 其次, 该方法需要判断一个抽象空间所表示的物理空间在两个程序点间是否变化, 这种需求使得它不能有效地处理复杂控制结构, 不能进行跨循环体边界和跨过程边界的强更新. 以上两点表明 AL 方法不能全面地识别系统中的强更新, 除此以外, 它还依赖于一个特定的流敏感、上下文敏感指向分析算法和一个特定的堆空间命名方法, 因此不具有普遍适用性, 并且这样的算法本身也并不适合处理大型程序.

图 1 的例子中, $F4$ 是对 $F6$ 定义的一个强更新, 但 AL 方法并不能识别这种更新. 这首先是因为在 AL 方法下, $F4$ 上 s 的指向集含有不止一个元素, 即 $F4$ 和 $F6$ 并没有唯一地定义同样的抽象空间, 另一方面也因为从 $F6$ 到 $F4$ 跨越了循环体边界. 同样的, AL 方法也不能发现 $F4$ 对 $F3$ 的强更新和 $F9$ 对 $F6$ 的强更新. 而图 1 中的 $bar()$ 方法则表明 AL 方法只能在一个特定的堆命名方式下工作. 当根据对象的分配语句来命名堆空间时, AL 方法将错误地认为 $B2$ 是对 $B1$ 的一个强更新.

通过强更新可识别许多冗余依赖, 但即便如此仍有大量堆内存上的冗余依赖不能由此消除. 图 1 的 $reset()$ 方法中, 从方法入口的 it 参数传入了一组 $Node$ 对象 x 域的定义, $R1$ 只是对其中一个对象的 x 域的赋值, 它不是一个强更新, 因此 $reset()$ 入口的这些定义信息将能到达 $R3$, 造成 $R3$ 对方法入口的伪依赖. 事实上, 相对于 $R3$ 的引用, 传入的每个 $Node$ 对象 x 域都会被 $R1$ 重新定义, 因此 $R3$ 并不直接数据依赖于方法入口, 即相对于 $R3$ 的引用, $R1$ 的定义可看作是对 $reset()$ 入口传入的 $Node$ 对象 x 域的强更新, 这种相对更新目前并没有得到研究.

就我们所知, 当前除文献[9]外极少有研究深入地探讨依赖分析中的强更新问题, 更没有研究探讨相对更新, 而图 1 展示的情况在 Java 程序中很常见, 因此有必要研究新的方法来处理它们. 为识别更多强更新、相对更新, 避免伪依赖, 本文提出了一种基于点间确定别名的依赖分析方法. 采用该方法可以在任一堆空间命名方式下获得更精确的依赖信息.

3 点间确定别名的基本概念

在 Java 中, 访问路径^[10]是在变量上施加一组

域访问或数组元素访问所得的左值表达式, 其长度即这些访问的个数. 本文忽略含数组元素访问的访问路径, 因为在它们之上难以构造确定别名. $ap.\delta$ 表示在访问路径 ap 上增加一个额外的域访问序列 δ 所得的新访问路径 (这里我们约定文中出现的所有访问路径均有物理意义). 抽象空间 p 被称为访问路径 ap 的一个决策指针, 当且仅当 p 的不同取值能使 ap 表示不同的存储空间. 比如 q 是访问路径 $q.next$ 的决策指针. 本文记程序点 n 上访问路径 ap 表示的抽象空间集为 $Loc(n, ap)$, 记 n 上 ap 的决策指针集为 $Dec(n, ap)$. 对于语句 n , 如果其上 q 指向 o_1 , $o_1.next$ 指向 o_2 , 那么 $Dec(n, q.next.next) = \{q, o_1.next\}$, $Loc(n, q.next.next) = \{o_2.next\}$.

一个访问路径在不同点上可能有不同的状态, 我们用“ n 点上的访问路径 ap ”来强调对 n 点上访问路径 ap 状态的关注. 通常一个别名仅表示同一点上两个访问路径之间的等价关系. 它不能描述不同点上访问路径之间的关系, 因此不同点上的别名间通常并不具有传递性. 比如, 从 n_1 点上的别名 $\langle p.x, q.x \rangle$ 和 n_2 点上的别名 $\langle s.x, q.x \rangle$ 并不能获知 n_1 点的 $p.x$ 和 n_2 点的 $s.x$ 存在别名关系. 在不使用堆的程序中, 确定指向关系可导出不同程序点上的访问路径之间的确定别名关系. 然而, 由于 Java 中所有的指针都是堆指针, 即便是上述推导也不再可行. 为表示不同程序点上访问路径之间的确定别名关系, 必须引入新的别名表示.

本文将相同或不同程序点上访问路径之间的别名关系统称为点间别名, 而传统意义上的别名则称为点内别名. 点内别名是点间别名的特例. 与点内确定别名类似, 一个点间确定别名是在所有执行中成立的别名. 它能描述不同点上的两个访问路径关于其所表示空间的确定等价关系, 可以用于实现指针分析、可达定义分析等分析中的强更新. 在不引起混淆的情况下, 我们也用别名来代指点间别名.

定义 1. 程序点 m 上的前向确定别名 $\langle n: ap_1, ap_2 \rangle$ 表示当程序流从别名源节点 n 的最近一次出现流向 m 时, n 出口的访问路径 ap_1 和当前点 m 入口的访问路径 ap_2 总表示相同的物理空间.

定义 2. 程序点 m 上的后向确定别名 $\langle ap_1, n: ap_2 \rangle$ 表示当程序流从 m 流向后继节点 n 的第一次出现时, 当前点 m 出口的访问路径 ap_1 和 n 入口的访问路径 ap_2 总表示相同的物理空间.

点间别名上可进一步扩展上流信息. 按流向不同, 我们可将点间确定别名扩展为前向确定别名和

后向确定别名. 定义 1,2 给出了这两种别名的定义. 这里, 前向和后向别名都带有流向条件, 因此一个别名在所有执行中都成立, 实际上只需要它在所有满足流向条件的路径上成立. 对于这些别名, 当 n 用特殊符号 ' \perp ' 代替时, 它表示当前点上 (即当前点入口) 的访问路径 ap_1 和当前点上的访问路径 ap_2 存在别名关系. 仅在这时别名才是对称的, 即别名 $\langle \perp : ap_1, ap_2 \rangle$ 也意味着别名 $\langle \perp : ap_2, ap_1 \rangle$.

在图 1 的例子中, 由于 s 在 $F2$ 和 $F3$ 之间并未被重新定义, $\langle F2 : s.x, s.x \rangle$ 是 $F3$ 上的前向确定别名, $\langle s.x, F3 : s.x \rangle$ 是 $F2$ 上的后向确定别名. 此外, $\langle F6 : s.x, s.x \rangle$ 是 $F4$ 上的前向确定别名. $F6$ 也许不出现在 $F4$ 的前面, 但是一旦其出现, 从 $F4$ 的角度看, 由于 s 在从最近一个 $F6$ 到 $F4$ 的路径上未被重新定义, 最近一次 $F6$ 执行后的 $s.x$ 总是和当前点上的 $s.x$ 存在别名关系. 类似的, 由于 s 在从 $F6$ 到下一个 $F4$ 的路径上没有被重新定义, $\langle s.x, F4 : s.x \rangle$ 是 $F6$ 上的一个后向确定别名. $F9$ 上的前向确定别名 $\langle F6 : s.x, q.x \rangle$ 是一个更具代表性的例子. 不管 $F6$ 点上 s 的指向如何改变, 从 $F9$ 的角度看, 最近的一次 $F6$ 执行后的 $s.x$ 总是和当前点上的 $q.x$ 表示一样的空间. 这个前向别名并不对应 $F6$ 上的后向确定别名 $\langle s.x, F9 : q.x \rangle$, 因为站在 $F6$ 的角度, 其出口的 $s.x$ 未必和 $F9$ 入口的 $q.x$ 表示相同的空间. 后向确定别名也未必总对应一个前向确定别名. 由于从 $F3$ 的角度看, 下一次 $F4$ 出现 ($F4$ 在循环中的首次出现) 上的 $s.x$ 总和当前点出口的 $s.x$ 表示相同的空间, $\langle s.x, F4 : s.x \rangle$ 是 $F3$ 上的一个后向确定别名. 但它对应的 $\langle F3 : s.x, s.x \rangle$ 并不是 $F4$ 上的一个前向确定别名, 因为站在 $F4$ 的角度, 最近一次 $F3$ 执行后的 $s.x$ 未必和当前点入口的 $s.x$ 表示相同的空间. 除了这些, $\langle R1 : p.x, q.x \rangle$ 是 $R3$ 上的一个前向确定别名, $\langle p.x, R3 : q.x \rangle$ 是 $R1$ 上的一个后向确定别名. 以上仅是图 1 程序中具有代表性的点间别名, 其它点间别名在此并未一一列出. 一些点间别名并无实际的使用价值, 因此也根本不需要分析.

较之可能别名, 确定别名只关心在所有执行中都成立的别名, 因此数量相对较少. 另外, 丢弃一个确定别名并不影响分析结果的正确性, 因此确定别名分析也易于保证安全性. 故而确定别名适合作为指向分析的一种补充, 以进一步优化程序的分析.

4 点间确定别名的计算

点间确定别名可通过数据流迭代进行分析. 其

计算依赖于指向分析, 指向分析确定了每个程序点可能定义的抽象空间, 从而使数据流迭代能够有效地对一个程序点上的流入别名集进行更新. 这里我们假定所有指向信息均已知. 本节首先介绍过程内和过程间的前向确定别名分析, 然后分析其相关属性. 由于后向点间确定别名的计算是前向确定别名计算的对称问题, 本文不对其作详细展开. 下面的别名分析未经强调均指前向确定别名分析.

4.1 确定别名分析

设 $Alias_-[n]$ 和 $Alias_+[n]$ 分别表示程序点 n 入口和出口的确定别名集, 前向确定别名的计算可用下列数据流方程描述:

$$Alias_-[n] = merge(\langle \langle m, Alias_+[m] \rangle \mid m \in pred(n) \rangle) \quad (1)$$

$$Alias_+[n] = transit(Alias_-[n] - Kill[n], Gen[n]) \quad (2)$$

方程 (1)、(2) 分别计算程序点 n 入口和出口的确定别名集, $pred(n)$ 是 n 的前驱节点集. 对程序点 n , 若其为非控制流会合点, 则 $merge$ 操作直接返回 $Alias_+[m]$, m 是 n 的唯一前驱. 否则 $merge$ 返回会合后仍成立的确定别名. 对于点内别名, 一个别名继续成立, 仅当它在点 n 的每个前驱的出口都成立. 对于点间别名, 别名 $\gamma = \langle s : ap_1, ap_2 \rangle$ 继续成立仅当:

$$\exists m \in pred(n) \{ \gamma \in Alias_+[m] \} \wedge \forall m \in pred(n) \{ \gamma \notin Alias_+[m] \rightarrow !path(s, m) \} \quad (3)$$

其中谓词 $path(s, m)$ 检查从点 s 到点 m 是否存在一条路径, 它可以通过构造一个基本块层次的局部路径表在常数时间内得到检验^[11]. 这里一个点间确定别名继续成立的条件是它在从其源结点到当前节点间的每条路径上都成立, 也即它必须出现在每个从其源节点出发可达的前驱节点的出口别名集中.

图 2 的代码构建一个以 hd 为首的链表. 现考虑节点 12 上的会合操作. 确定别名 $\langle 3 : hd.next, p.next \rangle$ 出现在语句 5 的出口, 但不出现在语句 10 的出口. 由于语句 10 从语句 3 可达, 在从语句 3 的最近一次出现到语句 12 的路径上, 语句 3 出口的 $hd.next$ 未必和语句 12 入口的 $p.next$ 表示相同的物理空间, 因此这个别名将不能在会合后成立. 确定别名 $\langle 9 : p.next, q.next \rangle$ 同样不同时出现在会合节点的两个前驱上. 然而, 由于从语句 9 到语句 5 并不存在一条程序路径, 当控制流从语句 9 的最近一次出现流向语句 12 时, 语句 9 出口的 $p.next$ 总和语句 12 入口的 $q.next$ 表示相同的空间, 故该别名将在会合后成立.

```
class Node(int x; Node next;){
```

```
1 Node hd, p, q;
2 hd = new Node();
3 hd.next = null;
4 if(flag){
5   p=hd, q=hd;
6 }
7 else{
8   p=new Node();    (3: hd.next, p.next)
9   p.next = hd;
10  hd=p, q=p;
11 }
12 hd.x = ...
```

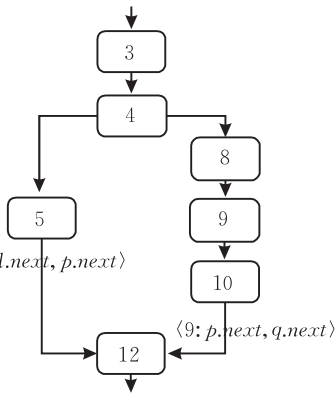


图 2 一个 Java 程序片段和它对应的控制流图

从程序点入口的别名集到出口的别名集, 方程(2)所表示的流函数首先去除那些不再成立的确定别名, 然后再根据赋值生成新的确定别名, 并传递地计算程序点出口能够成立的所有确定别名. 对形如“ $n:ap_1 = \dots$ ”的赋值语句, $Kill$ 集定义为

$$Kill[n] = \{\langle n: \times, \times \rangle\} \cup$$

$$\{\langle \times: \times, ap \rangle \mid Loc(n, ap_1) \cap Dec(n, ap) \neq \emptyset\},$$

其中前一分量被去除是因为 n 的最近出现已发生变化(这里“ \times ”表示任一可能取值). 比如图 1 中确定别名 $\langle R1: p.x, q.x \rangle$ 会随数据流迭代被传播到 $R1$ 的入口, 再次分析 $R1$ 时, 它将被淘汰, 因为这时 $R1$ 的最近出现发生了改变, 前面关于其最近出现的别名判定已无意义. 后一分量被去除则是因为 n 的定义可能影响到当前访问路径 ap 的最终表示. 图中, $F5$ 将淘汰任何形如 $\langle \times: \times, s.x \rangle$ 的别名, 因为这时 s 被重新定义, 基于旧 $s.x$ 所表示空间的别名判定也将变得无效. 对于后一个分量中的对称别名, 一旦其中之一被淘汰, 另一个也将被淘汰.

去除了不再成立的别名后, 对形如“ $n:ap_1 = rhs$ ”的赋值, 流函数通过下列公式获得程序点 n 上直接生成的别名集:

$$Gen[n] = \{\langle \perp: ap_1.\delta, ap_2.\delta \rangle \mid rhs = ap_2 \in$$

$$AccessPath \wedge Loc(n, ap_1) \cap Dec(n, ap_2) = \emptyset\} \cup$$

$$\{\langle n: ap_1, ap_1 \rangle \mid Dec(n, ap_1) \neq \emptyset\},$$

其中前一分量是当前点上直接生成的点内别名. 条件 $rhs = ap_2 \in AccessPath$ 要求赋值右边同样是一个访问路径, 条件 $Loc(n, ap_1) \cap Dec(n, ap_2) = \emptyset$ 要求 ap_1 的定义不影响赋值右边的访问路径 ap_2 的最终表示. 比如, 对图 1 中的 $F7$ 语句“ $q=s$ ”, 由于 q 的定义不会影响 s 的取值, 该点上就可以生成诸如 $\langle \perp: q.x, s.x \rangle$ 的一组点内别名. δ 是任一有效的访问路径扩展. 我们限制扩展后的访问路径长度不超过 k , 这样做是安全的, 它将保证算法的可终止性.

$Gen[n]$ 中后一个分量是当前点上直接生成的点间别名, 它表示 n 最近一次出现后的 ap_1 和当前的 ap_1 访问同一物理空间. 例如, 在图 1 的 $F4$ 上, 我们将生成点间别名 $\langle F4: s.x, s.x \rangle$, 该别名可以被进一步传递以衍生其它别名. 除了赋值语句, 指针的等值测试也能够生成确定别名, 该生成与赋值上的别名生成是类似的.

在进行了别名的去除与生成之后, $transit$ 操作返回一个依旧有效的程序点入口传入别名和新生成别名的传递闭包. 设 $transit$ 的结果为 Φ , Φ 首先蕴含别名集 $Alias_ [n] - Kill[n]$ 和 $Gen[n]$. 在别名传递中, 点间别名之间并不能进行传递, 但点间别名和点内别名之间可以, 即

$$\forall ap_1, ap_2, ap_3 \{ \langle \times: ap_1, ap_2 \rangle \in \Phi$$

$$\wedge \langle \perp: ap_2, ap_3 \rangle \in \Phi \rightarrow \langle \times: ap_1, ap_3 \rangle \in \Phi \}.$$

比如, 在图 1 的 $reset()$ 方法中, $R2$ 的出口有别名 $\langle R1: p.x, p.x \rangle$, 它可以和赋值产生的别名 $\langle \perp: q.x, p.x \rangle$ 进行传递, 得到确定别名 $\langle R1: p.x, q.x \rangle$.

为进行别名的数据流迭代, 初始化时, 方法入口的别名集被设定为一个对方法入口的上下文无关保守描述(没有任何点内别名), 其它程序点上的别名集被设定为所有别名的全集(可不以物理的方式表示出来). 在迭代过程中, 各程序点上的确定别名集单调地减小, 当这些集合均不再变化时, 计算收敛. 方法入口的别名集通常被初始化为一组形如 $\langle entry: ap, ap \rangle$ 的点间别名的集合, 比如对图 1 的 $foo()$ 方法, 我们可以在它的入口别名集中包含 $\langle entry: e.x, e.x \rangle$ 这样的别名. 传递这些别名就可以获知方法体内的访问路径与方法入口上的访问路径之间的关系.

在过程间, 所有方法首先被拓扑排序以保证被调方法尽可能在施调方法前分析. 每个方法仅被分析一遍. 分析一个方法内的语句时, 如果该语句含调用, 我们先根据被调方法的副作用将所有可能受被调方法内定义影响的确定别名丢弃. 然后, 对于被调方法, 如果它已经分析过, 我们收集其出口的所有点内确定别名, 并将它们映射为调用返回后实参上的确定别名; 如果它还未被分析过, 我们直接忽略调用可能产生的新点内确定别名. 此外, 对施调方法内的访问路径 ap , 如它表示的空间可能被被调方法修改, 我们还将调用点出口建立确定别名 $\langle callsite: ap, ap \rangle$ 以方便可达定义分析时的强更新.

以上仅是前向确定别名的分析算法, 后向点间确定别名的计算与此类似. 相对于前向确定别名的

计算,它主要有两点改变.首先,数据流迭代由前向变成了后向.其次,在一个程序点上,我们先传递点间别名,然后再通过计算 *Kill* 集来去除无效别名.比如对于图 1 中的 *R2* 语句,从其出口传入的后向点间别名集为 $\{\langle q.x, R3:q.x \rangle\}$. 我们先根据赋值“ $q=p$ ”产生的点内确定别名来传递别名集,获得新别名 $\langle p.x, R3:q.x \rangle$,然后再分析赋值对象,去除 $\langle q.x, R3:q.x \rangle$,从而得到其入口的别名集 $\{\langle p.x, R3:q.x \rangle\}$.

4.2 算法属性

过程内前向确定别名分析具有确定的终止性和语义正确性,以这些性质为基础,可以构造过程间前向确定别名分析的终止性和正确性证明,类似的也可以构造后向确定别名分析的终止性和正确性证明.限于篇幅,本节仅讨论过程内前向确定别名分析的相关性质.

定理 1. 过程内前向确定别名分析算法能够终止.

证明. 前向点间确定别名的分析是一个数据流迭代过程.迭代开始前,任一程序点 n 入口和出口的别名集 $Alias_-[n], Alias_+[n]$ 都是有限集.可以证明在每一轮迭代中任一语句 n 入口和出口的别名集 $Alias_-[n], Alias_+[n]$ 都不增大,即每个程序点上的别名集单调减小.因此,整个数据流迭代最终必将收敛.更详细的证明参见附录. 证毕.

定理 2. 过程内前向确定别名分析算法是语义正确的.

证明. 本文采用与文献[12]一书中类似的方法来证明前向确定别名分析的语义正确性.为构造证明,我们首先定义了一个插桩语义(instrumented semantics),然后根据此语义定义一个别名集与程序执行轨迹的正确性关系 R .可以证明,在程序执行任一语句前后,分析所得的别名集与程序执行轨迹的正确性关系 R 都能得到满足,因此前向确定别名分析是语义正确的.详细证明参见附录. 证毕.

在算法复杂性方面,设一个方法有 n 条语句, l 个局部或全局变量,每个类最多有 f 个非静态属性域,访问路径长度上限为 k ,并且在任一程序点上,一个访问路径最多和 α 个访问路径形成点间确定别名,一个指针最多指向 β 个对象.显然,该方法内最多有 lf^k 个访问路径,每个访问路径最多可能访问 β^k 个抽象空间.由于本文的别名分析只关心各语句上被定义的访问路径和其它点上访问路径间的关系,因此在一个程序点上,描述不同点上访问路径间

关系的别名最多有 $O(\alpha n)$ 个,点内别名最多有 $O(\alpha lf^k)$ 个,总点间别名最多有 $O(\alpha n + \alpha lf^k)$ 个.

在点间别名的迭代中,设控制流图上有 e 条边,对于每条边,由于其出发点上的别名集每轮迭代中单调减小,它最多被处理 $O(\alpha n + \alpha lf^k)$ 次.处理一条边时,首先要计算该边目标点的入口别名集.由于只有一个前驱节点的出口别名集发生变化时,merge 过程的复杂度为 $O(\alpha n + \alpha lf^k)$,计算所有边目标节点入口别名集的总时间复杂度为 $O(e(\alpha n + \alpha lf^k)^2)$.对于程序点出口的别名集,由于一条语句的入口别名集最多变化 $O(\alpha n + \alpha lf^k)$ 次,它最多被计算 $O(\alpha n + \alpha lf^k)$ 次.对这些计算,数据流迭代前准备各访问路径的 *Loc* 集和 *Dec* 集以及各语句 *Gen* 集的总时间代价为 $O(n\beta^k lf^k)$.在计算一条语句出口别名集的过程中,生成新别名的代价是常数,去除失效别名的代价为 $O(\beta^k(\alpha n + \alpha lf^k))$,别名传递的代价为 $O(\alpha lf^k(\alpha n + \alpha lf^k))$,因此总的计算程序点出口别名集的时间代价最多为 $O(n(\alpha lf^k + \beta^k)(\alpha n + \alpha lf^k)^2)$.由于一般程序控制流节点的出度上限可以认为是常数(if 和 while 语句有两个出度,switch 语句的出度略多,但均非常有限),控制流图边数 e 可以认为是 $O(n)$.故此,整个别名分析的最大时间复杂度可以认为是 $O(n(\alpha lf^k + \beta^k)(\alpha n + \alpha lf^k)^2)$.

在空间方面,点间确定别名分析的复杂性主要源于存储点间确定别名的代价,由于所有程序点上点间别名的总数为 $O(n(\alpha n + \alpha lf^k))$,整个别名分析的最大空间复杂度为 $O(nk(\alpha n + \alpha lf^k))$.

事实上,分析中 k 通常取 1、2 这样的小整数.对于实际的程序,绝大多数指针仅指向一到两个抽象空间^[13].我们对 Java 标准库中多于 1000 个类的统计分析表明一个类平均只有 5 个非静态属性域(包括从父类继承的).每个方法中使用的变量数目 l 也远小于语句数 n .而确定别名分析要求一个别名在所有执行中都成立的特性决定了 α 也不可能太大.综合来看,点间确定别名分析实际的时间和空间复杂度并不会太高,整个分析又是上下文不敏感的,因此它们相对于繁琐的依赖分析是可以接受的.

5 点间确定别名在依赖分析中的应用

利用点间确定别名可以生成强更新和相对更新,从而有效地提高数据依赖分析的精度.为突出核心问题,本文仅以最普通的命名方式来抽象堆空间,程序点 p 上分配的对象统一地用 o_p 来表示.依赖关

系计算中,每个程序点上定义了可达定义集和可达引用集两种数据流集合.可达定义集记录能够到达该点的定义的集合,而可达引用集记录通过该点能够到达的变量引用的集合.语句 s 依赖于语句 t 当且仅当 t 上的某定义能够到达语句 s ,且语句 s 上存在一个变量的引用能够到达 t .可达定义集和可达引用集可分别通过前向和后向数据流迭代获得.

为精确分析可达定义,一条定义信息被记为 $\langle n, ap, l \rangle$,表示语句 n 通过其出口的访问路径 ap 定义了抽象空间 l .例如对语句“ $n: p.i = 1$ ”,如果 p 指向抽象空间 o_x ,则该语句上将产生定义 $\langle n, p.i, o_x.i \rangle$.对于语句“ $n: ap = \dots$ ”,若语句执行前访问路径 ap 访问空间 l ,而语句执行后,受定义本身的影响, ap 未必再访问 l ,则我们仅将定义记为 $\langle n, \top, l \rangle$,表示语句 n 通过其出口的某未知访问路径定义了抽象空间 l .

5.1 利用点间确定别名进行强更新

前向和后向确定别名都可用来识别强更新.对于形如“ $n: ap_1 = \dots$ ”的赋值,可达定义 $\langle m, ap_2, l \rangle$ 在下列两个条件之一得到满足时可以被强更新掉:

- (a) n 的入口有前向确定别名 $\langle m: ap_2, ap_1 \rangle$ 且定义 $\langle m, ap_2, l \rangle$ 并不出现在 m 的入口;
- (b) m 的出口有后向确定别名 $\langle ap_2, n: ap_1 \rangle$.

条件(a)可用于强更新是因为站在 n 的角度,当前点上的 ap_1 总是和最近一次 m 执行后的 ap_2 表示相同的空间.一旦 m 的入口没有定义 $\langle m, ap_2, l \rangle$,即每次仅有最近的一个 m 的定义能传播到 n 点,那么 n 的定义就可以覆盖由 $\langle m, ap_2, l \rangle$ 表示的定义.如果 m 的入口存在可达定义 $\langle m, ap_2, l \rangle$,那么它表明 m 点上有不止一次关于 l 的定义能够向下传播.由于 l 可能表示多个物理空间,同样一个定义 $\langle m, ap_2, l \rangle$ 这时可能表示对多个不同物理空间的定义,我们并不能确定 n 的定义是否一定覆盖所有这些定义.

条件(b)可以被用来进行强更新是因为站在 m 的角度,下一次 n 出现上的 ap_1 总是和当前语句出口的 ap_2 表示相同的空间,因此任何经 ap_2 产生的定义都将被 n 点的定义覆盖.

在图 1 的例子中, $F2$ 的定义可被表示为 $\langle F2, s.x, o_{F1}.x \rangle$. $F3$ 入口的前向别名 $\langle F2: s.x, s.x \rangle$ 揭示了 $F3$ 总是和最近的 $F2$ 定义相同的空间,而没有 $F2$ 的可达定义出现在 $F2$ 入口表明只有最近一次 $F2$ 的定义能够向下传播,根据强更新条件 a, $F3$ 的定义能够淘汰可达定义 $\langle F2, s.x, o_{F1}.x \rangle$.类似的,可以发现 $F4$ 和 $F9$ 的定义能够淘汰 $F6$ 产生的定义.除

了上述强更新,由于 $F3$ 出口有后向确定别名 $\langle s.x, F4: s.x \rangle$,也即 $F3$ 和其后最早的 $F4$ 出现定义了相同的空间,我们还可以发现 $F4$ 的定义是对 $F3$ 定义的强更新.前向别名和后向别名所能识别的强更新相交,但并不完全相同. $F3$ 定义对 $F2$ 定义的覆盖、 $F4$ 定义对 $F6$ 定义的覆盖通过前向和后向确定别名均能识别,但 $F9$ 定义对 $F6$ 定义的覆盖只能通过前向确定别名识别, $F4$ 定义对 $F3$ 定义的覆盖只能由后向确定别名识别.上述强更新中, $F3$ 对 $F2$ 定义的更新可以被 AL 方法识别,而其它强更新都是现有方法所不能识别的.

5.2 过程间强更新

面向对象程序倾向于有较多的模块,因此过程间的强更新显得尤为重要.目前几乎没有依赖分析方法支持关于堆空间的过程间强更新.为解决该问题,我们为每个长度不超过 k 的形参上的访问路径生成方法入口点的可达定义 $\langle entry, ap, l \rangle$,以表示方法入口处通过访问路径 ap 传入了对抽象空间 l 的定义.对那些可能通过多个访问路径访问的抽象空间,我们同时还生成可达定义 $\langle entry, \top, l \rangle$,以表示 l 的定义可能由不确定的访问路径传入.对图 3 中的方法 $clear()$,设其入口处 n 指向 o , $o.next$ 仍指向 o .当以 $k=1$ 为访问路径的长度上限时,算法将生成入口点的可达定义集 $\{ \langle entry, n, n \rangle, \langle entry, n.x, o.x \rangle, \langle entry, n.next, o.next \rangle, \langle entry, \top, o.x \rangle, \langle entry, \top, o.next \rangle \}$.这些定义将参与方法内的可达定义分析.

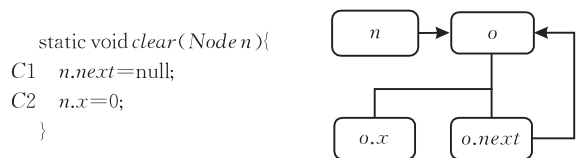


图 3 $clear()$ 方法及其入口的指向关系

当遇到方法调用时,我们先找出被调方法强更新掉的定义,再将被调方法内产生的外部可见定义转化为当前方法内的定义.被调方法能覆盖的定义是那些出现在其入口却不出现在其出口的定义.比如对图 3 中的 $clear()$ 方法,其入口定义 $\langle entry, n.x, o.x \rangle$ 和 $\langle entry, n.next, o.next \rangle$ 就被方法体覆盖了.当遇到对 $clear()$ 的调用时,这些被覆盖定义中的 $n.x$ 和 $n.next$ 可以被映射为实参上的访问路径.然后,方法调用可视为是对这两个实参上访问路径的定义,过程间的强更新也就变成了过程内的强更新.

在去除了失效的可达定义后,我们将被调方法产生的外部可见的定义映射为施调方法内的定义,

并添加到调用语句出口的可达定义集中. 映射分为两步. 首先, 定义 $\langle n, ap_1, l \rangle$ 将被映射为方法出口处的一个定义 $\langle exit, ap_2, l \rangle$, 其中 ap_2 是未被修改的形参上的一个访问路径, 虚拟的语句 “ $exit; ap_2 = \dots$ ” 应满足对 $\langle n, ap_1, l \rangle$ 进行强更新的条件. 前者保证被调方法内的定义可以进一步地被映射为施调方法内的定义, 后者要求 $\langle exit, ap_2, l \rangle$ 能够完全地表示定义 $\langle n, ap_1, l \rangle$ 的最终效果. 如果找不到这样一个映射, 定义 $\langle n, ap_1, l \rangle$ 将被映射为 $\langle exit, \top, l \rangle$ 以表示方法内经某个未知的访问路径定义了抽象空间 l . 此后, 定义 $\langle exit, ap_2, l \rangle$ 将被进一步映射为 $\langle callsite, ap_3, l \rangle$ 以表示调用点经访问路径 ap_3 定义了空间 l , ap_3 是形参上访问路径 ap_2 对应的实参上访问路径; 定义 $\langle exit, \top, l \rangle$ 将被映射为一个访问路径未知的定义 $\langle callsite, \top, l \rangle$. 这样被调方法内的定义就变成了一个施调方法内的定义.

对于图 3 的 `clear()` 方法, 假设有一个方法调用 “ $m:clear(p)$ ”, 方法内定义 $\langle C2, n.x, o.x \rangle$ 将首先被映射为方法出口定义 $\langle exit, n.x, o.x \rangle$, 然后它将进一步被映射为施调方法内定义 $\langle m, p.x, o.x \rangle$ 并添加到语句 m 出口的可达定义集中. 同样的, 方法内定义 $\langle C1, n.next, o.next \rangle$ 也将被映射为 $\langle m, p.next, o.next \rangle$ 添加到调用 m 的出口.

5.3 利用点间确定别名进行相对更新

对某抽象空间 l 的定义也许能到达程序点 n , 但 n 上对 l 的引用却未必会取到该定义所设定的值, 因为该引用所访问的物理空间可能此前已被 l 的另一定义更新了. 图 1 中 $R3$ 使用的 $q.x$ 就是刚好被 $R1$ 重新定义了, 因此 $R3$ 并不会数据流依赖于方法入口传入的 x 域取值. 本文称 $R1$ 这样的定义为相对更新, 它不能淘汰某个可达定义, 但是对于个别特殊引用, 可以起到类似强更新的效果, 使该可达定义不可见.

相对更新可以通过可达引用分析来识别. 一个

可达引用具有形式 $\langle m, ap \rangle$, 表示 m 点引用了访问路径 ap 的取值. 语句 m 数据流依赖于语句 d , 仅当 m 上生成的一个可达引用能够传播到 d . 可达引用集可由后向数据流迭代^[2]来求解, 其计算与可达定义计算是对称的. 在过程间, 可达引用的计算也与 5.2 节的可达定义计算类似, 被调方法内的引用先被映射为该方法入口的引用, 再被射为施调方法内的引用. 由于篇幅限制, 对于可达引用的计算, 本文就不再作详细阐述.

前向确定别名可用来在可达引用分析中进行 “相对更新”. 设赋值语句 n 能产生定义 $\langle n, ap', l \rangle$, 对于从后继传入的可达引用 $\langle m, ap \rangle$, 若 m 点有前向确定别名 $\langle n; ap', ap \rangle$, 则此引用将不能从 n 向前传播, 因为 n 点上最近一次定义的正是 m 点上引用的空间, 前面关于 ap 所表示空间的定义将不可能对引用 $\langle m, ap \rangle$ 产生影响. 在图 1 的 `reset()` 方法中, $R3$ 上将生成一个引用 $\langle R3, q.x \rangle$. 它会被传播到 $R1$ 点. 由于 $R1$ 最近一次出现中定义的 $p.x$ 一定和 $R3$ 上引用的 $q.x$ 表示相同的物理空间, 该引用不会继续向方法入口传播, $R3$ 对 $q.x$ 的引用也不会依赖于方法入口处传入的 `Node` 对象 x 域取值. 对于方法调用, 由于被调方法内的引用已转化成了施调方法内的引用, 过程间的相对更新, 与 5.2 节类似, 变成过程内相对更新.

6 实验分析

为验证本文所提出方法的有效性, 我们在 Soot^① 字节码分析平台上实现了一个带有堆空间上可达定义强更新和相对更新的 Java 程序依赖分析系统原型. 它实现了前向和后向点间确定别名的分析, 并在此基础上通过计算可达定义集和可达引用集获得程序中的依赖关系. 其中别名分析以 Spark^[13] 的流不敏感、上下文不敏感指向分析为基础.

表 1 实验对象

程序名称	描述	可达方法数	实际分析方法数
print	一个用 java.io 库打印简单对象信息的程序	3390	140
list	一个建立并遍历 java.util.LinkedList 列表的程序	3392	145
net	一个用 java.net.Socket 创建网络连接的程序	3534	1225
gui	一个用 java.awt.Frame 显示窗口的程序	5247	3240
graph	一个用 graphviz ^② 库解析 dot 格式文件的程序	3866	1562

实验中, 我们主要分析研究了 Java 1.4.1 标准库中的代码. 所有实验结果均在一台装有 Windows XP 的 1GHz Pentium CPU 和 1GM RAM 的机器上获得. 表 1 列出了本实验采用的实验对象. 其中可达

方法数是在 Soot 默认方式产生的调用图上从程序入口开始可达的方法的总数, 而实际分析方法数是

① <http://www.sable.mcgill.ca/soot/>
② <http://www.graphviz.org/>

本实验实际分析的方法的总数. 后者比前者小很多首先是因为我们忽略了许多负责日志维护等操作的无副作用方法, 以保证分析的效率; 另外, 我们也没有考虑异常机制以及多线程. 前 4 个实验对象主要用于验证本文所提出方法对于 Java 标准库的效果, 而最后一个实验对象则旨在探究该方法对于 Java 应用程序的效果.

表 2 列出了本实验的结果, 其中优化前依赖分

析时间包括可达定义分析的时间和依赖图构造的时间, 优化后的依赖分析时间除此之外还包括可达引用的分析时间. 堆空间依赖指关于堆空间的数据流依赖. 这里我们并未列出整个分析中其它过程消耗的时间(比如构造中间表示的时间、指向分析的时间和副作用分析的时间等), 因为它们在优化前后的依赖分析中并没有改变.

表 2 访问路径长度限制为 2 时的分析结果

程序名称	别名分析时间/s	依赖分析时间/s		堆空间依赖数		堆空间依赖的精度提高/%
		优化前	优化后	优化前	优化后	
print	6.5	5.1	6.4	711	610	14.2
list	7.0	5.4	6.9	754	639	15.3
net	176.9	138.7	154.8	6432	5638	12.3
gui	1669.5	1103.7	1411.5	17392	14283	17.9
graph	322.4	233.6	292.8	8449	7122	15.7

分析结果中 print 和 list 得到的数据非常接近, 因为它们本身的规模很接近. 两者的结果分别反映了本文的方法在 Java 的输入输出库和容器库上的效果. 5 个实验对象的实验结果表明该方法在 Java 标准库及其应用程序上效果都很明显. 这些实验中, 分析 net, gui 和 graph 程序所消耗的时间增长很快, 这主要是因为大量的内存消耗使得 Java 虚拟机的性能迅速下降, 而不是因为算法本身. 事实上, 尽管分析时间增长较快, 优化前和优化后依赖分析所消耗的时间在比例上还是维持在一个较稳定的水平.

本实验的结果表明, 尽管别名与可达引用的分析在一定程度上增加了依赖计算的复杂性, 但是该复杂性增长相对于原有的依赖关系计算是非常有限的. 对于几个实验对象, 整个依赖分析算法所消耗的时间几乎没有出现数量级上的增长. 其中, 主要的原因是无论是点间别名还是可达引用的分析都是上下文不敏感的, 而面向对象程序设计追求较小的模块、较多的消息传递, 因此少量的过程内数据流迭代并不会从根本上影响依赖分析的效率. 另一方面, 确定别名分析和可达引用分析都易于保证安全性, 实验中, 通过忽略一些别名生成、不考虑一些可达引用, 我们成功地避免了对同一个方法的多次分析(也仅有递归方法需要多次分析), 节省了大量时间. 事实上, 由于方法间差异并不大, 当限定每个方法仅分析一次时, 整个算法的复杂性与方法的个数将呈现接近线性的关系.

本实验亦表明, 利用强更新和相对更新, 堆空间上依赖分析的精度能有明显的提高, 这对于许多后继应用将有重要意义. 事实上, 本文给出的只是访问路径长度限制在 2 以内时的实验结果. 更长的访问

路径长度的上限设置, 可能有更大的精度优化. 对此, 我们将在今后的工作中进一步研究.

7 相关工作

在别名分析领域, 目前主要的研究都集中在可能别名, 特别是指向关系的分析上^[3,14]. 仅有少数学者研究了确定别名的问题. Landi 和 Ryder 提出了一种面向单级指针的确定别名分析方法^[15], 但其别名本质上是点内别名, 难以用于堆空间上的强更新识别. Jagarmathan 等人针对高阶语言提出了一种用 single 标记来标志那些在一次计算中只表示一个物理空间的变量的方法^[16]. 该方法与文献[9]中的方法类似, 它们都试图通过单一指向性来识别不同点上访问路径之间的关系, 其缺点也和文献[9]中的方法相似. 我们在文献[17]中初步研究了点间别名的问题, 但文献[17]仅考虑了前向点间别名及其在 C 程序指向分析中的应用. 除了以上这些方法, SSA 和值编号技术^[2]也可以用来识别点间别名. 但 SSA 仅能识别同一访问路径的不同出现之间的等价关系, 值编号技术只便于处理程序中直接出现的访问路径之间的关系. 它们难以全面识别带有流向信息的点间别名, 也都不便于在过程间分析中采用.

在含指针程序的依赖性分析方面, 文献[7,18]研究了指向分析基础上的依赖分析问题, 但它们并没有对堆内存上的依赖关系分析作深入探讨. 此外, 文献[19-20]研究了数据结构形态分析和依赖分析之间的关系, 文献[1]研究了在有指针情况下的依赖关系分类, 它们所关注的都是与本文不同的方面.

8 结 论

本文提出了前向和后向确定别名两种不同形式的点间别名. 它们提供了一种清晰地描述不同程序点上访问路径间关系的手段, 可用来在依赖分析中进行强更新和相对更新, 从而提高依赖分析的精度. 在以堆命名来抽象堆内存的方式下, 与其它旨在通过强更新来提高堆内存上数据依赖分析精度的方法相比, 基于点间确定别名的方法不但理论上更加清晰, 能更系统、全面地识别强更新、相对更新, 而且具有更好的可配置性, 它既不依赖于特定的指针分析算法, 也不依赖于特定的堆命名方法.

与可能别名不同, 点间确定别名的安全性易于保证, 因此其计算也便于在精度和效率之间进行权衡. 由于本文的点间确定别名分析并不需要上下文敏感性, 对于倾向于多模块、小方法的面向对象程序而言, 其计算的复杂度是可以接受的. 我们的实验研究也表明, 点间确定别名分析可在相对较少的额外时间消耗内, 有效地提高在 Java 程序堆空间依赖分析上的精度.

由于多线程、异常等机制并不会从根本上影响确定别名的分析, 本文没有对此作深入探讨, 今后我们将对这些问题作进一步展开. 此外, 我们还拟进一步拓宽点间确定别名在其它程序分析中的应用.

参 考 文 献

- [1] Orso A, Sinha S, Harrold M J. Classifying data dependences in the presence of pointers for program comprehension, testing, and debugging. *ACM Transactions on Software Engineering and Methodology*, 2004, 13(2): 199-239
- [2] Muchnick S S. *Advanced Compiler Design and Implementation*. USA: Morgan Kaufman Publishers, 1997
- [3] Hind M. Pointer analysis: Haven't we solved this problem yet? // *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. Snowbird, Utah, 2001: 54-61
- [4] Chen Zhen-Qiang, Xu Bao-Wen. An approach to analyzing dependence of concurrent programs. *Journal of Computer Research and Development*, 2002, 39(12): 159-164 (in Chinese) (陈振强, 徐宝文. 一种并发程序依赖性分析方法. *计算机研究与发展*, 2002, 39(12): 159-164)
- [5] Chen Zhen-Qiang, Xu Bao-Wen. An approach to measurement of class cohesion based on dependence analysis. *Journal of Software*, 2003, 14(11): 1849-1856 (in Chinese) (陈振强, 徐宝文. 一种基于依赖性分析的类内聚度量方法. *软件学报*, 2003, 14(11): 1849-1856)
- [6] Chase D R, Wegman M, Zadek F K. Analysis of pointers and structures // *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. White Plains, New York, USA, 1990: 296-310
- [7] Binkley D W, Lyle J R. Application of the pointer state subgraph to static program slicing. *Journal of Systems and Software*, 1998, 40(1): 17-27
- [8] Whaley J, Lam M S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams // *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. Washington DC, USA, 2004: 131-144
- [9] Altucher R Z, Landi W. An extended form of must alias analysis for dynamic allocation // *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*. San Francisco, California, USA, 1995: 74-84
- [10] Larus J R, Hilfinger P N. Detecting conflicts between structure accesses // *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. Atlanta, Georgia, USA, 1988: 21-34
- [11] Burke M, Carini P, Choi J-D, Hind M. Interprocedural pointer alias analysis. IBM T. J. Watson Research Center, Research Report #21055, 1997
- [12] Nielson F, Nielson H R, Hankin C. *Principles of Program Analysis*. 2nd Edition. Secaucus, NJ, USA: Springer-Verlag, 2005
- [13] Lhoták O, Hendren L. Scaling Java points-to analysis using Spark // *Proceedings of the International Conference on Compiler Construction*. Warsaw, Poland, 2003: 153-169
- [14] Ryder B G. Dimensions of precision in reference analysis of object-oriented programming languages // *Proceedings of the International Conference on Compiler Construction*. Warsaw, Poland, 2003: 126-137
- [15] Landi W, Ryder B G. Pointer-induced aliasing: A problem classification // *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*. Orlando, FL, USA, 1991: 93-103
- [16] Jagarmathan S, Thiemam P, Weeks S, Wright A. Single and loving it: Must-alias analysis for higher-order languages // *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, California, USA, 1998: 329-341
- [17] Xu Bao-Wen, Qian Ju, He Yan-Xiang. Point-wise must alias and its application to point-to analysis // *Proceedings of the 9th IASTED International Conference on Software Engineering and Applications*. Phoenix, AZ, USA, 2005: 421-424
- [18] Atkinson D C, Griswold W G. Effective whole-program analysis in the presence of pointers // *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Lake Buena Vista, Florida, USA, 1998: 131-142
- [19] Horwitz S, Pfeiffer P, Reps T W. Dependence analysis for pointer variables // *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. Portland, Oregon, USA, 1989: 28-40
- [20] Amme W, Zehendner E. Data dependence analysis in programs with pointers. *Parallel Computing*, 1998, 24(3-4): 505-525

附 录

定理 1. 过程内前向确定别名分析算法能够终止.

证明. 该别名计算是个数据流迭代过程, 每步迭代处理一个语句. 设程序中所有语句的集合为 N . 对语句 n , 不妨用 $Alias_-[n]_i$ 和 $Alias_+[n]_i$ 表示第 i 步迭代前其入口和出口的别名集.

(1) 首先, 整个数据流迭代前, $\forall n \in N, Alias_-[n]_0, Alias_+[n]_0$ 均初始化为有限集.

(2) 在第 k 步迭代中, $\forall n \in N$, 设 $\forall m \in pred(n) \{Alias_+[m]_k \subseteq Alias_+[m]_{k-1}\}$, 则根据 *merge* 过程的计算规则, $Alias_-[n]_k \subseteq Alias_-[n]_{k-1}$, 即 n 点入口的别名集或者保持不变, 或者其中部分元素因 n 某前驱的出口别名集中相关别名的失效而不再成立.

(3) 在第 k 步迭代中, $\forall n \in N$, 设 $Alias_-[n]_k \subseteq Alias_-[n]_{k-1}$, 由于 $Kill[n]$ 和 $Gen[n]$ 不随迭代变化, 根据别名计算规则, *transit* 操作后所得的 n 出口的别名集也不会增大, 即 $Alias_+[n]_k \subseteq Alias_+[n]_{k-1}$.

由(1)(2)(3)的结论, 通过数学归纳法可知, 在任一步迭代 k 中有

$$\forall n \in N \{Alias_-[n]_k \subseteq Alias_-[n]_{k-1} \wedge Alias_+[n]_k \subseteq Alias_+[n]_{k-1}\}.$$

整个数据流迭代是各点别名集单调减小的过程, 它必有一个不动点, 算法最终一定收敛. 证毕.

定理 2. 过程内前向确定别名分析算法是语义正确的.

证明. 由于 Java 语言的复杂性, 对整个语言构造别名分析的正确性证明非常困难. 为此, 我们首先定义了一个简化语言 SIMPLE, 它仅包含顺序、if 和 while 三种控制结构, 每条语句是从一个访问路径到另一个访问路径的拷贝, 或者用 null 和 new 表达式对某访问路径的赋值. 该语言表达了 Java 程序的基本特征, 其上别名分析的正确性证明能够反映 Java 程序上别名分析算法的正确性. 附图 1 给出了 SIMPLE 的基本语法, 由于该语言仅用于正确性证明, 我们并未给出所有语言细节, 其中 b 是一个不确定的布尔值, C 是一个类.

```

Program := begin Statement end
Statement := Assignment | Statement; Statement
           | if  $b$  then Statement else Statement
           | while  $b$  do Statement
Assignment := AccessPath = Value | AccessPath = AccessPath
Value := new C | null

```

附图 1 SIMPLE 的基本语法

为构造证明, 我们先为 SIMPLE 定义一个插桩语义 (instrumented semantics). 该语义中, AP 是所有访问路径的集合. 每个赋值有一个标签, $Label$ 是所有赋值语句标签的集合. 访问路径状态映射 $APState: AP \rightarrow Location$ 描述了各访问路径所表示的物理空间. 迹 tr 是一个序列, 它的每个元素是赋值标签和该赋值执行后访问路径状态映射的二元组, 即 $tr \in Trace = (Label \times APState)^*$, ϵ 表示空迹. 该插装语义中的状态变迁有如下形式:

$$\langle S, \sigma, tr \rangle \rightarrow \langle S', tr' \rangle \text{ 或 } \langle S, \sigma, tr \rangle \rightarrow \langle S', \sigma', tr' \rangle,$$

其中 S 是一条语句, S' 是它的后继, σ, σ' 是 S 执行前后的程序状态, tr, tr' 是 S 执行前后的迹. 前一个变迁表示程序执行

S 后结束, 后一个变迁表示程序执行 S 后继续执行 S' .

在该语义中, 我们再定义 tr_n^a 为迹 tr 中语句 n 最近一次执行后访问路径 a 所表示的空间. 若 n 在 tr 中尚未出现, 则 tr_n^a 定义为 \top . 设 tr^a 是迹 tr 中语句序列执行完后访问路径 a 所表示的空间, 则前向确定别名 γ 与迹 tr 的正确性关系 R 可定义为

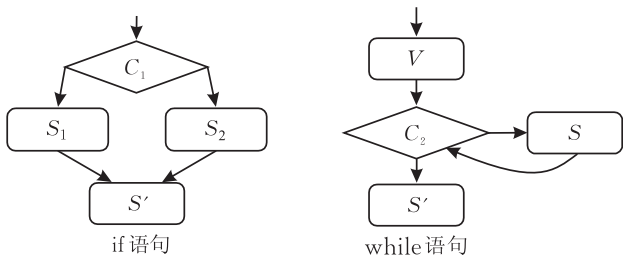
$$tr R \gamma \text{ iff } \forall n \neq \perp, a, a' \{ (\gamma = \langle \perp, a, a' \rangle \rightarrow tr_n^a = tr_n^{a'}) \wedge (\gamma = \langle n, a, a' \rangle \rightarrow (tr_n^a = \top \vee tr_n^a = tr_n^{a'})) \}.$$

别名集 Φ 与迹 tr 存在正确性关系 R 当且仅当其中每个元素都满足正确性关系, 即

$$tr R \Phi \text{ iff } \forall \gamma \in \Phi, tr R \gamma.$$

$tr R \Phi$ 表明别名集 Φ 能够正确地描述程序执行到当前点 (产生轨迹 tr) 时, 相同或不同点上访问路径间的前向确定别名关系.

证明前向确定别名分析的正确性即要证别名分析的结果与迹的正确性关系 R 在每条语句 S 计算前后都能得到保证. 设 $\Phi(S)$ 是别名分析所获得的 S 语句入口的别名集 (对于赋值语句, $\Phi(S) = Alias_-[S]$; 对于顺序语句 $S = S_1; S_2$, $\Phi(S) = Alias_-[S_1]$; 对于 if 语句 $\Phi(S) = Alias_-[C_1]$, 如附图 2 所示, C_1 是 if 语句的控制流入口; 对于 while 语句 $\Phi(S) = Alias_-[V]$, 如附图 2 所示, V 是表示循环入口的虚拟节点), 现要证对任一语句 S 及其执行前的迹 tr , 有 $tr R \Phi(S)$.



附图 2 if 和 while 语句的控制流结构

设程序第一条语句 S_0 执行前的迹为 tr_0 , tr_0 是在空迹 ϵ 上增加一条表示程序 *entry* 的空赋值后的结果. 对 tr_0 , 显然 $\forall a \in AP, tr_0^a = tr_0^a$. 由于 *entry* 点出口的别名集中仅有形如 $\langle entry: a, a \rangle$ 的别名, S_0 入口的别名集就是 *entry* 点出口的别名集, 根据正确性关系 R 的定义一定有 $tr_0 R \Phi(S_0)$, 即程序第一条语句执行前的别名集满足正确性关系 R . 而程序执行后的别名集别名分析算法并未定义, 因此对于变迁 $\langle S, \sigma, tr \rangle \rightarrow \langle S', \sigma', tr' \rangle$ 总可通过适当地构造程序结束后的别名集来保证算法的正确性. 要证正确性关系 R 在每条语句 S 计算前后能得到保持, 现只需证对于状态变迁 $\langle S, \sigma, tr \rangle \rightarrow \langle S', \sigma', tr' \rangle$, 总有 $tr R \Phi(S) \rightarrow tr' R \Phi(S')$.

下面我们对几种基本的程序结构进行归纳证明:

(1) 首先, 对于简单赋值语句 $S: a_1 = a_2$. 设 $tr R \Phi(S)$, 我们先来证 $tr' R (\Phi(S) - Kill[S])$, 即要证明 $\forall \langle \perp, a, a' \rangle \in \Phi(S) - Kill[S]$, 有 $tr^a = tr'^a$ 且 $\forall \langle n, a, a' \rangle \in \Phi(S) - Kill[S]$, 有 $tr_n^a = \top \vee tr_n^a = tr'^a$.

由于 $\forall a \in AP$, 若 $Loc(S, a_1) \cap Dec(S, a) = \emptyset$, 即 a_1 的定义不影响 a 的表示, 则 $tr^a = tr'^a$. 而别名集 $\{\langle \times, \times, a \rangle \mid Loc(n, a_1) \cap Dec(n, a) \neq \emptyset\}$ 以及这些别名对应的对称别名均在 $Kill[S]$ 中, 故 $\forall \langle \times, \times, a \rangle \in \Phi(S) - Kill[S]$ 有 $tr^a = tr'^a$.

$\forall \langle \perp; \alpha, \alpha' \rangle \in \Phi(S) - Kill[S]$ 有 $tr'^\alpha = tr^\alpha, tr'^{\alpha'} = tr^{\alpha'}$. 对别名 $\langle \perp; \alpha, \alpha' \rangle \in \Phi(S) - Kill[S]$, 证明 $tr'^\alpha = tr^{\alpha'}$ 只需证 $tr^\alpha = tr^{\alpha'}$. 由 $tr R \Phi(S)$ 可知, 这显然成立. 另一方面, 由于 $\forall \langle \times; \times, \alpha \rangle \in \Phi(S) - Kill[S]$ 有 $tr'^\alpha = tr^\alpha$, 而 $\langle \langle S; \times, \times \rangle \rangle \subseteq Kill[S]$ 使得 $\forall \langle n; \alpha, \alpha' \rangle \in \Phi(S) - Kill[S]$ 有 $tr'_n^\alpha = tr_n^\alpha$ (对于 tr' 中任一 S 以外的语句 S^* , 由于其最近一次出现并未改变, 故 $\forall \alpha \in AP$, 有 $tr_{S^*}^\alpha = tr_{S^*}^{\alpha'}$), 因此 $\forall \langle n; \alpha, \alpha' \rangle \in \Phi(S) - Kill[S]$, 证明 $tr'_n^\alpha = \top \vee tr'_n^\alpha = tr^{\alpha'}$ 只需要证 $tr_n^\alpha = \top \vee tr_n^\alpha = tr^{\alpha'}$. 由 $tr R \Phi(S)$ 可知, 这也显然成立. 综上, $tr' R (\Phi(S) - Kill[S])$ 成立.

根据 $Gen[S]$ 的计算规则, 显然 $\forall \langle \perp; \alpha, \alpha' \rangle \in Gen[S]$ 有 $tr'^\alpha = tr^{\alpha'}$. 由于 $Gen[S]$ 中仅包含形如 $\langle S; \alpha, \alpha \rangle$ 的点间别名, 即 $\forall \langle n; \alpha, \alpha' \rangle \in Gen[S] \{ n = S \wedge \alpha = \alpha' \}$, 而 S 是迹 tr' 中的最后一条语句, 因此 $\forall \langle n; \alpha, \alpha' \rangle \in Gen[S]$, 有 $tr'_n^\alpha = tr^{\alpha'}$. 对于整个别名集 $Gen[S]$, 总有 $tr' R Gen[S]$. 由 $tr' R (\Phi(S) - Kill[S])$ 和 $tr' R Gen[S]$ 可知, $transit$ 操作前所得的别名均满足相应的正确性关系. $transit$ 操作包括点内别名间的传递和点间别名与点内别名的传递两种形式. 对于 $\langle \perp; \alpha, \beta \rangle$ 和 $\langle \perp; \beta, \gamma \rangle$ 的传递, 由于这两个别名都满足正确性关系, 即 $tr'^\alpha = tr'^\beta, tr'^\beta = tr'^\gamma$, 显然对传递结果 $\langle \perp; \alpha, \gamma \rangle$ 有 $tr'^\alpha = tr'^\gamma$; 对于 $\langle n; \alpha, \beta \rangle$ 和 $\langle \perp; \beta, \gamma \rangle$ 的传递, 由于传递前两个别名满足正确性关系, 即 $tr'_n^\alpha = tr'^\beta, tr'^\beta = tr'^\gamma$, 故对传递后的别名 $\langle n; \alpha, \beta \rangle$ 有 $tr'_n^\alpha = tr'^\gamma$. 由上可知传递后所得的新别名也满足正确性关系.

综合起来, 对于赋值语句, 如果其入口的别名集满足正确性关系, 别名分析处理过该语句后所得的所有别名一定都满足正确性关系, $tr R \Phi(S) \rightarrow tr' R \Phi(S')$ 得证.

(2) 对顺序语句 $S = S_1; S_2$. 设 S_1 执行后的迹为 tr^* , 根据归纳假设, 有

$$tr R \Phi(S_1) \rightarrow tr^* R \Phi(S_2), tr^* R \Phi(S_2) \rightarrow tr' R \Phi(S'),$$

由于 $\Phi(S_1) = \Phi(S)$, 固 $tr R \Phi(S) \rightarrow tr' R \Phi(S')$ 得证.

(3) 对 if 语句 $S = \text{if } b \text{ then } S_i \text{ else } S_j$. 若 b 取真, 则根据归纳假设有 $tr R \Phi(S) \rightarrow tr_i R \Phi(S')$, tr_i 是真分支执行后的迹, $\Phi_i(S')$ 是 S_i 执行后的别名集. 式(3)保证:

$\forall \gamma \in \Phi(S') \{ \gamma \in \Phi_i(S') \vee \exists n, \alpha, \alpha' \{ \gamma = \langle n; \alpha, \alpha' \rangle \wedge tr_n^\alpha = \top \} \}$, 即 $tr_i R \Phi_i(S') \rightarrow tr_i R \Phi(S')$, 因此总有 $tr R \Phi(S) \rightarrow tr_i R \Phi(S')$. 同理, 当 b 取假时, 也总有 $tr R \Phi(S) \rightarrow tr_j R \Phi(S')$, tr_j 是执行假分支后得到的迹. 由于 tr' 是 tr_i 和 tr_j 的二取一, 总有 $tr R \Phi(S) \rightarrow tr' R \Phi(S')$.

(4) 对 while 语句 $S = \text{while } b \text{ do } S_b$. $\Phi(S')$ 即 $Alias_{C_2}$, 如附图 2, C_2 是 while 语句控制流结构中的控制条件点. 首先, 当 b 直接取假, 循环体得不到执行时, $tr' = tr$, 故 $tr R \Phi(S) \rightarrow tr' R \Phi(S)$. 式(3)保证 $\forall \gamma \in \Phi(S')$ 有 $\gamma \in \Phi(S) \vee \exists n, \alpha, \alpha' \{ \gamma = \langle n; \alpha, \alpha' \rangle \wedge tr_n^\alpha = \top \}$, 即 $tr' R \Phi(S) \rightarrow tr' R \Phi(S')$, 因此总有 $tr R \Phi(S) \rightarrow tr' R \Phi(S')$.

当 b 取真时, 根据 $tr' = tr$ 时的结论 $tr R \Phi(S) \rightarrow tr' R \Phi(S')$, 有 $tr R \Phi(S) \rightarrow tr R \Phi(S')$. 由于 $\Phi(S')$ 是数据流迭代的不动点, 以 $\Phi(S')$ 为 C_2 入口的别名集, 执行一次循环体, 再回到 C_2 入口时分析所得的别名集仍将是 $\Phi(S')$. 设第一次执行完循环体后、会合前的别名集是 Φ_1 , 迹是 tr_1 , 由于循环体入口的别名是 $\Phi(S')$, 根据对循环体的归纳假设有 $tr R \Phi(S') \rightarrow tr_1 R \Phi_1$. 对于会合后的别名集 $\Phi(S')$, 式(3)保证 $\forall \gamma \in \Phi(S')$ 有 $\gamma \in \Phi_1 \vee \exists n, \alpha, \alpha' \{ \gamma = \langle n; \alpha, \alpha' \rangle \wedge tr_n^\alpha = \top \}$, 即 $tr_1 R \Phi_1 \rightarrow tr_1 R \Phi(S')$, 因此, 总有 $tr R \Phi(S') \rightarrow tr_1 R \Phi(S')$. 同理, 执行第 k 次循环体前后有 $tr_{k-1} R \Phi(S') \rightarrow tr_k R \Phi(S')$. 无论循环执行多少次, 都有 $tr R \Phi(S') \rightarrow tr' R \Phi(S')$, 因而也总有 $tr R \Phi(S) \rightarrow tr' R \Phi(S')$.

由以上分析可知, 对于循环语句, 无论循环如何执行, $tr R \Phi(S) \rightarrow tr' R \Phi(S')$ 都成立.

综合几种语句上的证明可知, 别名分析结果与程序执行轨迹的正确性关系 R 经任一语句 S 都能得到保证, 算法正确性得证. 证毕.



QIAN Ju, born in 1981, Ph. D. candidate. His research interests include software analysis and testing.

XU Bao-Wen, born in 1961, Ph. D., professor, Ph. D. supervisor. His research interests include programming languages, software engineering, parallel and network software, acquisition technique on knowledge and information etc.

ZHOU Yu-Ming, born in 1974, Ph. D.. His major research interest is software engineering.

Background

This work is partly supported by the National Outstanding Young Scientists Foundation of China aiming to provide some useful approaches and tools for software analysis and testing. The mainly concerned problem in this paper is data dependence analysis on heap memory locations. Data dependences are widely used in software engineering activities. The massive use of heap memory locations has already brought lots of problems to the precise analysis of data dependences. To solve the problem, currently most of the researchers try to build more elaborate heap memory abstractions to improve dependence analysis of heap locations. The resulting approaches

have got many achievements. However, yet rarely any of them is practical enough. A more practical solution to high precision data dependence analysis of heap locations could be taking focus off the heap memory abstractions and seeking more strong updates or other kinds of updates that are powerful in discarding superfluous data dependences. This paper just focuses on the strong (or relative) update problem. It uses interstatement must aliases to resolve strong updates. The interstatement must aliases can be used to find more strong (or relative) updates in an acceptable time.