一种利用指向组合优化依赖图构建的方法

张 磊 陶彬贤 钱 巨
(南京航空航天大学计算机科学与技术学院 南京 210016)

摘 要 指针的动态性使得程序分析中一个指针变量往往被认为有多个可能的指向目标,构成多个指向关系。现有的依赖图构建方法虽然较全面地考虑了指针的多指向性,但并未考虑指向关系之间的可组合性,因此精度上仍存在许多不足。为此,提出了一种利用无效指向组合优化依赖图构建的方法,新方法可以排除现有方法所不能识别的伪依赖,从而有效地提高依赖图的构建精度。

关键词 依赖图,指针,组合,别名,程序切片
中图法分类号 TP311.5 **文献标识码** A

Using Points-to Combinations to Optimize Dependence Graph Construction

ZHANG Lei TAO Bin-xian QIAN Ju
(School of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China)

Abstract The dynamic nature of pointers makes a pointer possibly points to many different locations in an execution in program analysis. The existing dependence graph construction algorithms have already taken these multiple points-to relations into consideration. However, they do not consider the combination of points-to relations. Many points-to relations are not combinable. Without excluding these invalid combinations, we may lose precision in dependence graph construction. To address the problem, this paper proposed an approach that uses the invalid combinations of points-to relations to optimize dependence graph construction. The approach can discard many false dependences which cannot be identified by the existing approaches, and thereby improve the precision of dependence graph construction.

Keywords Dependence graph, Pointer, Combination, Alias, Program slicing

1 引言

作为一种重要的程序表示方式,依赖图[6]在软件工程领域有广泛的应用。对于不含指针的程序,其依赖图可以通过分析可达定义[3]、必经节点[6]等信息获得。指针的存在为依赖图的获取增加了很大的困难。通常,为构造依赖图,人们必须首先获得程序中各指针的指向信息,识别间接的内存访问,然后可使用针对不含指针程序的方法构造依赖图。文献[7-10]均采用了这样的思路。

指针的动态性[11]使得程序分析中一个指针变量往往被认为可能有多个指向目标,构成多个可能的指向关系。另一方面,这些指向关系并不总是可组合的,一些指向关系不能同时发生。现有依赖图构建方法在进行依赖分析时,考虑了指针的多指向性,但并没有考虑无效的指向组合。而在程序中,指向关系的组合问题是普遍存在的,忽视它们将会影响依赖图的构建精度。

为改善依赖图的构建,本文以 Java 程序为例,给出了一组利用无效指向关系组合优化依赖图构造的方法。本文利用指向组合方面的限制,识别内存访问相关的约束,进而排除伪依赖。首先讨论了过程内依赖图构建的优化,然后讨论了过程间依赖图构建的优化。新的优化方法将能更多地避免伪依赖,提高依赖图分析的精度。另一方面,本文所使用的组合信息主要来自于别名关系和路径关系,这些信息都可以通过较为高效的方式获得,其可获取性保证了所提优化方法的可行性。

2 相关背景

通常人们用可达定义分析来计算程序中的依赖关系。该分析中,一个二元组$\langle s,d\rangle$表示程序点 s 上对变量 d 的定义。可达定义分析顺着程序流将一个定义传播到它所能影响到的程序点上。如果定义$\langle s,d\rangle$能够传播到语句 t 上,且语句 t 引用了变量 d,那么 t 将被认为依赖于语句 s。

依赖图是在程序点的基础上,通过连接数据流依赖边和控制流依赖边而形成的有向图。过程内依赖图(PDG)仅包含单个过程内的依赖关系,而过程间依赖图(SDG)除了包含各个过程内的依赖关系外,还包括过程间的数据流传递和过程调用造成的控制依赖。

本文主要通过确定别名和路径关系来识别无效的指针访

到稿日期:2012-03-28 返修日期:2012-07-14 本文受国家自然科学基金(60903026)资助。
张 磊(1987—),男,硕士生,主要研究领域为软件分析与测试;陶彬贤(1988—),男,硕士生,主要研究领域为软件分析与测试;钱 巨(1981—),男,博士,副教授,CCF 学生会员,主要研究领域为软件分析与测试,E-mail:jqian@nuaa.edu.cn(通信作者)。

问组合。确定别名(must alias)指两个访问路径(内存访问表达式)关于其表示空间的确定等价关系,它主要由指针等值造成。本文中的确定别名除了包括传统意义上的描述语句内访问路径间关系的确定别名[12]外,还包含新型的用以描述相同或不同程序点上的访问路径间关系的点间确定别名[4]。点间确定别名又可分为多种类型。其中,前向确定别名是典型的点间别名。程序点 m 上的前向确定别名$\langle n:\alpha_1,\alpha_2\rangle$表示当程序流从程序点 n 的最近一次出现流向 m 时,n 上的访问路径 α_1 和当前点 m 上的访问路径 α_2 表示了完全相同的存储空间。在图 1 的例子中,由于 p 在语句 1 和 2 间未被重新定义,因此$\langle 1:p.f,p.f\rangle$是语句 2 上的前向确定别名,即语句 1 执行后的访问路径 $p.f$ 和语句 2 上的访问路径 $p.f$ 访问相同的空间。而由于从语句 6 的角度看,最近的一次语句 4 执行后的 $p.f$ 总是与当前点上的 $q.f$ 表示一样的空间,因此$\langle 4:p.f,q.f\rangle$是语句 6 上的前向确定别名。

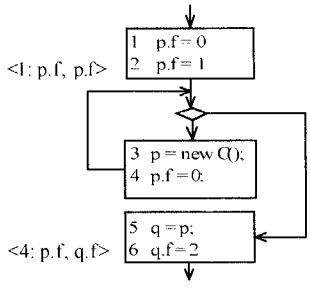

图 1 点间确定别名示例

3 过程内依赖图的优化

本节通过确定别名和路径关系来确定无效的指向组合,进而识别不可能同时存在的存储空间访问,并由此优化过程内依赖图的构建。

3.1 利用语句内确定别名排除伪依赖

一条语句对某存储空间 l 的定义或引用是以相关访问路径访问空间 l 为条件的。比如语句"$p.next=null$"定义对象 o 的 $next$ 属性域 $o.next$ 就以访问路径 $p.next$ 访问存储空间 $o.next$ 为条件。

在一条语句内,由于指针使用方面的限制,不同访问路径在它们所访问的存储空间上可能存在一致性约束。当语句中出现的两个访问路径之间存在确定别名时,它们访问的空间必须是相同的。比如,对于语句:

$\alpha_1.f_1=\alpha_2.f_2$

如果访问路径 α_1 和 α_2 存在确定别名关系,且它们表示的空间集为$\{o_1,\cdots,o_n\}$,则 α_1 和 α_2 只可能访问同一个空间 o_k $(1\leqslant k\leqslant n)$。这种存储空间上的一致性约束也使得该语句上定义和引用的空间之间存在相应的约束,导致语句上只能有从 $o_k.f_1$ 到 $o_k.f_2$ 的依赖。

一般地,令 $L_{lhs}(n)$为程序点 n 上赋值号左边表达式可能表示的空间集,$L_{rhs}(n)$为赋值号右边表达式可能表示的空间集,则对于语句"$n:\alpha_1.\sigma_1=\alpha_2.\sigma_2$"($\sigma_1$、$\sigma_2$ 是两个由域访问或数组元素访问构成的访问路径扩展),其上的定义集 $def(n)$和引用集 $use(n)$可分别表示为:

$$def(n)=\{\langle\alpha_1.\sigma_1\rightarrow l,l\rangle\mid l\in L_{lhs}(n)\} \quad (1)$$

$$use(n)=\{\langle\alpha_2.\sigma_2\rightarrow l,l\rangle\mid l\in L_{rhs}(n)\} \quad (2)$$

这里,$\langle\alpha.\sigma\rightarrow l,l\rangle$表示语句 n 定义或引用存储空间 l 以访问路径 $\alpha.\sigma$ 访问 l 为条件。

根据定义集 $def(n)$和引用集 $use(n)$可知,语句 n 上被定义的变量 l 所依赖的空间集 $use(n,l)$为:

$$use(n,l)=\{x\mid\alpha_1.\sigma_1\rightarrow l\wedge\alpha_2.\sigma_2\rightarrow x\wedge x\in L_{rhs}(n)\} \quad (3)$$

上述公式不能使条件"$\alpha_1.\sigma_1\rightarrow l\wedge\alpha_2.\sigma_2\rightarrow x$"成立的空间都不可能被依赖。当 α_1 和 α_2 存在确定别名关系时,$\alpha_1.\sigma_1$ 和 $\alpha_2.\sigma_2$ 访问的存储空间之间存在约束,对 $\alpha_2.\sigma_2$ 的解析构成了限制,不是所有 $L_{rhs}(n)$中的空间都可能被 $\alpha_2.\sigma_2$ 引用。已有方法并不能识别上述问题,而利用语句内确定别名,可排除现有方法所不能识别的伪依赖。

传统的依赖图构造方法认为一条语句上定义的空间依赖于该语句引用的所有空间,因此只用一个点来表示该语句,这相当不精确。图 2 左边的程序中,设语句 2 上分配的空间被抽象为 o_1,而语句 5 上分配的空间被抽象为 o_2。按照传统的依赖图构造方式对语句 11 上的变量 i 进行程序切片[5],将发现语句 3 也是被 i 间接依赖的,而事实上,i 的计算与语句 3 并不相关。

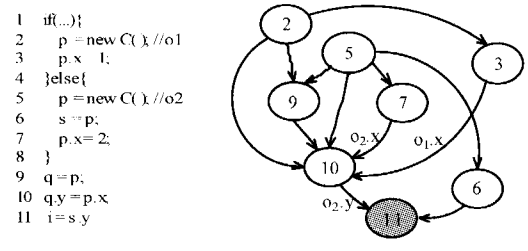

图 2 一段简单的程序及其数据依赖结构

为充分考虑访问路径解析过程中由于语句内确定别名等关系而存在的一致性约束,以提高后继的切片等应用的分析精度,本文在构造依赖图时,对语句 n,如果存在一个空间 l,有 $use(n,l)\subset L_{rhs}(n)$,则针对 l 引入一个 n 的副本,并令该点上只有 $use(n,l)$中空间所造成的依赖边,这样切片过程就能避免遍历伪依赖路径。比如对图 2 的程序,语句 10 对应的依赖点可以拆解为两个节点,它们分别对应空间 $o_1.y$ 和 $o_2.y$,这样对变量 i 进行切片就不会再包含语句 3,节点分解后的数据依赖结构如图 3 所示。

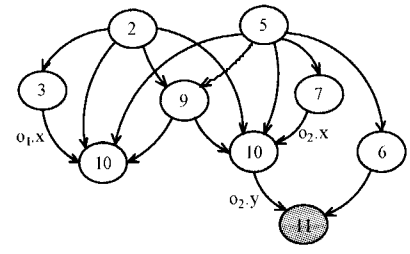

图 3 节点分解后的数据依赖结构

3.2 利用语句间确定别名排除伪依赖

通常在通过可达定义分析获得的依赖关系中,由栈空间和全局空间造成的依赖具有下列性质。

性质 1 在过程内,如果语句 S_2 因为栈空间或全局空间

依赖于 S_1，则 S_2 所依赖的一定是 S_1 的最近一个执行实例。

这是因为 S_2 对 S_1 的依赖表明，S_1 上所定义的是一个栈空间或全局空间。而对定义栈空间或全局空间的语句，它的最近一次执行一定能够覆盖其前面所有次执行的效果。

设 a、b 是栈变量，根据性质 1，对于以下语句序列：

S_1 $a=\alpha_1.f_1$

…… ……

S_2 $b=\alpha_2.f_2$

…… ……

S_3 $\alpha_3.f_3=a+b$

如果语句 S_3 依赖于 S_1 和 S_2，那么被依赖分别是 S_3 前最近的一个 S_1 和 S_3 前最近的一个 S_2。

对上述语句序列，一旦 S_3 上有前向确定别名$\langle S_1:\alpha_1,\alpha_3\rangle$，即从 S_3 的角度看，最近的一个 S_1 上的 α_1 一定与 S_3 上的 α_3 访问相同的空间，那么当 S_3 上的 α_3 访问空间 l 时，被依赖的 S_1 上 α_1 访问的也一定是 l。同样地，一旦 S_3 有前向确定别名$\langle S_2:\alpha_2,\alpha_3\rangle$，那么当 S_3 上的 α_3 访问空间 l 时，被依赖的 S_2 上 α_2 访问的也一定是空间 l。

以上分析表明，由于点间确定别名的存在，不同程序点上的存储空间访问之间也可能存在一致性约束。若不考虑这种一致性，在构造出来的依赖图上就会包含不可实现的数据流路径，从而影响依赖图的应用。设前面 3 条语句 S_1、S_2 和 S_3 中 α_1、α_2 和 α_3 所能访问的空间集是$\{l_1,l_2\}$，图 4(a)给出了这 3 条语句的依赖结构。在该图上，S_1 依赖于对 $l_1.f_1$ 和 $l_2.f_1$ 的定义，S_2 依赖于对 $l_1.f_2$ 和 $l_2.f_2$ 的定义，而 S_3 依赖于 S_1 和 S_2 对 a、b 的定义，同时向外输出对 $l_1.f_3$ 和 $l_2.f_3$ 的定义。当对 S_3 后的 $l_1.f_3$ 进行程序切片时，将需要沿着依赖图追溯 S_1 上 $l_2.f_1$ 的数据流。而根据 S_3 上 α_3 与 S_1 上 α_1 在存储空间访问上的一致性约束，S_1 上访问的一定是 $l_1.f_1$，S_1 上 $l_2.f_1$ 的计算与 S_3 上 $l_1.f_3$ 的计算是不相关的。显然，利用传统的依赖图进行分析，将使大量不相关计算被包含到切片中。

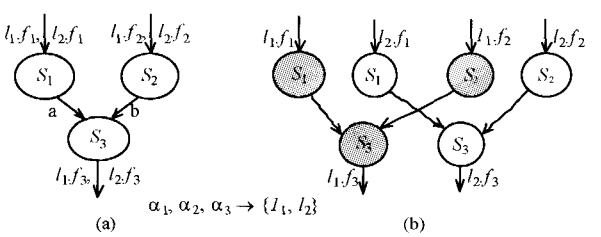

图 4　利用语句间别名进行节点分解的例子

为改进依赖图，使之更好地服务于程序切片等后继应用，与第 3.1 所述的方法类似，将 S_1、S_2 和 S_3 按照它们所定义和引用的空间拆分成多个节点副本，每个副本只定义或引用一个空间，并依据点间确定别名确立的存储空间一致性约束连接依赖边，这样就能避免形成伪依赖路径。图 4(b)给出了对图 4(a)进行节点分解后所得的依赖图，新依赖图上对 S_3 后的 $l_1.f_3$ 切片只需要追踪到 S_1 上的 $l_1.f_1$ 和 S_2 上的 $l_1.f_2$，而不需要再追踪 S_1 上的 $l_2.f_1$ 和 S_2 上的 $l_2.f_2$。

上述优化的关键是要确定对于从语句 Q 到语句 P 的依赖，被依赖的是 P 在 Q 前最近的一个执行实例，而此前更早的 P 的执行实例并没有被 Q 所依赖。只有这样，才能结合点间确定别名识别不同语句上空间访问之间的一致性关系。事实上，判断依赖的是否是最近一个执行实例，除了依据性质 1，还可以根据可达定义[3]信息进行推导。

性质 2　如果语句 P 上的定义$\langle P,l\rangle$传递到 Q 造成了 Q 对 P 的依赖，而语句 P 自身入口却没有到它的定义$\langle P,l\rangle$，则 Q 所依赖的一定是 P 最近的执行实例。

上述性质中，没有定义$\langle P,l\rangle$传递到语句 P 的入口，表明 P 上只有最近一次的定义能够向下传播。这或者是因为 P 没有出现在循环中，或者是因为 P 对 l 的定义总是在传到下一个 P 出现的过程中被其它定义覆盖。

3.3　利用路径关系排除伪依赖

除了确定别名外，执行路径也可能约束指向关系，进而约束存储空间访问的组合。如果两个内存访问依赖于不同的执行路径，这些访问也可能不能同时发生。

在图 5 的例子中，令 o_1、o_2、o_3、o_4 分别代表语句 2、3、5、6 上分配的 4 个不同的对象。在语句 8 上指针 p 可能指向$\{o_1,o_3\}$，指针 q 可能指向$\{o_2,o_4\}$。在一般的依赖分析中，我们将语句 8 解释为 4 种可能的赋值：$o_1.f=o_2.f$，$o_1.f=o_4.f$，$o_3.f=o_2.f$ 和 $o_3.f=o_4.f$。而事实上，p 指向 o_1 与 q 指向 o_4，和 p 指向 o_3 与 q 指向 o_2 是不可能存在的组合。语句 8 只能被解释为 $o_1.f=o_2.f$ 或 $o_3.fo_4.f$。

```
1. if(…){
2.    p =new C( );//o1
3.    q =new C( );//o2
4. }else{
5.    p=new C( );//o3
6.    q=new C( );//o4
7. }
8. p.f=q.f;
```

图 5　执行路径约束单一点上内存访问组合的示例

为更准确地处理图 5 中所示的这种情况，本文引入集合 $\pi(n,\alpha\rightarrow l)$表示能使程序点 n 上的访问路径 α 表示存储空间 l 的路径集，则对于语句“$n:\alpha_1.\sigma_1=\alpha_2.\sigma_2$”，3.1 节的式(1)、式(2)可进一步放松为：

$$def(n)=\{\langle\pi(n,\alpha_1.\sigma_1\rightarrow l),l\rangle\mid l\in L_{lhs}(n)\} \quad (4)$$

$$use(n)=\{\langle\pi(n,\alpha_2.\sigma_2\rightarrow l),l\rangle\mid l\in L_{rhs}(n)\} \quad (5)$$

式中，$\langle\pi(n,\chi\rightarrow l),l\rangle$表示 n 点上访问路径 χ 访问 l 以程序执行 $\pi(n,\chi\rightarrow l)$中的路径为条件。类似地，式(3)可被放松为：

$$use(n,l)=\{x\mid\pi(n,\alpha_1.\sigma_1\rightarrow l)\cap\pi(n,\alpha_2.\sigma_2\rightarrow x)\neq\varnothing\wedge x\in L_{rhs}(n)\} \quad (6)$$

通过检查路径集 $\pi(n,\alpha_1.\sigma_1\rightarrow l)$和 $\pi(n,\alpha_2.\sigma_2\rightarrow x)$是否相交，即可判断 n 点上 $\alpha_1.\sigma_1\rightarrow l$ 和 $\alpha_2.\sigma_2\rightarrow x$ 这两个空间访问是否可组合。准确地判断 $\pi(n,\alpha_1.\sigma_1\rightarrow l)$和 $\pi(n,\alpha_2.\sigma_2\rightarrow x)$是否相交较为困难，一个保守但高效的判断方法是找出空间 l 和 x 所属对象的生成点，一旦这两个点之间不存在一条程序路径，就认为集合 $\pi(n,\alpha_1.\sigma_1\rightarrow l)$和集合 $\pi(n,\alpha_2.\sigma_2\rightarrow x)$不可能相交。

比如对图 5 的例子，由于 o_1 与 o_4 以及 o_2 和 o_3 之间不存

在程序路径，因此 $\pi(8,p.f\rightarrow o_1.f)$ 和 $\pi(8,q.f\rightarrow o_4.f)$ 不可能相交，$\pi(8,p.f\rightarrow o_3.f)$ 和 $\pi(8,q.f\rightarrow o_2.f)$ 也不可能相交，最终只能有 $use(8,o_1.f)=\{o_2.f\}$，$use(8,o_3.f)=\{o_4.f\}$。

获得了各个 $use(n,l)$ 集合后，采用与第 3.1 节中相同的节点拆分方法就可以构造更准确的依赖图。

与图 5 不同，图 6 给出了一个执行路径约束不同点上内存访问组合的示例。该例子中：

$use(8)=\{\langle\pi(8,p.f\rightarrow o_1.f),o_1.f\rangle,\langle\pi(8,p.f\rightarrow(o_3.f),o_3.f)\}$

$def(9)=\{\langle\pi(9,q.f\rightarrow o_2.f),o_2.f\rangle,\langle\pi(9,q.f\rightarrow o_4.f),o_4.f)\}$

由于 $\pi(8,p.f\rightarrow o_3.f)$ 和 $\pi(9,q.f\rightarrow o_2.f)$ 不相交，$\pi(8,p.f\rightarrow o_1.f)$ 和 $\pi(9,q.f\rightarrow o_4.f)$ 不相交，因此，当语句 9 定义 $o_2.f$ 时，语句 8 不会引用 $o_3.f$；而当语句 9 定义 $o_4.f$ 时，语句 8 不会引用 $o_1.f$。通过采用与第 3.2 节类似的方法，对依赖图上的相邻语句 8 和 9 进行拆分，分别建立依赖边，将能够获得更精确的依赖图，以便于后继应用的处理。

```
1. if(…){
2.    p =new C( );//o1
3.    q =new C( );//o2
4. }else{
5.    p=new C( );//o3
6.    q=new C( );//o4
7. }
8. a=p.f;
9. q.f=a;
```

图 6　执行路径约束不同程序点上内存访问组合的示例

4　过程间依赖图的优化

动态绑定使得一个虚方法调用语句可能调用多个不同的目标方法，安全的程序分析必须考虑所有这些可能的调用。现有的依赖图构建方法通常将虚方法调用当作 if/switch 这样的条件结构来处理，将一个调用点拆分为多个独立调用来分别进行分析。这样做虽然简单安全，但它没有考虑由指针使用限制造成的多个多态调用之间的组合约束，使得依赖图构造中考虑了许多不可能执行的路径，从而产生了伪依赖边，最终影响分析的精度。

4.1　利用点间确定别名排除过程间伪依赖

本章首先利用点间确定别名来排除过程间伪依赖。在此方面，以图 7 中的 $foo()$ 方法为例，其中 S_1 点可能调用方法 $A.m()$ 或 $B.m()$，S_2 点可能调用方法 $A.n()$ 或 $B.n()$。当不知道 S_1 上变量 a 和 S_2 上变量 b 之间的关系时，将得到图 7 右边的依赖结构（该图仅列出了过程调用相关的依赖边，其中每个节点 S_1 或 S_2 派生出两个实际可能发生的调用，in 表示被调方法的输入参数，ret 表示方法返回值，粗边表示控制流转移的控制依赖边，细边和点状边表示参数传递的过程间数据流依赖边）。图 7 中的点状边都是伪依赖边，因为在 $foo()$ 的一次执行中，a 和 b 只可能同时具有一种类型，不可能存在 A 类型方法和 B 类型方法之间的依赖边。

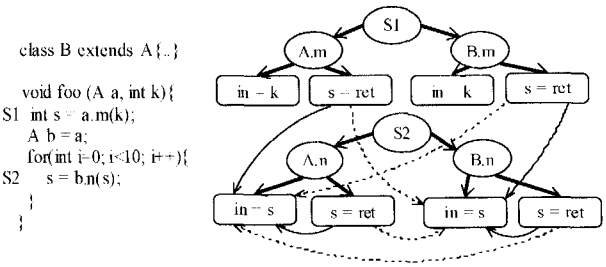

图 7　利用点间确定别名优化过程间依赖图构建示例

事实上，令 * 表示任一属性域，通过分析，可以得到 S_2 上的前向确定别名 $\langle S_1:a.*,b.*\rangle$，它表明从 S_2 的角度看，最近一次 S_1 上的 a 总是与当前的 b 具有相同的类型。因此，如果当前 S_2 上调用的是 $A.n()$，那么最近一个 S_1 上调用的一定是 $A.m()$；如果 S_2 上调用的是 $B.n()$，那么最近一个 S_1 上调用的一定是 $B.m()$。对于从 S_2 的方法输入到 S_1 的方法输出的依赖，一旦根据性质 1 或性质 2 判定被依赖的一定是 S_1 的最近一次执行，就可以用 S_1 和 S_2 上 a、b 类型之间的一致性来识别伪依赖。

在图 7 中，从 S_1 调用到 S_2 调用的点状边表示的都是由栈变量导致的依赖，根据性质 1 可判定被依赖的都是 S_1 语句的最后一次执行，结合前向点间确定别名 $\langle S_1:a.*,b.*\rangle$ 可以判断它们都是伪依赖。类似地，通过 S_2 上的前向确定别名 $\langle S_2:b.*,b.*\rangle$，可以发现连续的两个 S_2 执行中调用的一定是同样类型的方法，从而发现 S_2 调用之间的点状边也是伪依赖边。

4.2　利用路径关系排除过程间伪依赖

利用执行路径对指向组合的限制，同样可以排除过程间伪依赖。在一个程序中，某一程序点 s_1 上变量 a 具有 A 类型和某一程序点 s_2 上变量 b 具有 B 类型可能受执行路径的限制而不会发生在同一次执行中。比如图 8 中，如果 S_1 上的变量 p 具有 A 类型，那么 if 语句必然执行的是真分支，S_2 上的 q 必定也具有 A 类型；同样，如果 S_1 上的 p 具有 B 类型，那么 S_2 上的 q 也一定具有 B 类型。p 具有 A 类型同时 q 具有 B 类型，或者 p 具有 B 类型同时 q 具有 A 类型的情况永远不会发生。

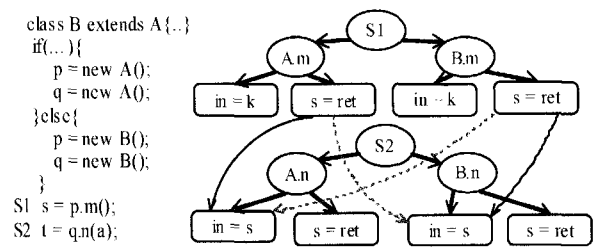

图 8　利用路径关系优化过程间依赖图构建示例

这种由执行路径造成的对指针所指向对象类型组合的限制，将使得一些方法调用不具有可组合性。比如图 8 中不可能存在 S_1 点调用 $A.m()$ 且 S_2 点调用 $B.n()$ 的情况，也不可能存在 S_1 点调用 $B.m()$ 且 S_2 点调用 $A.m()$ 的情况。这也意味着图 8 中以上述调用组合为基础的点状参数传递依赖永远都不可能发生，它们是伪依赖。

为识别更多的伪依赖，用集合 $call(n)$ 表示语句“$n:t=r.$

$m()$"上可能发生的调用集。

$$call(n)=\{\langle n:r \diamond T, T.m\rangle\}$$

式中，r 是指向方法接收对象的引用型变量，T 是接收对象的类型。$r\diamond T$ 表示变量 r 具有 T 类型，而 $\langle n:r\diamond T, T.m\rangle$ 表示 n 点上对方法 $T.m()$ 的调用以 r 访问 T 类型的对象为条件。

令 $\pi(n:r\diamond T)$ 表示能够使得 n 点上 r 访问 T 类型对象的路径的集合，则 $call(n)$ 可以进一步放松为：

$$call(n)=\{\langle \pi(n:r\diamond T), T.m\rangle\}$$

从 s_1 点上方法调用 $T_1.m_1()$ 到 s_2 点上方法调用 $T_2.m_2()$ 的有效数据流必须满足下列条件：

$$\pi(s_1:r_1\diamond T_1)\bigcap\pi(s_2:r_2\diamond T_2)\neq\emptyset$$

式中，r_1 和 r_2 分别是 s_1 和 s_2 上方法调用的接收对象。检查使得 r_1 具有 T_1 类型的变量定义点和使得 r_2 具有 T_2 类型的变量定义点是否能够发生在同一条路径上就能判断此条件是否成立，从而判定一条依赖是否是伪依赖。

本文第 4.1 节和第 4.2 节给出了利用点间确定别名和路径关系优化过程间依赖图构建的方法。多态调用在面向对象语言中极为普遍，利用这些方法将能避免大量的过程间伪依赖。特别地，在程序切片中，避免了一条过程间伪依赖，就可能避免将一个方法包含到最终的切片结果中，这将有效地减小最终切片的大小。

5 确定别名与路径关系的计算

本文的依赖图优化主要使用到确定别名关系和路径关系。对于语句内访问路径间的确定别名关系，目前已经有一些较为高效的计算方法，包括基于数据流迭代的方法[11,12]、基于编译中值编号技术的方法[13]和基于可能和确定信息组合分析的方法[14]等。利用这些方法可在较短的时间内获得确定别名的信息。

关于描述不同程序点上的访问路径间关系的点间确定别名，文献[4]已经给出过一种基于数据流迭代的计算方法。该文献中的实验研究表明，这种别名也可以通过较为高效的方式获得。

对于路径关系，本文主要需要获得每个指针指向对象的分配点和指向关系的建立点。前者通过读取程序行文即可直接获得，后者通过扫描指针赋值语句也可取得。这两种信息都可结合指针分析的过程获取，并不需要设计单独的算法来进行复杂的计算。

上述信息的可获取性，表明本文基于指针组合的优化依赖图构建的方法是可行的。

在程序切片方面，利用所构建的依赖图进行切片与一般的程序切片过程并无差别。事实上，本文只是针对已有依赖图构建中未考虑指针组合的问题给出了一种建立更精确数据依赖边的方法，而没有定义新的切片算法。已有的各种切片方法[2,5,8,9]在所得的依赖图上均同样适用。由于提高了依赖图的构建精度，我们认为多种切片算法均可从中获益。

结束语 本文提出了一组利用指向组合避免不可能存在的存储空间访问组合，从而优化依赖图构建的方法。在实际程序中，大量的空间访问组合是不可能发生的。利用这些不可实现的组合可以提高过程内和过程间依赖图的构建精度，为更精确的后继分析奠定基础。文中仅对指向组合造成的依赖图构建问题进行了理论分析，在后继工作中，还将选择实际程序进行实验研究，以更充分说明所提方法的有效性。

参 考 文 献

[1] Chase D, Wegman M, Zadek F. Analysis of pointers and structures[C]// Proceedings of the SIGPLAN Conference on Program Language Design and Implementation. 1990: 96-310

[2] Horwitz S, Reps T, Binkley D. Interprocedural slicing using dependency graphs[J]. ACM Transaction on Programming Languages and Systems, 1990, 22(1): 26-60

[3] Nielson F, Nielson H R, Hankin C. Principles of Program Analysis (2nd ed.)[M]. Springer, 2005

[4] Qian Ju, Xu Bao-wen, Min Hong-bo. Interstatement Must Aliases for Data Dependence Analysis of Heap Locations[C]// International workshop on Program Analysis for Software Tools and Engineering. 2007

[5] Xu Bao-wen, Qian Ju, Zhang Xiao-fang, et al. A brief survey of program slicing [J]. ACM SIGSOFT Software Engineering Notes, 2005, 30(2): 10-45

[6] Ferrante J, Ottenstein K, Warren J. The program dependence graph and its use in optimization[J]. ACM Transactions on Programming Languages and Systems, 1987, 9(3): 319-349

[7] Atkinson D C, Griswold W G. Effective whole-program analysis in the presence of pointers[C]// Proceedings of the 6th ACM International Symposium on the Foundations of Software Engineering. 1998: 46-55

[8] Livadas P E, Rosenstein A. Slicing in the presence of pointer variables[R]. SERC-TR-74-F. Computer Science and Information Services Department, University of Florida, Gainesville, FL, June 1994

[9] Lyle J R, Binkley D. Program slicing the presence of pointers[C]// Proceedings of the 3rd Annual Software Engineering Research Forum. Orlando, FL, Nov. 1993

[10] Orso A, Sinha S, Harrold M J. Classifying data dependences in the presence of pointers for program comprehension, testing, and debugging[J]. ACM Transactions on Software Engineering and Methodology, 2004, 13(2): 199-239

[11] Hind M. Pointer analysis: Haven't we solved this problem yet [C]// ACM Workshop on Program Analysis for Software Tools and Engineering. June 2001

[12] Landi W, Ryder B G. Pointer-induced aliasing: A problem classification[C]// Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages. January 1991: 93-103

[13] Bodden E, Lam P, Hendren L. Object representatives: a uniform abstraction for pointer information[C]// Proceedings of the International Conference on Visions of Computer Science. 2008: 391-405

[14] Godefroid P, Nori A V, Rajamani S K, et al. Compositional may-must program analysis: unleashing the power of alternation[C]// Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2010: 43-56