计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2015.20140336
52(7):1620-1630, 2015

流敏感按需指针别名分析算法

逄 龙 苏小红 马培军 赵玲玲
(哈尔滨工业大学计算机科学与技术学院 哈尔滨 150001)
(hitpanglong@gmail.com)

Research on Flow Sensitive Demand Driven Alias Analysis

Pang Long, Su Xiaohong, Ma Peijun, and Zhao Lingling
(*School of Computer Science and Technology, Harbin Institute of Technology, Harbin* 150001)

Abstract In order to improve the response efficiency of pointer alias queries in the interactive environment, researches are interested in the demand driven strategy to reduce the cost for analyzing the unrelated pointer variables with respect to the objectives. The demand driven alias analysis based on the context free grammar has been proposed. However, its precision is only limited to the flow-insensitivity. The flow-insensitivity pointer alias restricts the precision of overlying analysis, so the bug detection results in more false alarms than the one with flow-sensitive alias analysis. In this paper, we propose a demand driven pointer alias analysis based on the graph reachability and the context free grammar to provide the flow sensitive precision, which has tolerable additional overhead comparing with the flow-insensitive alias analysis. First the updates of pointer variables are discriminated by the partial single static assignment to filter out the unrelated pointer variables as early as possible. Then the sequence of control flow along these assignments is expressed in the form of level linearization code, which is used to construct the assignment flow graph. Finally, the query of alias in demand driven is formalized as the search of reachability of target nodes in the assignment flow graph to achieve the precision of flow sensitivity. The experiments demonstrate that this presented method can improve the flow sensitivity the precision of alias analysis in demand driven with overhead tolerable.

Key words alias analysis; flow sensitivity; demand driven search; context free language; graph reachability

摘 要 为了提高交互环境下指针别名查询的响应效率,近期研究提出通过只分析与目标相关指针的按需分析策略来降低浪费在与目标无关的指针分析的额外开销.典型的代表是基于上下文无关文法的按需别名分析算法.但是,该算法的精度只局限于控制流不敏感.控制流不敏感的别名关系将约束上层分析的精度.针对该不足,提出了具有流敏感精度的按需别名分析算法.首先采用不完全静态单赋值语句形式来区分指针变量赋值实例,然后通过层次线性化编码方法来表达控制流图中的流敏感信息以构建赋值流图,最后将别名关系查询问题转换为在赋值流图上搜索目标结点间在控制流可达条件下赋值路径的可达性问题,进而实现流敏感的按需别名分析.实验表明,与流不敏感的按需别名分析相比,该方法可以在保证查询效率的前提下,有效提高按需别名分析的精度.

关键词 别名分析;流敏感精度;按需查询;上下文无关语言;图可达性

中图法分类号 TP311

收稿日期:2014-04-16;**修回日期**:2014-10-13
基金项目:国家自然科学基金项目(61173021)

在含有指针运算的编程语言中,别名关系是指由若干指向相同内存位置的指针表达式构成的等价关系.通过指针或数组等间接引用来访问数据是运算语句获取参数和存储结果的主要方式,通过别名关系可以分析出这些语句的实际作用范围.但是,当与上层程序分析目标实际相关的语句比较稀疏时,会导致大量时间和空间被浪费于计算无用的别名关系.针对这个问题,近期有学者提出了基于上下文无关文法的按需别名分析方法[1].该方法首先计算低开销的索引结构,然后在查询别名关系时,通过该结构动态地计算其别名关系.但是该方法分析精度局限于流不敏感.流敏感的指针分析区分指针变量在控制流不同位置的指向信息,而流不敏感的指针分析是综合指针变量在所有位置的指向信息,流敏感与否对上层分析精度有重要影响.如图 1 所示,在流不敏感指针分析中指针变量 s 指向集合中含有空指针 null,基于此结果的空指针引用检测会报告第⑦行存在缺陷.但是,根据流敏感指针分析可知,指针变量 s 在该处实际只指向变量 p,所以该报告属误报.流不敏感的别名信息制约了上层分析的精度[2].因此,如何在保证效率的前提下提高按需别名分析的精度仍然是程序分析领域内具有重要的研究意义和应用价值的关键问题.

当前具有流敏感精度的指针分析算法难以直接实现指针别名关系的高效按需分析.该类指针分析算法是通过计算指向集是否相交来判断别名关系.程序按需分析是指在给定分析目标后,只计算与目标相关的信息,而不计算全局信息的一种分析策略[3-4].其出发点是有针对性地避免与分析目标无关的冗余信息计算,进而提高在分析目标小且查询频率低环境下的程序分析响应速度.但是,当前流敏感精度的指针分析算法依赖于所有指针的指向集,且指向集的计算独立于目标别名查询,使得该类算法的流敏感精度难以直接推广到按需别名分析中.

Code Snip	Flow-Sensitive Point-to Set	Flow-Insensitive Point-to Set
① $p=0$;	① $t\to\varnothing, s\to\varnothing$;	① $t\to s, s\to q$, null, p;
② $q=1$;	② $t\to\varnothing, s\to\varnothing$;	② $t\to s, s\to q$, null, p;
③ $s=\&q$;	③ $t\to\varnothing, s\to q$;	③ $t\to s, s\to q$, null, p;
④ $t=\&s$;	④ $t\to s, s\to q$;	④ $t\to s, s\to q$, null, p;
⑤ $*t=$null;	⑤ $t\to s, s\to$null;	⑤ $t\to s, s\to q$, null, p;
⑥ $*t=\&p$;	⑥ $t\to s, s\to p$;	⑥ $t\to s, s\to q$, null, p;
⑦ $*s=2$;	⑦ $t\to s, s\to p$;	⑦ $t\to s, s\to q$, null, p;

Fig. 1　Example for flow-sensitive and insensitive point-to set.

图 1　流敏感与流不敏感指向集合对比

针对以上问题,本文在分析了流敏感信息表达形式与图可达性关系的基础上,提出基于层次线性化编码的按需别名分析方法.总体技术路线如图 2 所示.首先从程序中提取与指针操作相关的语句,并构通过层次线性化编码(level linearization coding, LLC)来表达流敏感信息,进而构建含有控制流次序信息的赋值流图(assignment flow graph, AFG),以约简后续搜索范围.然后,当给定别名查询目标的需求后,在 AFG 上以对应结点为入口、搜索满足控制流次序并且满足别名关系的上下文无关语言(context free language, CFL)约束的赋值流,从而在满足按需查询策略的条件下保证构成别名关系的控制流的有效性.其中,控制流次序约束检测通过基于 LLC 图可达性的流敏感信息的按需查询来实现.最后,如果存在赋值流则查询目标间构成别名关系,否则不构成别名关系.实验结果表明,该算法能够在保证查询效率的前提下,有效提高按需别名分析的精度.

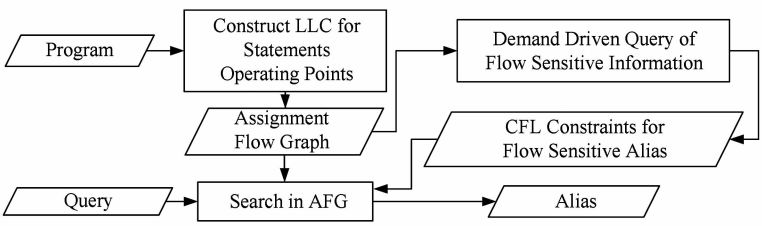


Fig. 2　Design for flow-sensitive demand-driven alias analysis.

图 2　流敏感的按需指针别名分析算法框图

1　流敏感对别名关系按需查询的影响分析

本节分别从别名分析的按需实现技术和流敏感信息表达方法 2 个方面来讨论流敏感精度对当前的流不敏感按需别名分析的影响,分析流敏感精度的按需别名分析中所需解决的问题.

1) 按需别名分析

近期文献[1]将别名分析和指针指向分析进行了区分,并利用 CFL 可达性问题来表达别名关系,

通过按需搜索赋值关系来实现按需判断别名关系. CFL 可达性问题是用来表达程序分析问题的主要方法之一. 当给定一个含有边标签的有向图时,该图的结点间关系 R 可以转换为 CFL 可达性问题. 表达的方法是建立一个满足如下关系的上下文无关文法 G:

假设构成关系 R 的 2 个结点为 n 和 n',从结点 n 到 n' 间存在一条路径,由这条路经所含边对应的标签序列 S,文法 G 构造的语言 $L(G)$ 包含序列 S.

该方法通过文法 G 指导搜索路径,其优势在于避免对全局指向信息的依赖,有效地过滤了与目标别名无关的其他别名关系,进而实现了针对别名关系的按需分析策略.

但是,该方法不具备流敏感分析精度,主要原因在于没有区分含标签边对应语句的顺序. 由于程序动态执行的不确定性,静态分析下语句间不具有完全的执行先后可比性,执行顺序只具有偏序关系. 为了保证别名分析的流敏感精度,需要保守地估计控制流所有可能执行顺序. 因此,如何提供控制流次序信息按需查询是具有流敏感精度的按需别名分析的难点,这也是本文主要研究内容之一.

2) 流敏感信息的表达

文献[5]提出了通过将静态单赋值语句(static single assignment, SSA)与流不敏感指针分析相结合来提高指针分析精度的方法,该方法首先用流不敏感指针分析结果来标记指针变量的指向集合,据此作为初始依据的指向集合来替代指针解引用操作,进而构建含有 SSA 的程序中间表示,然后再利用流不敏感指针分析方法来标记该中间表示中指针变量的指向集合,如果该指向集合与初始依据的指向集合不同,则以此新指向集合为下一轮执行的初始依据,迭代执行以上步骤,直至指向集合稳定为止. 但该文献没有给出实验结果. 文献[6-7]分别从利用 SSA 传递指向信息策略和指向集合存储与操作等方面给出了实际可行的算法,并给出实验结果. 相较于传统基于迭代数据流的流敏感指针分析方法,该类方法在保证流敏感精度的前提下,在分析效率方面有明显的提升. 但是,需要遍历所有指针变量的相关语句,仍然不具备理想的按需分析能力.

在程序分析中,SSA 是将控制流次序信息蕴含在数据流中的一种表达方式,其主要思想是每个变量只能被定义赋值一次[8]. 若变量在程序中被多次赋值时,该变量将会被拆分不同变量实例,每 1 次赋值对应一个实例. 在控制流图的合并结点处,同一变量的不同实例通过函数 ϕ 合并,将合并结果赋值给该变量的新实例. SSA 形式由于具备显式的定义-使用链信息,可以将数据流信息由变量实例的定义处传递到使用处,因此可以有效地过滤掉与分析目标无关的变量,提高分析效率,实现稀疏化分析的目的,是实现按需分析策略的基础.

但是,当存在通过指针间接引用赋值时,需要准确的指针指向信息来展开间接引用,而当前构建 SSA 的目的也是为了分析指针指向集合,这样导致了前提与目标互相依赖. 文献[5-7]采用迭代策略打破循环依赖,然而,迭代策略需要计算并传播所有指针变量的指向信息,但并不是所有指针变量都与目标相关,这与按需分析策略相悖. 因此,为了避免处理指针间接引用的复杂情况,本文借鉴文献[6]中不完全 SSA 形式来表达指针变量,即只将没有被取址符操作过的变量变换为 SSA 形式,而保留其他指针变量,从而提高与分析目标指针相关语句的稀疏程度,缩小后续搜索的范围. 同时,与文献[5-7]中按语句顺序迭代传播所有指针的指向信息不同,本文只搜索能满足流敏感约束的赋值流并且与目标指针构成别名关系的指针信息,从而避免迭代传递与分析目标无关的其他指针变量的指向信息. 由于本文的搜索过程是采用以目标指针变量为入口、逆向追溯定义-使用链的方式来查询别名关系,所以无法采用文献[5-7]中以入口为开始、顺序遍历语句的方式来获得流敏感信息. 因此,本文着重研究的另一项内容是如何按需搜索具有流敏感精度的指针变量定义-使用链.

2 流敏感信息的按需查询

为了提供控制流次序信息按需查询的能力,本文将控制流位置的次序查询问题转换为有向图可达性判断问题. 控制流图中的区域是单入口单出口的子图,是与控制语句对应的结构[8]. 本文根据控制流图区域间的层次化特点,采用 LLC 方法来标记控制流图结点,进而以常数时间复杂度来实现结点间可达性的按需查询. LLC 编码的形式化定义如下所示:

$$\begin{aligned} LLCode &:= LLCode_{\text{parent}} \times LevelCode,\\ LevelCode &:= BranchNum \times SeqenceNum,\\ LLCode_{\text{parent}} &:= \varnothing \mid LLCode, \end{aligned} \tag{1}$$

其中,$BranchNum$ 表示结点所属区域的所在分支的序号,顺序区域默认分支序号为 1;$SequenceNum$ 表示结点在所属区域内的顺序序号. 为了方便后续讨

论,任意结点 M 的 LLC 编码可以总结如下:

$$LLCode_M := (LLCode_{\text{parent}}, LevelCode_M) := (LC_0^M, LC_1^M, \cdots, LC_{m-1}^M, LC_m^M) := (LC_{(0,m-1)}^M, LC_M) = LC_{(0,m)}^M. \quad (2)$$

定义 1. 控制流图结点 M 和结点 N 的层内编码构成从属关系,其中:

$$LLCode_M := (LC_{(0,m-1)}^M, LC_M) := (LC_{(0,m-1)}^M, (BranchNum_M, SequenceNum_M)), \quad (3)$$

$$LLCode_N := (LC_{(0,m-1)}^N, LC_N) := (LC_{(0,m-1)}^N, (BranchNum_N, SequenceNum_N)), \quad (4)$$

记为 $LC_M \leqslant_{\text{lsub}} LC_N$,当且仅当 $SeqNum_M \leqslant SeqNum_N$ 并且 $BranchNum_M = BranchNum_N$.

定义 2. 控制流图结点 M 和结点 N 的层间编码构成从属关系,其中 $LLCode_M = (LC_{(0,i)}^M, LC_M)$, $LLCode_N = (LC_{(0,j)}^N, LC_N)$,记为 $LLCode_M \leqslant_{\text{sub}} LLCode_N$,当且仅当,存在 $0 \leqslant k < i, j$ 使得 $LC_l^M = LC_l^N, l \in [0,k]$, $LC_{k+1}^M \leqslant_{\text{lsub}} LC_{k+1}^N$.

定理 1. 结点 M 和 N,结点 M 可达 N,当且仅当 $LLCode_M \leqslant_{\text{sub}} LLCode_N$ 或者存在回边 OP,其头结点和尾结点编码分别为 $LLCode_P$ 和 $LLCode_O$,使得 $LLCode_M \leqslant_{\text{sub}} LLCode_O, LLCode_P \leqslant_{\text{sub}} LLCode_N$.

证明. 必要性显然.下面简要证明充分性.当 $LLCode_M \leqslant_{\text{sub}} LLCode_N$ 时,构成从属关系,显然 M 可达 N.当存在回边 OP 时,$LLCode_M \leqslant_{\text{sub}} LLCode_O$ 保证了 M 可达 O,$LLCode_P \leqslant_{\text{sub}} LLCode_N$ 保证了 P 可达 O,而回边是从 O 到 P,因此 M 可达 N. 证毕.

3　赋值流图

为了保证查询效率并提供流敏感的查询精度,本文提出了赋值流图来表达具有流敏感信息的指针操作.该图在程序表达式图(program expression graph, PEG)[1]基础上进行了如下改进:1)利用层次线性化编码对有向边进行标记;2)参照不完全 SSA 形式,对未被取址的指针变量的不同赋值实例进行区分,分别创建对应的结点.赋值流图形式化定义为 6 元组,即:

$$G_{\text{AFG}} = (N_{\text{addr}}, N_{\text{var}}, E_{\text{addr}}, E_{\text{flow}}, Map_{\text{var}}(N_{\text{var}}, Var_{\text{SSA}}), Map_{\text{flow}}(E_{\text{flow}}, LLCode)). \quad (5)$$

与 PEG 相似,AFG 图中的结点和边也分别划分成 2 种.N_{var} 包含指针变量结点,N_{addr} 包含与指针变量对应的指向变量结点和地址变量结点;E_{flow} 表示赋值关系,记为 A 边;E_{addr} 表示指向关系,记为 D 边.此外,AFG 图通过增加 2 个映射关系来实现流敏感信息的按需查询.其中,Map_{var} 建立了结点与未被取址指针变量赋值实例的映射关系;Map_{flow} 建立了赋值关系边与流敏感信息的映射关系.

将赋值流图中结点和边进行划分的目的是为了统一表达针运算关系.在 C 编程语言中指针操作符包括取址操作符(&)和解指针操作符(*).为了消除取址操作符,假设每个变量 $n \in N_{\text{var}}$ 都有一个隐含的地址变量 $n_{\text{addr}} \in N_{\text{addr}}$ 与之对应;那么当对变量 n 进行取值操作时,$m = \&n$ 可以表达为 $m = n_{\text{addr}}$;当对变量 n 进行赋值时,$n = 3$ 可以表达为 $(*n_{\text{addr}}) = 3$.这样地址变量和解指针操作符就可以替换取址操作符.另外,为进一步缩短指针赋值链长度,参照不完全 SSA 表达形式,将未被取址操作的指针变量替换为其对应的赋值实例.

根据如下规则构建赋值流图:

$$\begin{aligned} &\&m :- m_{\text{addr}} \in N_{\text{addr}}, (m_{\text{addr}}, m) \in E_{\text{addr}};\\ &*n :- n_{\text{pointto}} \in N_{\text{addr}}, n \in N_{\text{var}}, (n, n_{\text{pointto}}) \in E_{\text{addr}};\\ &m = n :- m_{i+1} \in N_{\text{var}}, (m,n) \in E_{\text{flow}}, (m_{i+1}, m) \in Map_{\text{var}};\\ &p = *q(LLCode_j) :- (p, q_{\text{pointto}}) \in E_{\text{flow}}, ((p, q_{\text{pointto}}), LLCode_j) \in Map_{\text{flow}}. \end{aligned} \quad (6)$$

根据规则构建的赋值流图样例如图 3 所示.其中,左侧是 C 程序指针赋值语句;右侧是对应的赋值流图.虚线边是指向关系,即 D 边,实线边是赋值关系,即 A 边.实线有向边通过标记的层次线性化编码来表达赋值语句的次序信息.指针变量的下标代表了未被取址变量的赋值实例.没有下标的指针变量则表示已经被取址过,例如变量 t,z 和 r.在流敏感精度条件下,在 $LLCode_7$ 处与指针变量 $*s$ 构成别名关系的指针变量只有 x.而在流不敏感精度条件下,还会包含变量 t 和 z.因此,赋值流图可以为分析流敏感的别名关系提供基础.

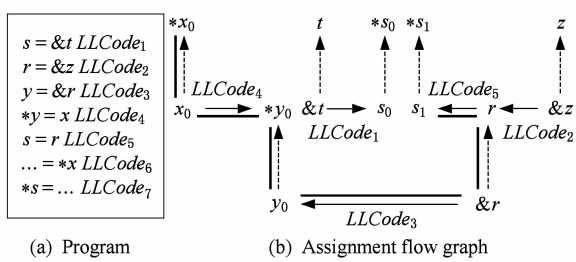


Fig. 3　Example for assignment flow graph.

图 3　赋值流图样例

4 赋值流搜索

为了保证按需别名分析具有流敏感精度,本文将别名判断问题转换为赋值流图上对应结点间满足控制流执行顺序约束的可达性判断问题.该可达性判断依赖于能否搜索到满足如下约束的赋值流:

1) 赋值流的头结点和尾结点构成别名关系.

2) 赋值流中赋值边的执行顺序保证传递赋值有效.

能够保证2个指针表达式是别名关系的赋值流可以由2个等价关系迭代构成:

1) 内存地址相等.指向相同的内存地址,通过数值传递来获取相同内存位置.

2) 数值传递相等.指针变量所含数值相同,在执行顺序的约束下,通过相同地址的等价关系来实现数值传递.

为了能够使用上下无关文法来驱动搜索,将以上2个等价关系以产生式形式进行定义,如式(7)所示,与之对应的自动机如图4所示.其中,终结符 A 与 D 分别表示赋值流图中的赋值关系有向边和指向关系有向边;终结符 $\bar{\mathrm{A}}$ 和 $\bar{\mathrm{D}}$ 分别表示赋值关系的逆向边和取地址边;终结符 $MostRecent(A,LLCode_{\mathrm{cur}})$ 表示相对当前控制流位置最近的起作用的正向传递,逆向等同.非终结符 Mem 表示内存地址相等;非终结符 Val 表示数值传递相等;非终结符 $Flow$ 表示通过流敏感的正向传递;非终结符 $\overline{Flow}$ 表示流敏感的逆向传递.

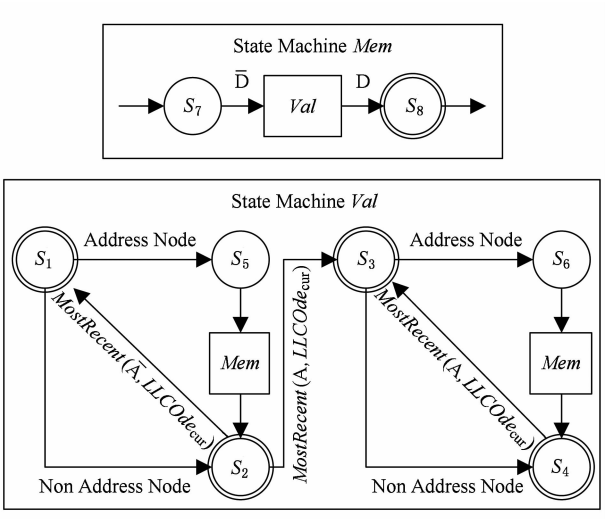


Fig. 4 Hierarchical state machine for searching flow-sensitive assignments.

图4 搜索赋值路径的层次化状态机

$$\begin{aligned}
&Mem := \bar{D}ValD;\\
&Val := \overline{Flow}(Mem?)Flow;\\
&Flow := (MostRecent(\mathrm{A}, LLCode_{\mathrm{cur}})Mem?)^{*};\\
&\overline{Flow} := (Mem?MostRecent(\bar{\mathrm{A}}, LLCode_{\mathrm{cur}}))^{*}.
\end{aligned} \tag{7}$$

图4中,将非终结符 $Flow$ 与 $\overline{Flow}$ 对应的自动机蕴含在 Val 中.为了保证赋值流搜索能够满足流敏感精度,首先通过是否具有地址结点来区分非SSA格式指针变量;当结点是非SSA格式指针变量时,即状态 S_5,通过自动机 Mem 优先计算相对当前位置有效的内存指向等价候选结点集合,最后根据最近有效原则即函数 $MostRecent(\bar{\mathrm{A}}, LLCode_{\mathrm{cur}})$ 来确定有效的数值传递边,到达状态 S_2,重复此过程直到发现正向赋值边进入状态 S_3.而后正向赋值边搜索过程与此前逆向赋值边搜索过程相对称.

根据执行顺序,函数 $MostRecent(\mathrm{A}, LLCode_{\mathrm{cur}})$ 计算相对当前位置有效的赋值有向边,具体如算法1所示.算法所依赖的关系如下所示,

1) $Loop := (NodeBack_{\mathrm{EdgeFrom}}, Node_{\mathrm{BackEdgeTo}})$,循环 $Loop$ 以回边的头尾结点标记.

2) $FromNode(Loop) = Node_{\mathrm{BackEdgeFrom}}$, $ToNode(Loop) = Node_{\mathrm{BackEdgeFrom}}$, $BelongLoop(TarNode) := \{Loop \mid LLCode_{TarNode} \leqslant_{\mathrm{sub}} LLCode_{FromNode(Loop)} \wedge LLCode_{ToNode(Loop)} \leqslant_{\mathrm{sub}} LLCode_{TarNode}\}$,通过目标结点的LLC编码与循环回边头尾结点的LLC编码来过滤目标结点所属循环 $BelongLoop$.

3) $Candidate(TarNode) := \{a \mid a \in Nodes(G_{\mathrm{AFG}}) \wedge (LLCode_a \leqslant_{\mathrm{sub}} LLCode_{TarNode} \vee (LLCode_{TarNode} \leqslant_{\mathrm{sub}} LLCode_a \leqslant_{\mathrm{sub}} LLCode_{FromNode(Loop)} \wedge Loop \subseteq BelongLoop(TarNode)))\}$,在赋值流图中,沿控制流逆向的赋值边候选集合 $Candidate$ 由在当前控制流位置前驱的赋值边和后继并能通过回边作用的赋值边构成.

4) $NextCandidate(TarNode, Base) := \{a \mid a \in Nodes(G_{\mathrm{AFG}}), LLCode_{TarNode} \leqslant_{\mathrm{sub}} LLCode_a \leqslant_{\mathrm{sub}} LLCode_{Base}\}$,沿控制流顺向赋值边候选集合 $NextCandidate$ 由位于出在当前跟踪到控制流位置和查询目标位置之间的赋值边构成.

5) $SplitByLLC(Cndts, TarNode) = \{(A,B) \mid A = \{a \in Cndts \mid LLCode_a \leqslant_{\mathrm{sub}} LLCode_{TarNode}\}, B = \{b \in Cndts \mid LLCode_{TarNode} \leqslant_{\mathrm{sub}} LLCode_b\}\}$,$SplitByLLC$ 的功能是根据目标结点LLC将赋值边候选项划分为顺序前导集合 A 和后继回边作用集合 B.

6) $Killed(Cndts, Base)=\{a \mid a, b \in Cndts, LLCode_a \leqslant_{sub} LLCode_{Base}, LLCode_b \leqslant_{sub} LLCode_{Base}, (LLCode_a \leqslant_{sub} LLCode_b, b \in post\_dom(a)) \vee (LLCode_a \leqslant_{sub} LLCode_i, LLCode_i \in Wrts, DomBranchNum(Wrts, a)=SuccDomNum(a)-1)\}$,被注销的赋值边候选项 $Killed$ 由 2 部分组成,一部分是由处在后向支配($post\_dom$)位置的赋值边注销,另一部分是当所有非支配后继分支都存在赋值边时被集体注销.被集体注销的含义是指即使后向支配结点处没有赋值边,但是所有后继分支都存在新的赋值边,同样注销该赋值边的影响,但是,如果不是每一条分支都存在,则该赋值边同其他后续赋值边保留.其中,$DomBranchNum$ 是其他赋值边在控制流图前向支配树中所占有的分支个数,$SuccDomNum$ 是赋值边在控制流图前向支配树中的出度,减 1 的目的是消去所属的后向支配占用的一条分支,由此实现被注销赋值边所有后续分支都存在新赋值边的约束目的.

7) $Live(Cndts, Base)=Cndts-Killed(Cndts, Base)$,当前有效赋值边 $Live$ 由赋值边候选项与被注销的候选项的差集构成.

8) $Delegate(Cndts, Loop)=\{(a,b) \mid a \in Nodes(G_{AFG}), b \in Cndts, a=ToNode(Loop), c=FromNode(Loop), LLCode_b \leqslant_{sub} LLCode_c\}$, $Next(Cndts)=\{a \mid a \in Cndts, \forall b \in Cnts, LLCode_a \leqslant_{sub} LLCode_b\}$,在循环回边作用下,赋值边候选项新的代理位置 $Delegate$ 由所处循环回边的头结点构成.赋值边候选集合中最先执行子集 $Next$ 根据 LLC 偏序关系的上确界获得.

算法 1. 流敏感有效赋值边搜索算法.

输入:当前节点 $Asgn$,搜索起始节点 $LLCode_{cur}$;

输出:有效的赋值边.

$MostRecent(Dir(Asgn):Direction(\text{LLCode}), LLCode_{cur}:\text{LLCode})$

① if ($backward(Asgn)$)
　/ * 逆向赋值情况 * /
② 　{$cndidts=Candidate(Asgn)$;
③ 　if ($single(cndidts)$ or $empty(cndidts)$)
④ 　　{$result=Live(cndidts, Asgn)$; return $result$;}
⑤ 　{$[A, B]\}=SpliteByTarLLC(cndidts, Asgn)$;
⑥ 　foreach $loop$ in $BelongLoop(Asgn)$
⑦ 　　{$C=Live(B, FromNode(loop))$;
⑧ 　　$D+=Delegate(C, loop)$; }
⑨ 　$result=Live(A+D, LLCode_{cur})$;}
⑩ else { / * 前向赋值情况 * /
⑪ 　$cndidts=NextCandidate(Asgn, LLCode_{cur})$;
⑫ 　$result=Next(B)$;}
⑬ return $result$.

在算法 1 中,首先(行①)根据查询结点搜索方向分别计算有效赋值边.当搜索方向为后向时,沿控制流逆向的下一赋值边候选集合.当该集合为单元素或空时(行②～③)则可直接计算有效赋值边并结束返回(行④).否则,将赋值边候选集合划分为前驱集合 A 和后继集合 B(行⑤).然后,通过遍历当前赋值边所处循环,寻找后继集合有效的赋值边在回边作用下代理位置集合 D(行⑥～⑨),这样将受循环作用下的后继集合 B 与前驱集合 A 的影响方向相统一,便于后续计算.最后,合并前驱集合 A 与代理位置集合 D 计算最终有效的赋值边(行⑩).当搜索方向为前向时,沿控制流顺向搜索下一赋值边候选集合(行⑪～⑫).

5　流敏感按需别名分析算法

在第 3 节和第 4 节分别提出的赋值流图及相应的搜索状态机基础上,本节给出具有流敏感精度的按需别名分析算法,如算法 2 所示.算法 2 是由以取址深度为权重的加权工作列表作为驱动,当给定查询目标结点 e_1 和 e_2、控制流位置 $tarpost$ 后,通过在 AFG 上搜索由位于位置 $tarpost$ 之前执行的语句形成的赋值流来判断目标结点间是否构成别名关系.该搜索过程是在图 4 中层次化状态机指导下的实现.其中,函数 $MostRecent$ 根据临近原则分别计算赋值边 A 或 $\bar{\text{A}}$ 的前驱和后继有效赋值边(由图 5 定义),解引用 D 边和取址 $\bar{\text{D}}$ 边分别由函数 $derref$ 和函数 $adrs$ 来表达.算法 2 主要通过数值传递相等关系 V,通过 $aliasMem$ 来表示由数值相等构成的内存地址相等关系 M.4 元组($n, s, state, level$)构成加权工作列表元素,表示在状态 c 下从结点 n 到结点 s 存在赋值链,状态 c 表示图 4 中自动机 Val 的 4 个终结状态,取址操作符深度 $level$ 是权重.为了保证单次数值传递相等关系的完整性,算法 2 优先搜索

取址层次较深的赋值链,从而保证在搜索上层数值传递相等关系时,由下层赋值链构成的内存地址相等关系已经计算完毕,使得仅通过单次搜索即可涵盖含有间接访问的赋值链.

算法 2. 流敏感按需别名分析算法.

输入: 赋值流图 G_{AFG},待查别名关系节点 e_1 和 e_2,当前位置 $tarpost$;

输出: 是否构成流敏感别名关系.

DDFSMayAlias(e_1: Expr, e_2: Expr, $tarpost$: LLCode)

① $w \leftarrow \{(adrs(e_1), adrs(e_1), S_1, level=1)\}$;
② while (w.*NotEmpty*()) {
③ $(n, s, state, level) = weightedpop(w)$;
④ if ($derref(n)=e_1 \wedge derref(s)=e_2$)
return true;
⑤ if ($derref(n)\neq$null $\wedge$ $derref(n) \notin$ $aliasMem(derref(s))$)
⑥ $aliasMem(derref(s)).append(derref(n))$
⑦ switch($state$):
⑧ case S_1:if($adrs(n)\neq$null) {
/* Transmit to State:S_5 */
⑨ *AddReachable*($adrs(n), adrs(n), S_1, level+1$);
⑩ *AddReachable*($adrs(n), adrs(s), S_2, level$); } break;
⑪ case S_2:foreach p in $aliasMem(n)$
⑫ $ValidFromAssign += MostRecent(backward(p), tarpost)$;
⑬ $ValidToAssign += MostRecent(forward(p), tarpost)$;
⑭ *AddReachable*($ValidFromAssign, s, S_1, level$);
⑮ *AddReachable*($ValidToAssign, s, S_3, level$); break;
⑯ case S_3:if($adrs(n)\neq$null)
/* 转移到状态 S_6 */
⑰ *AddReachable*($adrs(n), adrs(n), S_1, level+1$);
⑱ *AddReachable*($adrs(n), adrs(s), S_4, level$); break;
⑲ case S_4:foreach p in $aliasMem(n)$
⑳ $ValidToAssign += MostRecent(forward(p), tarpost)$;
㉑ *AddReachable*($ValidToAssign, s, S_3, level$); break; }
㉒ return false.

AddReachable($nodeN$:Expr, $nodeS$:Expr, $state$:State, $level$:Level)
㉓ if ($nodeN, nodes, state, level) \notin w$
㉔ $w \leftarrow w \cup \{nodeN, nodeS, state, level\}$.

算法 2 工作流程如下:首先以查询目标地址 $adrs(e_1)$、自动机 Val 状态 S_1 和取址层次为 1 来初始化加权工作列表(行①).然后检查搜索状态空间是否为空(行②).如果为空则返回 false 以示目标变量间无法构成别名关系并终止(行㉒).如果非空则根据取地址层次深度优先原则从加权工作列表弹出待搜索状态(行③).

接着判断当前赋值链首尾结点是否指向目标结点(行④),如果是则返回 ture 以示目标结点间构成别名关系并终止.如果不是则将当前可达结点的指向结点 $derref(d)$ 加入到源结点指向结点 $derref(s)$ 的内存地址相等关系 $aliasMem$ 中(行⑤~⑥).行⑦~⑳根据是在图 4 状态机 Val 中所处的对应状态控制赋值流搜索方向.其中,行⑧~⑩是状态 S_1 下的变迁,将具有地址结点的当前可达结点向自动机 Mem 展开,并将当前结点状态变换为 S_2.行⑪~⑮是状态 S_2 下的变迁,利用函数 $MostRecent$ 分别计算有效的 A 或 $\bar{A}$ 的前驱和后继有效赋值边的前向赋值边和后向赋值边,然后将对应的搜索状态分别设置为 S_1 和 S_3.行⑯~⑱是状态 S_3 下的变迁,与状态 S_1 相似,将具有地址结点的当前可达结点向自动机 Mem 展开,但将当前结点状态变换为 S_4.行⑲~㉑是状态 S_4 下的变迁,由于是由前向赋值边 A 来搜索的,只有继续延续前向搜索才能保证数值传递相等关系,所以与状态 S_2 不同,该状态下只搜索前向赋值边并设置下一状态为 S_3.

6 实验与分析

本文提出的算法已经通过扩展编译器框架 LLVM[9] 中端分析模块的方式来实现.以 PARSEC 程序集[10] 作为测试集,选择的程序基本信息如表 1 所示.其中,"别名查询数量"记录随机产生的目标查询的数量.为了满足稀疏查询特点,查询数量占结点数比例较小.

Table 1　Basic Information of Evaluation Programs

表 1　实验样例程序基础信息表

Program Name	Description	Code Size/line	Assignment Flow Graph		Number of Alias Queries
			Number of Nodes	Number of Edges	
Blackscholes	Financial Analysis	443	215	189	5
Facesim	Animation Software	1600	981	832	24
Fluidanimate	Animation Software	3280	2563	2106	56
Ferret	Search Algorithm	9665	7921	5872	86

6.1　按需策略下指针别名分析方法对比

为了比较本文方法与传统流敏感精度的指针指向分析算法在按需查询别名关系时的效率,以表 1 生成的测试集为输入,分别执行了基于 CFL 的本文方法和基于指向集合的文献[6]方法,测试结果如表 2 所示.其中,表 1 第 1～5 列是测试程序基本信息,第 6 列是随机产生的别名关系查询数量.表 2 第 2～4 列分别是本文方法的预处理时间、平均查询时间和平均内存消耗量,第 5～6 列分别是基于指向集合的流敏感指针别名分析方法的平均查询时间和平均内存消耗量,第 7 列是指本文平均查询时间占指向分析查询时间的比率;第 8 列是指本文方法查询用内存占指向分析查询用内存的比率.实验结果表明,当别名查询稀疏时,在时空效率上,按需分析策略明显优于传统指向分析.但是,随着查询目标复杂度的增加优势会缩小.在最坏情况下,算法会遍历所有赋值流图,与指向分析具有相同的时空复杂度.

Table 2　Flow-Sensitive Pointer Analysis Comparison between Demand Driven Alias Style and Point-To Style

表 2　流敏感精度的按需别名分析与指向分析实验结果表

Program Name	Flow-Sensitive CFL			Point-To		Ratio of Time/%	Ratio of Memory/%
	Average Preprocess Time/ms	Average Answer Time/ms	Average Used Memory/MB	Average Answer Time/ms	Average Used Memory/MB		
Blackscholes	24	0.23	10	84	73	28.8	13.7
Facesim	75	0.27	26	164	103	45.9	25.2
Fluidanimate	93	0.31	35	279	132	33.4	26.5
Ferret	186	0.29	83	323	145	57.7	57.2

6.2　流敏感与流不敏感按需别名分析对比

为了比较本文为提高分析精度而产生的额外开销占原方法开销的比例,以表 1 生成的测试集作为输入,我们测试了流不敏感的文献[1]方法与流敏感的本文方法的时空开销,比较结果如表 3 所示.表 3 第 2～4 列分别是本文方法的预处理时间、平均查询时间和平均内存消耗量,第 5～7 列分别是流不敏感指针别名按需分析方法的平均查询时间和平均内存消耗量,第 8 列表示在指针别名关系查询方面,本文方法的平均查询时间占基于基于流不敏感方法的

Table 3　Demand Driven Alias Analysis Comparison Between Flow-Sensitive Method and Flow-Insensitive Method

表 3　按需别名分析的流敏感精度方法与流不敏感精度方法实验结果表

Program Name	Flow-Sensitive CFL			Point-To			Ratio of Time/%	Ratio of Memory/%
	Average Preprocess Time/ms	Average Answer Time/ms	Average Used Memory /MB	Average Preprocess Time/ms	Average Answer Time/ms	Average Used Memory /MB		
Blackscholes	24	0.23	10	21	0.19	9	121.05	111.11
Facesim	75	0.27	26	65	0.15	24	128.57	108.33
Fluidanimate	93	0.31	35	78	0.21	31	119.23	112.90
Ferret	186	0.29	83	153	0.18	73	120.83	113.70

平均查询时间的比率,第 9 列表示在指针别名关系查询的空间开销方面,本文方法的空间开销占基于流不敏感按需方法的空间开销的比率.实验结果表明,本文方法在以可接受范围内的额外开销有效地提高了按需别名分析的精度.

6.3 分 析

表 2 的结果显示,在保证流敏感精度的条件下判断别名关系时,本文方法的单次平均时空开销优于传统方法,表明本文方法比传统基于指向信息的方法更适合在交互式环境下快速响应用户查询需求.其原因是相较于传统的流敏感指针别名分析算法,本文方法通过直接在赋值流图上搜索可构成别名关系的赋值流,避免了预先计算与分析目标无关的指针的指向信息,进而提高了流敏感的按需别名分析的反应效率.但是,由于本文方法中记录搜索结果的集合 *aliasMem* 是相对于查询位置的,当前算法中没有对所有控制流位置别名关系缓冲的管理,所以当新查询位置不同时搜索过程需要重新发起搜索,导致当查询密度较大时,本文方法累积的时空开销较大.因此,本文方法更适用于稀疏条件下的按需查询.

表 3 的实验结果表明,相较于现有按需别名分析,本文方法具备动态前向或逆向跟踪定义-使用链的能力,能够提供具有流敏感精度的按需别名分析能力;另外,在提高精度的同时,本文方法的整体额外开销控制在原方法开销 129% 以内.其原因是采用了不完全 SSA 稀疏化指针、按需判断一般图可达性等方法.但是,鉴于增加的开销,对只要求满足流不敏感精度同时对效率要求较高的方法不适宜采用本文方法.而对于其他要求流敏感精度,对效率要求不甚苛刻的场景,本文方法较为适用.

7 相关研究

7.1 指针指向分析与指针别名分析

指针指向分析是在程序编译阶段通过程序分析方法估计与指针表达式指向相同内存位置的其他指针表达式集合.指针别名分析是判断 2 个指针表达式是否指向相同的内存位置.指针的指向分析和别名分析都属于指针分析,应用于不同程序分析场景.指针别名分析仍然是不可判定问题[11].现有的别名分析方法属于近似逼近算法,以平衡计算精度和效率间不同的需求.别名分析算法可以根据精度、分析对象和内部表达方式划分成不同的类别.其中,精度划分依据包括:流敏感与否、上下文相关与否、路径敏感与否;分析对象划分包括:数组元素、域敏感、形态分析等;内部表示划分包括别名对、指向集合等.

7.2 流敏感与流不敏感的别名分析

流敏感与否主要根据别名分析过程中是否区别语句执行流的先后顺序.

1) 流不敏感的别名分析

Steensgaard 算法[12] 和 Andersen 算法[13] 是流不敏感别名分析算法中的典型代表.这 2 种算法都是将指针表达式分别赋予不同的类型,在计算赋值表达式时,根据类型规则推导,生成对应的约束,求解满足该约束的类型标注.Steensgaard 算法将赋值表达式左右两边看作双向等价类约束[12],而 Andersen 算法将等式两边对应的集合看作单向偏序包含约束[13].除上述 2 种算法外,文献[14]利用 BDD 表达指向集合实现了基于包含关系的流不敏感的别名分析算法.文献[15]将 Steensgaard 算法[12] 和 Andersen 算法[13] 方法泛化,提出了 k 约束的流不敏感的别名分析算法.文献[16]利用目标机器地址来区分结构体成员,并通过区别结构体类型来改进推理规则,进而提高 Steensgaard 算法的精度.

2) 流敏感的别名分析

流敏感的别名分析算法通常通过数据流分析框架定义.在每个程序点计算该语句消除和生成的别名信息,根据分析对象的不同精度要求,以对应结构记录该数据流信息.Emami 算法利用指针指向集合来存储别名信息[17].Lam 算法利用指向集合抽象处理单一内存位置[18].Chase 算法利用单个图存储别名信息,其中的一个结点可以指向一个或多个内存位置[19].Jones 算法利用多个图来存储别名相关的数据流信息[20].文献[21]提出了基于指向信息传播的上下文敏感和流敏感的指针分析算法.文献[22]提出了与指针标准化相结合的流敏感指针分析算法.文献[6]改造流不敏感算法计算了流敏感的别名分析算法.文献[7]提出了基于逐层分析指向集的流敏感和上下文敏感的指针分析算法.文献[17-20]是基于数据流方程来计算流敏感的指向信息,文献[6-7]是利用 SSA 稀疏性来将流不敏感的指针分析结果提高到流敏感精度,较前者提高了计算效率,本文受此影响,利用不完全 SSA 和一般图可达性来过滤出与分析目标相关的别名关系,避免计算所有指向信息.

7.3 上下文相关的别名分析

上下文相关的别名分析算法主要考虑函数调用

上下文环境信息的传递,例如形参通过内存地址向实参的传递和函数返回值向调用者的传递.传递过程主要通过2种方式实现:维护参数对应关系和函数内联.维护参数对应关系的方法又称为打标记,区分不同的调用上下文环境.如Fähndrich算法[23]、Landi算法[24]和Choi算法[25-26].Fähndrich算法利用类型推理维护函数调用实例中的参数映射关系[23].Landi算法利用判定假定别名对在函数入口合法性的方式计算上下文敏感别名信息[24].Choi算法通过函数入口别名信息、函数内别名信息来判定最终返回的别名信息[25-26].函数内联方法是针对每一个过程在不同的调用环境下内联后作为整体分别进行分析,Anderson算法[13]就是典型代表.

8　结　论

本文提出了一种具有流敏感精度的按需别名分析算法.利用不完全静态单赋值表达式和层次线性化编码来构建含有流敏感信息的赋值流图,将指针别名按需分析问题转换为满足控制流执行顺序约束的赋值流搜索问题.在查询稀疏的情况下,实现了指针别名分析效率与精度的平衡.另外,在分支结构中,由于距离目标结点越近的赋值流可能更快的发现别名关系.因此,引入控制流距离等启发规则可以加速赋值流搜索.但是,如何表达该距离并与当前层次深度权重相整合,仍有待进一步地研究,是本文未来的研究方向.

参　考　文　献

[1] Zheng X, Rugina R. Demand-driven alias analysis for C [C] //Proc of the 35th Annual ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages (POPL 2008). New York: ACM, 2008: 197-208

[2] Han Wei, He Yeping. Static analysis of TOCTTOU vulnerabilities in Unix-style file system [J]. Journal of Computer Research and Development, 2011, 48(8): 1430-1437 (in Chinese)
(韩伟,贺也平. 类Unix文件系统中TOCTTOU缺陷的静态分析方法[J]. 计算机研究与发展, 2011, 48(8): 1430-1437)

[3] Alpuente M, Feliú M A, Joubert C, et al. DATALOG_SOLVE: A datalog-based demand-driven program analyzer [J]. Electronic Notes in Theoretical Computer Science, 2009, 248: 57-66

[4] Chen Congming, Huo Wei, Yu Hongtao, et al. A survey of optimization technology of inclusion-based pointer analysis [J]. Chinese Journal of Computers, 2011, 34(7): 1224-1238 (in Chinese)
(陈聪明,霍玮,于洪涛,等. 基于包含的指针分析优化技术综述[J]. 计算机学报, 2011, 7: 1224-1238)

[5] Rebecca H, Susan H. Using static single assignment form to improve flow-insensitive pointer analysis [C] //Proc of the ACM SIGPLAN 1997 Conf on Programming Language Design and Implementation(PLDI 1997). New York: ACM, 1997: 97-105

[6] Hardekopf B, Lin C. Flow-sensitive pointer analysis for millions of lines of code [C] //Proc of the 9th Annual IEEE/ACM Int Symp on Code Generation and Optimization (CGO 2011). Los Alamitos, CA: IEEE Computer Society, 2011: 289-298

[7] Yu H, Xue J, Huo W, et al. Level by level: Making flow- and context-sensitive pointer analysis scalable for millions of lines of code [C] //Proc of the 8th Annual IEEE/ACM Int Symp on Code Generation and Optimization (CGO 2010). New York: ACM, 2010: 218-229

[8] Lam M S, Sethi R, Ullman J D. Compilers: Principles, Techniques, & Tools [M]. Boston, MA: Addison Wesley Longman, 2007

[9] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation [C] //Proc of the 2nd Anuual IEEE/ACM Int Symp on Code Generation and Optimization (CGO 2004). Los Alamitos, CA: IEEE Computer Society, 2004: 75-86

[10] Bienia C, Kumar S, Singh J P, et al. The PARSEC benchmark suite: Characterization and architectural implications [C] //Proc of the 17th Int Conf on Parallel Architectures and Compilation Techniques. New York: ACM, 2008: 72-81

[11] Ramalingam G. The undecidability of aliasing [J]. ACM Trans on Programming Languages and Systems (TOPLAS), 1994, 16(5): 1467-1471

[12] Steensgaard B. Points-to analysis in almost linear time [C] //Proc of the 23rd Annual ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages (POPL 1996). New York: ACM, 1996: 32-41

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

[14] Naik M, Aiken A. Conditional must not aliasing for static race detection [C] //Proc of the 34th Annual ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages (POPL 2007). New York: ACM, 2007: 327-338

[15] Shapiro M, Horwitz S. Fast and accurate flow-insensitive points-to analysis [C] //Proc of the 24th ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages (POPL 1997). New York: ACM, 1997: 1-14

[16] Yu Hongtao, Zhang Zhaoqing. An aggressive field-sensitive unification-based pointer analysis [J]. Chinese Journal of Computers, 2009, 32(9): 1722-1735 (in Chinese)
(于洪涛, 张兆庆. 激进域敏感基于合并的指针分析[J]. 计算机学报, 2009, 32(9): 1722-1735)

[17] Emami M, Ghiya R, Hendren L J. Context-sensitive interprocedural points-to analysis in the presence of function pointers [C] //Proc of the ACM SIGPLAN 1994 Conf on Programming Language Design and Implementation (PLDI 1994). New York: ACM, 1994: 242-256

[18] Wilson R P, Lam M S. Efficient context-sensitive pointer analysis for C programs [C] //Proc of the ACM SIGPLAN 1995 Conf on Programming Language Design and Implementation (PLDI 1995). New York: ACM, 1995: 1-12

[19] Chase D R, Wegman M, Zadeck F K. Analysis of pointers and structures [C] //Proc of the ACM SIGPLAN 1990 Conf on Programming Language Design and Implementation (PLDI 1990). New York: ACM, 1990: 50-63

[20] Jones N D, Muchnick S S. Flow analysis and optimization of lisp-like structures [C] //Proc of the 6th Annual ACM SIGACT-SIGPLAN Symp on Principles of Programming Languages (POPL 1979). New York: ACM, 1979: 244-256

[21] Huang Bo. Zang Binyu, Wei Junyin, et al. Context-sensitive interprocedural pointer analysis [J]. Chinese Journal of Computers, 2000, 23(5): 477-485 (in Chinese)
(黄波, 臧斌宇, 韦俊银, 等. 上下文敏感的过程间指针分析[J]. 计算机学报, 2000, 23(5): 477-485)

[22] Wang Tiantian, Su Xiaohong, Ma Peijun. Research on pointer analysis algorithm for program standardization [J]. Acta Electronica Sinica, 2009, 37(5): 1104-1108 (in Chinese)
(王甜甜, 苏小红, 马培军. 程序标准化转换中的指针分析算法研究[J]. 电子学报, 2009, 37(5): 1104-1108)

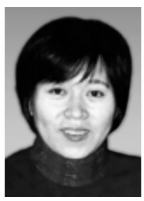
[23] Foster J S, Fähndrich M, Aiken A. Polymorphic versus monomorphic flow-insensitive points-to analysis for C [C] //Proc of the 7th Int Symp on Static Analysis (SAS 2000). Berlin: Springer, 2000: 175-198

[24] Landi W, Ryder B G. A safe approximate algorithm for interprocedural pointer aliasing [C] //Proc of the ACM SIGPLAN 1992 Conf on Programming Language Design and Implementation (PLDI 1992). New York: ACM, 1992: 235-248

[25] Choi J D, Burke M, Carini P. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects [C] //Proc of the 20th ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages (POPL 1993). New York: ACM, 1993: 232-245

[26] Hind M, Burke M, Carini P, et al. Interprocedural pointer alias analysis [J]. ACM Trans on Programming Languages and Systems (TOPLAS), 1999, 21(4): 848-894

Pang Long, born in 1984. PhD candidate. His research interests include program analysis.



Su Xiaohong, born in 1966. Professor of Harbin Institute of Technology. Member of China Computer Federation. Her research interests include software engineering and program analysis(sxh@hit.edu.cn).

Ma Peijun, born in 1963. Professor of Harbin Institute of Technology, member of China Computer Federation. His research interests include software engineering and information fusion(ma@hit.edu.cn).

Zhao Lingling, born in 1980. PhD, lecturer of Harbin Institute of Technology. Her research interests include information fusion, nonlinear filter and program analysis(zhaoll@hit.edu.cn).