第十二章 指针分析

冯 洋

简介

- 核心问题：指针可以指向哪些内存空间？
- 别名分析（Alias Analysis）：哪些指针表达式指向同一块内存空间
- 示例：

```
int x;
p = &x;
q = p;
```

*p 与 *q，x 与 *p，x 与 *q 是别名

简介

- 核心问题可能会因为如下情况产生
  - 指针的使用
    - int *p, i; p = &i;
  - 引用传递
    - void m(Object a, Object b) { … }
    - m(x,x); // a and b alias in body of m
    - m(x,y); // y and b alias in body of m
  - 数组索引
    - int i,j,a[100];
    - i = j; // a[i] and a[j] alias

指针分析讨论的问题

- 指针分析告诉我们代码在修改或使用哪些内存空间
- 有非常多的应用场景

```
*p = a + b;
y = a + b;
```

如果 *p 与 a 或者 b 互为别名，则 a+b 不是公共子表达式；

指针分析讨论的问题

- 指针分析告诉我们代码在修改或使用哪些内存空间
- 有非常多的应用场景

x = 3; *p = 4; y = x;

指针分析讨论的问题

- 指针分析告诉我们代码在修改或使用哪些内存空间
- 有非常多的应用场景

```
x = 3; *p = 4; y = x;
```

如果 x 与 *p 不是互为别名，则可以通过常量传播优化以上代码

指针分析讨论的问题

- 指针分析告诉我们代码在修改或使用哪些内存空间
- 有非常多的应用场景

x = 3; *p = 4; y = x;

如果 x 与 *p 不是互为别名，则可以通过常量传播优化以上代码；

如果 x 与 *p 一定是互为别名，则可以通过常量传播优化以上代码；

为什么指针分析很困难

- 对C语言来说特别困难，因为C 程序可以对指针进行任何计算
- Java 中指针称为引用，不支持算术运算，并且只能指向一个对象的开头

为什么指针分析很困难

- 对C语言来说特别困难，因为C 程序可以对指针进行任何计算
- Java 中指针称为引用，不支持算术运算，并且只能指向一个对象的开头
- 指针分析必须是过程间分析，没有过程间分析，我们必须假设任何方法调用都可能改变所有可被它访问的指针变量所指向的内容，造成所有过程内的指针别名分析非常低效

为什么指针分析很困难

- 对C语言来说特别困难，因为C 程序可以对指针进行任何计算
- Java 中指针称为引用，不支持算术运算，并且只能指向一个对象的开头
- 指针分析必须是过程间分析，没有过程间分析，我们必须假设任何方法调用都可能改变所有可被它访问的指针变量所指向的内容，造成所有过程内的指针别名分析非常低效
- 支持间接函数调用的语言对指针别名分析提出了一个新的挑战

为什么指针分析很困难

- 支持间接函数调用的语言对指针别名分析提出了一个新的挑战 -- 函数指针

指针分析讨论的问题

- 其他应用场景，包括，但不限于：
  - 过程内 / 过程间（Intraprocedural / Interprocedural）
  - 流敏感 / 流不敏感 （Flow-sensitive / Flow-insensitive）
  - 上下文敏感 / 上下文不敏感 （Context-sensitive / Context-insensitive）
  - 确定性
    - May versus must
  - 栈建模
  - 表现形式

指针分析讨论的问题

- 流敏感（Flow-Sensitive）与流不明感（ Flow-Insensitive ）指针分析
  - 流敏感指针分析：用于分析指针表达式可能指向的内存空间
    - 考虑程序语句执行的顺序
    - 用于分析指针表达式可能指向的内存空间
    - 流敏感分析方法的精度明显好于流不敏感的分析方法
  - 流不敏感指针分析：用于分析指针表达式在程序运行的任意时间点，可能指向的内存空间
    - 不考虑控制流
    - 流非敏感分析（flow-insensitive analysis）：如果把程序中语句随意交换位置（即：改变控制流）， 如果分析结果始终不变，则该分析为流非敏感分析

指针分析讨论的问题

- 流敏感指针分析 VS. 流不敏感指针分析
  - “变量x和y可能会指向同一位置”（流不明感分析）
  - “在执行第20条指令后，变量x和y可能会指向同一位置”（流敏感分析）
  - 一个流不敏感指针别名分析不考虑控制流，并认为所发现的别名在程序所有位置均成立
  - 流敏感指针分析代价通常较高，无法应用于大型复杂程序
  - 流不敏感分析通常可以被应用于大程序分析
  - 分析一下？

指针分析讨论的问题

- 流敏感指针分析 VS. 流不敏感指针分析
  - 流敏感指针分析代价通常较高，无法应用于大型复杂程序
  - 流不敏感分析通常可以被应用于大程序分析
  - 具体分析一下？
    - 活跃变量分析：语句数为n，程序中变量个数为m， 使用bitvector表示集合
    - 流不敏感的活跃变量：每条语句的操作时间为O(m)， 因此时间复杂度上界为O(nm)，空间复杂度上界为 O(m)
    - 流敏感的活跃变量分析：格的高度为O(m)，转移函 数、交汇运算和比较运算都是O(m)，时间复杂度上 界为$O(nm^2)$，空间复杂度上界为O(nm)

指针分析讨论的问题

Alias Mechanism	Intraprocedural May Alias	Intraprocedural Must Alias	Interprocedural May Alias	Interprocedural Must Alias
Reference Formals, No Pointers, No Structures	–	–	Polynomial[1, 5]	Polynomial[1, 5]
Single level pointers, No Reference Formals, No Structures	Polynomial	Polynomial	Polynomial	Polynomial
Single level pointers, Reference Formals, No Pointer Reference Formals, No Structures	–	–	Polynomial	Polynomial
Multiple level pointers, No Reference Formals, No Structures	$\mathcal{NP}$-hard	Complement is $\mathcal{NP}$-hard	$\mathcal{NP}$-hard	Complement is $\mathcal{NP}$-hard
Single level pointers, Pointer Reference Formals, No Structures	–	–	$\mathcal{NP}$-hard	Complement is $\mathcal{NP}$-hard
Single level pointers, Structures, No Reference Formals	$\mathcal{NP}$-hard[14]	Complement is $\mathcal{NP}$-hard	$\mathcal{NP}$-hard[14]	Complement is $\mathcal{NP}$-hard

Table 1: Alias problem decomposition and classification

Landi, William, and Barbara G. Ryder. "Pointer-induced aliasing: A problem classification." In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 93-103. 1991.

上下文敏感分析—定义

- May Analysis：在程序运行时可能（may）互为别名
- Must Analysis：在程序运行时一定（must）互为别名
- 通常情况下，两者均是非常有用的
  - 考虑如下例子：x = 3; *p = 4; y = x;

上下文敏感分析—表现形式

- 一些可能的表现形式
  - 指针指向分析（Points-to Analysis）
  - Points-to pairs：第一个元素指向第二个元素
    - 例如 (p → b), (q → b)
  - 指向同一内存地址的对（pairs）
    - 例如 (*p,b), (*q,b), (*p,*q), (**r, b)
  - 等价集（Equivalence Sets）集合类的所有指针均指向同一内存
    - 例如 {*p,*q,b}

内存地址建模

- 分析的第一个任务：描述内存地址
- 如何建模
  - 全局变量（Global Variables） -- 使用简单的节点进行表示（思考一下，我们前面的介绍？）
  - 局部变量（Local Variables）-- 在每个上下文中添加节点进行表示（如果是上下文不明感分析呢？）
  - 动态内存分配（Dynamically allocated memory）
    - 较为困难，需要通过**有限抽象**进行建模

Andersen风格的指针分析

- 在该方法中，将指针赋值看做一种子集约束 pts(p) 来表示（subset constraints）
- 通过约束来分析指向信息

Constraint type	Assignment	Constraint	Meaning
Base	a = &b	$a \supseteq \{b\}$	$loc(b) \in pts(a)$
Simple	a = b	$a \supseteq b$	$pts(a) \supseteq pts(b)$
Complex	a = *b	$a \supseteq *b$	$\forall v \in pts(b).\ pts(a) \supseteq pts(v)$
Complex	*a = b	$*a \supseteq b$	$\forall v \in pts(a).\ pts(v) \supseteq pts(b)$

Andersen风格的指针分析

- 可以通过集合运算分析，来解决这些约束；我们通过在pts(p)上的分析，完成对应的约束分析；

p = &a;	$p \supseteq \{a\}$
q = p;	$q \supseteq p$
p = &b;	$p \supseteq \{b\}$
r = p;	$r \supseteq p$

pts(p) = {∅, a, b}
pts(q) = {∅, a, b}
pts(r) = {∅, a, b}
pts(a) = ∅
pts(b) = ∅

Andersen风格的指针分析

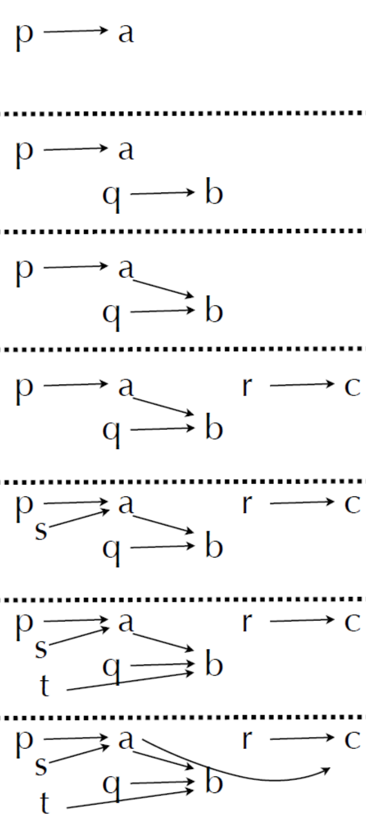
- 另外一个例子

p = &a;	$p \supseteq \{a\}$;	pts(p) = {a}
q = &b;	$q \supseteq \{b\}$;	pts(q) = {b}
*p = q;	$*p \supseteq q$;	pts(r) = {c}
r = &c;	$r \supseteq \{c\}$;	pts(s) = {a}
s = p;	$s \supseteq p$;	pts(t) = {b, c}
t = *p;	$t \supseteq *p$;	pts(a) = {b, c}
*s = r;	$*s \supseteq r$;	pts(b) = ∅
		pts(c) = ∅

Andersen风格的指针分析

- 可以通过集合运算分析，来解决这些约束

```
p = &a
q = &b
*p = q;
r = &c;
s = p;
t = *p;
*s = r;
```



pts(p) = {a}

pts(q) = {b}

pts(r) = {c}

pts(s) = {a}

pts(t) = {b,c}

pts(a) = {b,c}

pts(b) = ∅

pts(c) = ∅

Andersen风格的指针分析

- 转化为一个图闭包问题（graph closure problem）
- 对每一个pts(p)，pts(a) 添加节点
- 每个节点有一个关联的points-to集合
- 计算图的传递闭包，并通过复杂约束添加边

Assgmt.	Constraint	Meaning	Edge
a = &b	$a \supseteq \{b\}$	$b \in pts(a)$	no edge
a = b	$a \supseteq b$	$pts(a) \supseteq pts(b)$	$b \rightarrow a$
a = *b	$a \supseteq *b$	$\forall v \in pts(b).\ pts(a) \supseteq pts(v)$	no edge
*a = b	$*a \supseteq b$	$\forall v \in pts(a).\ pts(v) \supseteq pts(b)$	no edge

工作队列算法

- 通过简单约束初始化图与points to 集合
- Let W = { v | pts(v) ≠∅ } (all nodes with non-empty points to sets)

```
While W not empty
        v ← select from W
        for each a ∈ pts(v) do
                for each constraint p ⊇*v
                        add edge a→ p, and add a to W if edge is new
                for each constraint *v ⊇ q
                        add edge q→a, and add q to W if edge is new
        for each edge v→q do
                pts(q) = pts(q) ∪ pts(v), and add q to W if pts(q) changed
```

一个简单的例子

```
p  =  &a
q  =  &b
*p  =  q;
r  =  &c;
s  =  p;
t  =  *p;
*s  =  r;
```

$p \supseteq \{a\}$

$q \supseteq \{b\}$

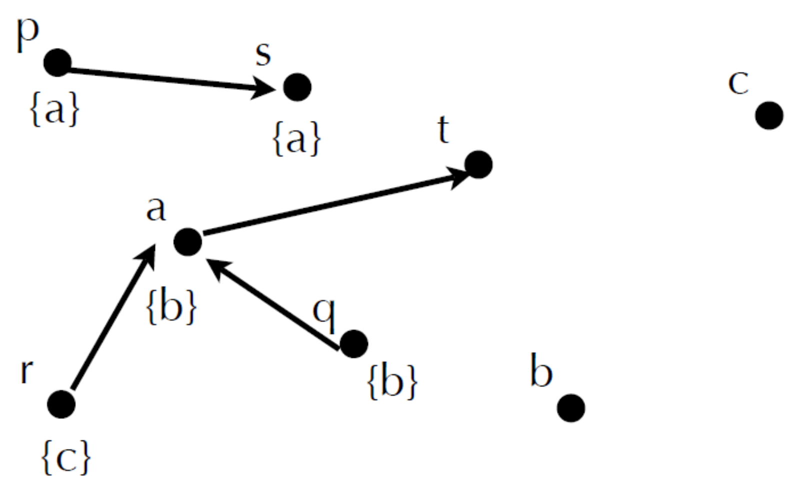
$*p \supseteq q$

$r \supseteq \{c\}$

$s \supseteq p$

$t \supseteq *p$

$*s \supseteq r$



W: p q r s a

While W not empty
- v ← select from W
- for each a ∈ pts(v) do
  - for each constraint p ⊇*v
    - add edge a→ p, and add a to W if edge is new
  - for each constraint *v ⊇ q
    - add edge q→a, and add q to W if edge is new
- for each edge v→q do
  - pts(q) = pts(q) ∪ pts(v), and add q to W if pts(q) changed

一个简单的例子

```
p = &a
q = &b
*p = q;
r = &c;
s = p;
t = *p;
*s = r;
```

$p \supseteq \{a\}$

$q \supseteq \{b\}$

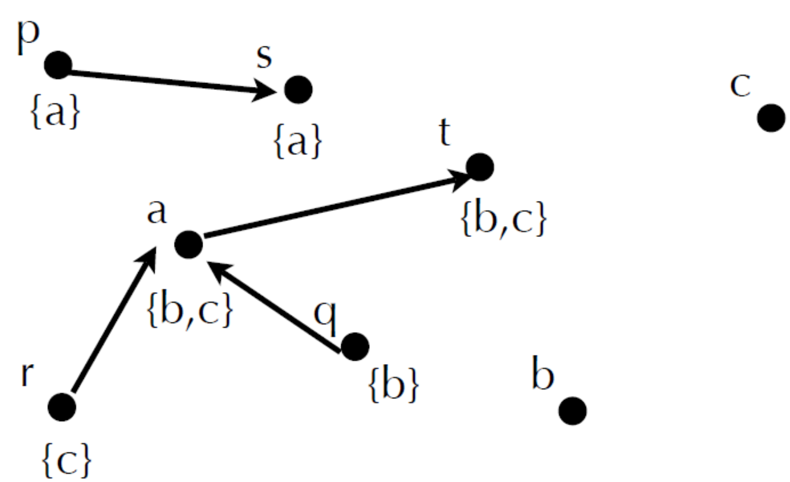
$*p \supseteq q$

$r \supseteq \{c\}$

$s \supseteq p$

$t \supseteq *p$

$*s \supseteq r$



While W not empty
- v ← select from W
- for each a ∈ pts(v) do
  - for each constraint p ⊇*v
    - add edge a→ p, and add a to W if edge is new
  - for each constraint *v ⊇ q
    - add edge q→a, and add q to W if edge is new
- for each edge v→q do
  - pts(q) = pts(q) ∪ pts(v), and add q to W if pts(q) changed

Andersen风格指针分析的讨论

- 理论时间复杂度$O(n^3)$
- 实际时间复杂度$O(n^2)$ [Sridharan and Fink, SAS 09]
- 主要优化方法：缩减n

Andersen风格指针分析的讨论

- 理论时间复杂度$O(n^3)$，但是实际时间复杂度$O(n^2)$ [Sridharan and Fink, SAS 09]
- 主要优化方法：缩减n
- 环的问题（如何消灭环？）

Andersen风格指针分析的讨论

- 理论时间复杂度$O(n^3)$，但是实际时间复杂度$O(n^2)$ [Sridharan and Fink, SAS 09]
- 主要优化方法：缩减n
- 环的问题（如何消灭环？）
  - Andersen风格指针分析中，最重要的一种优化方式
  - 通过图遍历算法检测points-to 图中的**强连通子图**，然后将相关节点坍缩成一个节点
    - 为什么？按照Andersen分析法的算法，所有的强连通子图中的节点最后会获得同样的points-to的关系

Andersen风格指针分析的讨论

- 环的问题（如何消灭环？）
  - Andersen风格指针分析中，最重要的一种优化方式
  - 通过图遍历算法检测points-to 图中的**强连通子图**，然后将相关节点坍缩成一个节点
    - 为什么？按照Andersen分析法的算法，所有的强连通子图中的节点最后会获得同样的points-to的关系
  - 如何高效地检测环？
    - 静态和动态的方法相结合：一些可以通过静态方法检测然后缩减；一些着需要在动态分析过程中进行检测；

Steensgaard风格的指针分析

- 也是一种基于约束的分析技术
- 基于**等价约束（equality constraints）**
- 通过实证研究，发现没有Andersen风格精确，但具有更好的可扩展性

Constraint type	**Assignment**	**Constraint**	**Meaning**
Base	a = &b	$a \supseteq \{b\}$	$loc(b) \in pts(a)$
Simple	a = b	a = b	$pts(a) = pts(b)$
Complex	a = *b	a = *b	$\forall v \in pts(b).\ pts(a) = pts(v)$
Complex	*a = b	*a = b	$\forall v \in pts(a).\ pts(v) = pts(b)$

Steensgaard风格的指针分析

- 可以通过Union-Find 算法快速实现
- 近似于线性复杂度$O(n\alpha(n))$
- 每条语句只需要被处理一次

Constraint type	Assignment	Constraint	Meaning
Base	a = &b	$a \supseteq \{b\}$	$loc(b) \in pts(a)$
Simple	a = b	a = b	$pts(a) = pts(b)$
Complex	a = *b	a = *b	$\forall v \in pts(b).\ pts(a) = pts(v)$
Complex	*a = b	*a = b	$\forall v \in pts(a).\ pts(v) = pts(b)$

一种结合的算法 – One-level Flow

- Das, Manuvir. "Unification-based pointer analysis with directional assignments." *Acm Sigplan Notices* 35, no. 5 (2000): 35-46.
- 主要灵感：最常见的C语言指针使用是为了传递复合对象或者可更新参数的地址 – 因此多级的指针使用是非常常见的
- 使用unification （很像 Steensgaard）但是避免了 top-level指针的unification
- Top-level指针：没有被其他指针指着的指针

一种结合的算法 – One-level Flow

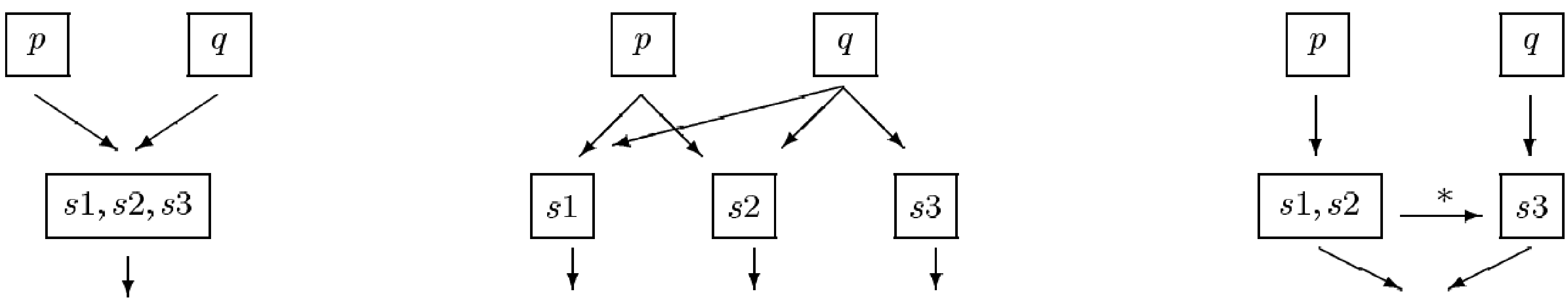
```
foo(&s1);
foo(&s2);
Bar(&s3);

foo(structs s *p){*p.a = 3; bar(p);}
bar(structs s *q){*q.b = 4;}
```

(a)

```
p = &s1;
p = &s2;
q = &s3;
q = p;
*p.3 = 3;
*q.b = 4;
```

(b)

左边的图(a)表达了C语言中的常见指针使用方法；(b)中忽略了其中的函数调用部分；下图c，d，e分别表示了Andersen，Steensgaard，以及one-level flow的算法；



(c) (d) (e)

Java中的指针分析

- 不同的语言需要不用的指针分析方法；
  - 许多C程序使用 & (address-of) 操作符而不是动态分配操作符；
    - & 创建 stack-directed 指针；malloc创建heap-directed指针
  - Java不允许stack-directed 指针
  - Java是一个强类型，指针只能指向一小部分类型
  - Java中依赖于调用图的构建以指针分析
  - Java中的解引用操作主要是通过自动回收机制