

一种面向动态链状数据结构的指针定值引用链算法

付小鹏^{1, 2}, 张 昱^{1, 2}, 张伟^{1, 2}, 汪晨^{1, 2}

¹ (中国科学技术大学 计算机科学与技术学院, 安徽 合肥 230027)

² (中国科学技术大学苏州研究院软件安全实验室, 江苏 苏州 215123)

E-mail : yuzhang@ustc.edu.cn

摘要: 本文采用流敏感的方法分析计算过程内操作动态链状数据结构的指针定值引用链。目的是连接对链状数据结构进行定值的语句和引用这些链状数据结构的语句, 具体地, 每条对链状数据结构进行定值的语句, 算法将找出所有引用被该语句定值的链状数据结构的语句的集合。该算法将被整合到我们设计和开发的并行语言 mini-SPC 中, 指导对操作动态链状数据结构的并行程序的正确分析和程序变换。实验表明基于本文的算法能保证操作动态链状数据结构的指针定值引用链的分析精度, 提高分析的效率。

关键词: 定值引用链; 到达定值; 别名分析; 程序分析; 动态链状数据结构

中图分类号: TP

文献标识码: A

文章编号: 1000-1220 (2010) 02--

Study on Definition-use Chains Algorithm in Dynamic Pointer-linked Data Structures

FU Xiao-peng^{1,2}, ZHANG Yu^{1,2}, ZHANG Wei^{1,2}, WANG Chen^{1,2}

¹ (School of Computer Science & Technology, University of Science & Technology of China, Hefei 230027)

² (Software Security Lab., Suzhou Institute for Advanced Study, University of Science & Technology of China, Suzhou, 215123)

Abstract: This paper presents a flow-sensitive algorithm to compute intra-procedural definition-use chains in dynamic pointer-linked data structures. The aim is to relate the statements that define the links of linked data structures(i.e. definitions) to statements that might refer to the structures though the links(i.e. uses). Specifically, for each statement s that defines links of linked data structures, the algorithm finds the set of statements that use the links which are defined by s . Our method will be incorporated into the implementation of the parallel language mini-SPC designed by us. It is used to be in charge of accurately analyzing and transforming parallel programs with dynamic pointer-linked data structures. Experimental results illustrate that the proposed algorithm for computing definition-use chains could maintain the quality of definition-use chains for dynamically allocated location and improve the analysis performance efficiently.

Key words: definition-use chains ; reaching-definition ; alias analysis ; program analysis ; dynamic linked data structures

1 引言

一个变量的定值引用链是连接该变量的定值语句到所有可能流经到的对该定值的引用语句。定值引用链被广泛地应用于优化和并行编译器[1], 甚至于软件工程工具中[2]。确定含有动态链状数据结构(如链表, 二叉链表示的树等)的指针程序中的定值引用链, 要比没有动态内存分配的程序复杂得多。这主要因为前者允许通过指针间接访问内存单元以及动态构造链状数据结构, 导致程序中存在复杂的别名问题。例如, 假设 q 指向单链表的节点, $p=q$ 导致 $p \rightarrow next$ 和 $q \rightarrow next$ 因代表同一个内存单元而互为左值别名, p 和 q 因指向同一个内存单元而互为右值别名。由于别名的存在, 导致在计算面向动态链状数据结构的指针定值引用链时, 往往

需要在数据流分析的过程中借助别名分析得到的别名集合(alias sets)或内存形状图 SSGs(store shape graphs)。在过去的 20 年里, 别名分析问题已经成为程序分析技术研究的重点和难点之一。现有的别名分析研究成果主要是在精度和时空效率之间做取舍。精确的别名分析, 可以移除不再有效的到达定值, 提高定值引用链的精度, 这样这些定值就不会被错误地传播到任何随后的引用。

本文设计一种新的面向动态链状数据结构的指针定值引用算法, 别名信息的获取是通过正向的数据流分析途径, 沿程序执行的所有路径, 在每个程序点收集所有能到达这个程序点的定值来识别的, 本质上是一个流敏感的别名分析。所谓流敏感就是分析中考虑程序中语句的执行顺序。它的主要特点是: 在某个程序点, 它并不计算出所有的可能别名集

合,而仅仅是识别那些需要知道别名信息的指针变量的可能别名;并且到达定值分析和别名信息的获取是在一个分析遍中完成,而不是先独立的进行别名分析,然后利用别名分析结果再进行到达定值分析。

这种方法的优点是:1、避免建立别名集合或内存形状图来概括所有可能的指针型域访问表达式的别名。2、避免利用到达定值和别名分析的别名信息而产生虚假的定值引用链。3、不会产生多级的访问路径表达式,避免多级域访问路径表达式的相等比较。

本文利用正向分析的数据流迭代算法,计算过程内的指针到达定值信息;在到达定值信息基础上,推导得到程序的引用定值链和定值引用链。该方法将被整合到我们设计和开发的并行语言 mini-SPC[3]中,指导对操作动态链状数据结构的并行程序的正确分析和程序转换。

2 问题描述

2.1 定值引用描述

在没有脱引用和取地址操作的程序中,一般指针变量的定值引用链算法和传统的方法[9]相同,本文的算法主要关注指针型域访问表达式的定值引用链。在含有动态链状数据结构的指针程序中,定值引用链可以扩展为:

针对每条对链状数据结构的域访问表达式进行定值的语句 s ,沿所有可能的程序执行路径,计算引用被语句 s 定值的内存位置的所有语句的集合。

具体地,如图 1(a)所示的程序,假设 s_i 是对 $p \rightarrow f$ 的定值, s_j 中包含对 $q \rightarrow f$ 的引用, s_i 是对 $q \rightarrow f$ 的定值,当且仅当:

1、至少有一条程序执行路径从 s_i 到 s_j , 这样 $p \rightarrow f$ 的定值能到达 s_j

2、 p 和 q 指向同一个内存单元



图 1 定值引用关系

Fig.1 Def-use association

如图 1(b)所示的程序,由于 p 和 q 右值别名, $p \rightarrow n$ 和 $q \rightarrow n$ 代表相同的内存单元,这样语句 s_2 和 s_3 都是对同一内存单元的赋值,这样可以知道 s_4 中对 $p \rightarrow n$ 的引用的定值在 s_3 , 而不在 s_2 。因此在计算定值引用链的时候,需要知道指针变量的右值别名信息,以便确定两个域访问表达式是否代表相同的位置,即是否为左值别名。

2.2 语言

为了形式化算法,我们使用了一个简单的语言,如图 2 所示。语言是强调指针类型的类 C 小语言,所有的数据类型 $type$ 都是结构体类型,所有声明的变量都是结构体指针变量。语言中有动态的内存分配 $malloc$ 和释放 $free$ 操作,但是禁止取地址&操作、强制类型转换和指针算术运算。程序中可能的值为指向内存位置的指针或空指针。假设 $malloc$ 的每次调用都能成功分配并且所分配空间和尚未释放空间不相交,而 $free$ 操作每次释放指针指向的内存单元后将该指

针置为空。

| | |
|----------|--|
| (Stmts): | $s \in S$ $s ::= [e_0 := e_1]^t s_0 ; s_1$ $ [e := malloc(type)]^t [free(e)]^t$ $ if [b]^t then s_0 else s_1 while [b]^t do s$ |
| (Exp) | $e \in E$ $e ::= null x e \rightarrow f$ |
| (BExp) | $b \in B$ $b ::= true false$ $ e e != e e == e \dots$ |
| (Var) | $x \in V$ |
| (Field) | $f \in F = \{f_1, \dots, f_n\}$ |
| (Lab) | $l \in L$ |

图 2 语言的语法

Fig. 2 Syntax of the language

在动态分配产生的结构体中,每个域保存一个内存位置。多级的域访问表达式,如 $x \rightarrow f \rightarrow f$,通过引入临时变量而被规范化(normalize)为一级的域访问表达式,如 $t = x \rightarrow f$, $t \rightarrow f$ 。虽然等式 $e_1 \rightarrow f = e_2 \rightarrow f$ 是有效的,但为了简化,表示为 $t = e_2 \rightarrow f$ 和 $e_1 \rightarrow f = t$,即只允许等式的一边出现域访问表达式。从而,在分析程序时只需考虑如下三种指针赋值语句:

1. $p = q$ (别名语句)
2. $p = q \rightarrow f$ (引用语句)
3. $p \rightarrow f = q$ (定值语句)

前两种语句将引起右值别名,而不会改变链状数据结构中节点之间的连接;而最后一种语句会导致从链状数据结构中移除一个连接(或 null),并引入一个新的连接,即指针型域的新的定值。

为了抽象具体的内存位置,用 H 来表示在程序执行的过程中,变量对应的具体内存,包含变量的栈位置、以及程序执行中产生的所有堆位置。具体的内存抽象是一个二元组 $\rho = (\rho_v, \rho_f)$:

$$\rho_v : V \rightarrow H$$

$$\rho_f : (H \times F) \rightarrow H$$

这样 ρ_v 将变量映射到它们的栈位置, $range(\rho_v)$ 表示是栈上的内存位置。假如一个位置 $h \in H$ 是动态分配的结构体存储位置,则 $\rho_f(h, f)$ 表示该结构体的 f 域,为了简化起见,表示成 $h || f$ 。可以知道 ρ_f 的定义域和值域的所有存储位置都是动态分配的。

在 H 中,对每个内存位置,引入下面几个辅助函数来获得它的属性:

函数 $\mathbb{H}_{entry} : H \rightarrow H$,如果某内存位置是由结构体指针变量(如指针 p)指向的,则返回结构体指针变量对应的内存位置 $(\rho_v(p))$;由域访问表达式(如 $p \rightarrow next$)指向的,则返回域访问表达式中结构体指针变量对应的内存位置 $(\rho_v(p))$ 。

函数 $\mathbb{H}_{hasField} : H \rightarrow \{true, false\}$,用来判断某内存位置是否由结构体域访问表达式指向。如果是,则返回 true;否则返回 false。

函数 $\mathbb{H}_{field} : H \rightarrow F \cup \{null\}$,如果内存位置是由结构体域访问表达式指向的,则返回其域名,否则返回 null。

3 定值引用算法

本节介绍所设计的过程内定值引用链的算法,它首先利用迭代的数据流分析算法得到到达定值信息;然后利用到达

定值分析的结果，得到每条语句的引用定值链；最后根据引用定值链，推导得到定值引用链。

3.1 到达定值分析

3.1.1 定值抽象

每条赋值语句被抽象为一个定值实体，每个定值实体 $d \in D \subseteq H \times H \times L$ 是一个三元组，用来表示程序中的一个定值点，如 $\langle h_1, h_2, l \rangle$ ， h_1 、 h_2 分别是赋值语句赋值号左、右两边表达式对应的内存抽象， l 是此语句的标号。这样到达定值分析计算的是，在每个程序点，沿所有的程序的执行路径能到达这个程序点的所有可能(may)的定值点的集合。

在定值抽象 D 上定义了一个格，格的底元是 \perp ，表示没有任何定值点；顶元是 \top ，表示程序的所有定值点都可行。对于任意的 $d_1, d_2 \in D$ ，格的并运算 \sqcup 定义为：

$$d_1 \sqcup d_2 = d_1 \cup d_2$$

3.1.2 数据流等式和转换方程

数据流方程如图 3 所示，其中符号 “_” 表示不关心具体的内存位置或标号，可以为任意的值。 p, q 分别表示指针变量 p 和 q 对应的内存位置，相应地用缩写形式 qf 表示域访问表达式 $q \rightarrow f$ 对应的内存位置，下面内容中此形式表示相同的含义。对任一语句 s ，引入两个程序点，分别代表正好到达该语句的入口位置和正好离开该语句的出口位置。函数 \mathbb{R}_{in} 和 \mathbb{R}_{out} 和辅助函数 \mathbb{R}_{init} 和 \mathbb{R}_{pre} 的作用是：

函数 $\mathbb{R}_{in}: L \rightarrow \mathcal{P}(D)$ ，计算可到达某语句入口程序点的定值实体的集合。

函数 $\mathbb{R}_{out}: L \rightarrow \mathcal{P}(D)$ ，计算可到达某语句出口程序点的定值实体的集合。

函数 $\mathbb{R}_{init}: \text{Stmts} \rightarrow L$ ，用来得到程序段的初始语句标号。

函数 $\mathbb{R}_{pre}: L \rightarrow \mathcal{P}(L)$ ，得到某语句在控制流图(容易从控制流语句的语法结构求得)中所有前驱语句的标号的集合。

对于赋值语句和条件语句的分析，引入函数 \mathbb{R}_{kill} 和 \mathbb{R}_{gen} 来计算：

函数 $\mathbb{R}_{kill}: S \rightarrow \mathcal{P}(D)$ ，被某语句注销的定值实体的集合，假如语句中对某个内存位置进行了赋值，则这个内存位置的其余定值实体会被注销。

函数 $\mathbb{R}_{gen}: S \rightarrow \mathcal{P}(D)$ ，语句中因赋值产生的定值实体的集合，只有赋值语句才产生定值实体。

既然每条赋值语句被抽象为一个三元组实体 $\langle h_1, h_2, l \rangle$ ，这样它不但可以用来表示对某个内存位置 h_1 的定值，而且记录和 h_1 右值别名的内存位置 h_2 ，它们指向相同的内存单元。如果 h_1 和 h_2 互为右值别名，则别名关系可以记作 $\langle h_1, h_2 \rangle$ 。这里我们用等价类表示[4]来表示指向同一个内存单元的所有指针变量对应的内存位置。这样的话：

如果右值别名序对有 $\langle h_n, h_{n-1} \rangle \langle h_{n-1}, h_{n-2} \rangle \dots \langle h_2, h_1 \rangle$ ，则可以用 h_1 作为它们的等价类表示，这样指向这个内存单元的变量对应的内存位置的右值别名都可以用 h_1 来表示，即通过传递性，有 $\langle h_n, h_1 \rangle \langle h_{n-1}, h_1 \rangle \dots \langle h_2, h_1 \rangle$ 。

如果某赋值语句抽象的定值实体为 $d_n = \langle h_3, h_2, l_n \rangle$ ，需要找出和 h_2 指向同一个内存单元的等价类表示，这里通过辅助函数 \mathbb{R}_{alias} (如图4)来实现，假如能够到达语句 l_n 的定值实体集合 $\mathbb{R}_{in}(l_n)$ 中有定值实体 $d_{n-1} = \langle h_2, h_1, l_{n-1} \rangle$ ，则可以知道和 h_2 右值

| \mathbb{R}_{kill} 和 \mathbb{R}_{gen} 方程 | |
|---|-------|
| $\mathbb{R}_{kill}([p = q])' = \{ \langle p, _ \rangle \}$ | |
| $\mathbb{R}_{kill}([p = q \rightarrow f])' = \{ \langle p, _ \rangle \}$ | |
| $\mathbb{R}_{kill}([p \rightarrow f = q])' = \{ \langle h \ f, _ \rangle \mid h \in \mathbb{R}_{alias}(\mathbb{R}_{in}(l), p) \}$ | |
| $\mathbb{R}_{kill}([p = \text{malloc}(\text{type})])' = \{ \langle p, _ \rangle \}$ | |
| $\mathbb{R}_{kill}([\text{free}(p)]) = \phi$ | |
| $\mathbb{R}_{kill}([b])' = \phi$ | |
| $\mathbb{R}_{gen}([p = q])' = \{ \langle p, h, l \rangle \mid h \in \mathbb{R}_{alias}(\mathbb{R}_{in}(l), q) \}$ | |
| $\mathbb{R}_{gen}([p = q \rightarrow f])' = \{ \langle p, h, l \rangle \mid h \in \mathbb{R}_{alias}(\mathbb{R}_{in}(l), qf) \}$ | |
| $\mathbb{R}_{gen}([p \rightarrow f = q])' = \{ \langle h \ f, h_1, l \rangle \mid h_1 \in \mathbb{R}_{alias}(\mathbb{R}_{in}(l), q), h \in \mathbb{R}_{alias}(\mathbb{R}_{in}(l), p) \}$ | |
| $\mathbb{R}_{gen}([p = \text{malloc}(\text{type})])' = \{ \langle p, \text{null}, l \rangle \}$ | |
| $\mathbb{R}_{gen}([\text{free}(p)]) = \phi$ | |
| $\mathbb{R}_{gen}([b])' = \phi$ | 数据流方程 |
| $\mathbb{R}_{in}(l) = \begin{cases} \phi & \text{if } l = \mathbb{R}_{init}(S) \\ \bigcup_{l' \in \mathbb{R}_{pre}(l)} \mathbb{R}_{out}(l') & \text{otherwise} \end{cases}$ | |
| $\mathbb{R}_{out}(l) = (\mathbb{R}_{in}(l) - \mathbb{R}_{kill}(l)) \cup \mathbb{R}_{gen}(l)$ | |

图 3 到达定值分析

Fig.3 Reaching-definition analysis

别名的内存位置为 h_1 ，这样用 h_1 去替代实体 d_n 中的 h_2 ，从而得到定值实体 $\langle h_3, h_1, l_n \rangle$ ，并向下传递。

因此在定值实体中所有引用的内存位置都用它们的等价类表示来替代。在数据流方程的 \mathbb{R}_{kill} 函数中，对于赋值语句等号左边的域访问表达式的指针变量用该指针的等价类表示来替代，而不关心等号右边的表达式；而 \mathbb{R}_{gen} 函数不但将赋值语句等号左边的域访问表达式的指针变量用该指针的等价类表示来替代，而且对于等号右边的表达式，也将用该表达式的等价类表示来替代。对各种赋值语句和条件语句的函数 \mathbb{R}_{kill} 和 \mathbb{R}_{gen} 的具体操作如图 3 所示。

辅助函数 \mathbb{R}_{alias} 如图 4 所示，体现了算法的核心。该函数主要用来找到能够到达某语句的入口程序点并可能(may)

```

 $\mathbb{R}_{alias}(D_{set}, h) \{$ 
1  $H_{set} = \phi;$ 
2  $\text{if}(D_{set} == \phi) \text{return } \{h\};$ 
3  $\text{for}(\text{each } \langle h_1, h_2, l \rangle \text{ in } D_{set}) \{$ 
4  $\text{if}(\mathbb{H}_{entry}(h) != \mathbb{H}_{entry}(h_1))$ 
5  $\text{continue};$ 
6  $\text{else if}(!\mathbb{H}_{hasField}(h_1))$ 
7  $\text{if}(\mathbb{H}_{entry}(h_2) != \text{null} \&\& !\mathbb{H}_{hasField}(h_2)) \{$ 
8  $\text{if}(\mathbb{H}_{hasField}(h))$ 
9  $H_{set} \cup = \mathbb{R}_{alias}(D_{set}, h_2 / \mathbb{H}_{field}(h));$ 
10  $\text{else}$ 
11  $H_{set} \cup = \{h_2\};$ 
12  $\}$ 
13  $\text{else if}(\mathbb{H}_{hasField}(h))$ 
14  $\text{if}(\mathbb{H}_{field}(h) == \mathbb{H}_{field}(h_1) \&\& !\mathbb{H}_{hasField}(h_2))$ 
15  $H_{set} \cup = \{h_2\};$ 
16  $\} // \text{end for}$ 
17  $\text{if}(H_{set} == \phi) H_{set} \cup = \{h\}$ 
18  $\text{return } H_{set};$ 
19  $\}$ 

```

图 4 辅助函数 \mathbb{R}_{alias}

Fig. 4 Helper function \mathbb{R}_{alias}

| Program | $\mathbb{R}_{in}/\mathbb{R}_{out}$ | Iterator 1 | Iterator 2 | Iterator 3 |
|----------------------|---|---|---|---|
| $[q=x]^1$ | $\mathbb{R}_{in}(1)$ | ϕ | ϕ | ϕ |
| | $\mathbb{R}_{out}(1)$ | $\langle q,x,1 \rangle$ | $\langle q,x,1 \rangle$ | $\langle q,x,1 \rangle$ |
| $[q!=null]^2$ | $\mathbb{R}_{in}(2)$ | $\langle q,x,1 \rangle$ | $\langle q,x,1 \rangle \langle q,t_1,8 \rangle \langle z,null,3 \rangle$ $\langle t_1,xn,4 \rangle \langle zn,t_1,5 \rangle \langle xn,z,6 \rangle$ $\langle t_2,z,7 \rangle$ | $\langle q,x,1 \rangle \langle q,t_1,8 \rangle \langle z,null,3 \rangle$ $\langle t_1,xn,4 \rangle \langle t_1,t_1n,4 \rangle \langle zn,t_1,5 \rangle$ $\langle xn,z,6 \rangle \langle t_1n,z,6 \rangle \langle t_2,z,7 \rangle$ |
| $[z=malloc(type)]^3$ | $\mathbb{R}_{out}(2)$ $\mathbb{R}_{in}(3)$ | $\langle q,x,1 \rangle$ | $\langle q,x,1 \rangle \langle q,t_1,8 \rangle \langle z,null,3 \rangle$ $\langle t_1,xn,4 \rangle \langle zn,t_1,5 \rangle \langle xn,z,6 \rangle$ $\langle t_2,z,7 \rangle$ | $\langle q,x,1 \rangle \langle q,t_1,8 \rangle \langle z,null,3 \rangle$ $\langle t_1,xn,4 \rangle \langle t_1,t_1n,4 \rangle \langle zn,t_1,5 \rangle$ $\langle xn,z,6 \rangle \langle t_1n,z,6 \rangle \langle t_2,z,7 \rangle$ |
| $[t_1=q->n]^4$ | $\mathbb{R}_{out}(3)$ $\mathbb{R}_{in}(4)$ | $\langle q,x,1 \rangle$ $\langle z,null,3 \rangle$ | $\langle q,x,1 \rangle \langle q,t_1,8 \rangle \langle z,null,3 \rangle$ $\langle t_1,xn,4 \rangle \langle zn,t_1,5 \rangle \langle xn,z,6 \rangle$ $\langle t_2,z,7 \rangle$ | $\langle q,x,1 \rangle \langle q,t_1,8 \rangle \langle z,null,3 \rangle$ $\langle t_1,xn,4 \rangle \langle t_1,t_1n,4 \rangle \langle zn,t_1,5 \rangle$ $\langle xn,z,6 \rangle \langle t_1n,z,6 \rangle \langle t_2,z,7 \rangle$ |
| $[z->n=t_1]^5$ | $\mathbb{R}_{out}(4)$ $\mathbb{R}_{in}(5)$ | $\langle q,x,1 \rangle \langle z,null,3 \rangle$ $\langle t_1,xn,4 \rangle$ | $\langle q,x,1 \rangle \langle q,t_1,8 \rangle \langle z,null,3 \rangle$ $\langle t_1,xn,4 \rangle \langle t_1,t_1n,4 \rangle \langle zn,t_1,5 \rangle$ $\langle xn,z,6 \rangle \langle t_2,z,7 \rangle$ | $\langle q,x,1 \rangle \langle q,t_1,8 \rangle \langle z,null,3 \rangle$ $\langle t_1,xn,4 \rangle \langle t_1,t_1n,4 \rangle \langle zn,t_1,5 \rangle$ $\langle xn,z,6 \rangle \langle t_1n,z,6 \rangle \langle t_2,z,7 \rangle$ |
| $[q->n=z]^6$ | $\mathbb{R}_{out}(5)$ $\mathbb{R}_{in}(6)$ | $\langle q,x,1 \rangle \langle z,null,3 \rangle$ $\langle t_1,xn,4 \rangle \langle zn,t_1,5 \rangle$ | $\langle q,x,1 \rangle \langle q,t_1,8 \rangle \langle z,null,3 \rangle$ $\langle t_1,xn,4 \rangle \langle t_1,t_1n,4 \rangle \langle zn,t_1,5 \rangle$ $\langle xn,z,6 \rangle \langle t_2,z,7 \rangle$ | $\langle q,x,1 \rangle \langle q,t_1,8 \rangle \langle z,null,3 \rangle$ $\langle t_1,xn,4 \rangle \langle t_1,t_1n,4 \rangle \langle zn,t_1,5 \rangle$ $\langle xn,z,6 \rangle \langle t_1n,z,6 \rangle \langle t_2,z,7 \rangle$ |
| $[t_2=q->n]^7$ | $\mathbb{R}_{out}(6)$ $\mathbb{R}_{in}(7)$ | $\langle q,x,1 \rangle$ $\langle z,null,3 \rangle \langle t_1,xn,4 \rangle$ $\langle zn,t_1,5 \rangle \langle xn,z,6 \rangle$ | $\langle q,x,1 \rangle \langle q,t_1,8 \rangle \langle z,null,3 \rangle$ $\langle t_1,xn,4 \rangle \langle t_1,t_1n,4 \rangle \langle zn,t_1,5 \rangle$ $\langle xn,z,6 \rangle \langle t_1n,z,6 \rangle \langle t_2,z,7 \rangle$ | $\langle q,x,1 \rangle \langle q,t_1,8 \rangle \langle z,null,3 \rangle$ $\langle t_1,xn,4 \rangle \langle t_1,t_1n,4 \rangle \langle zn,t_1,5 \rangle$ $\langle xn,z,6 \rangle \langle t_1n,z,6 \rangle \langle t_2,z,7 \rangle$ |
| $[q=t_2->n]^8$ | $\mathbb{R}_{out}(7)$ $\mathbb{R}_{in}(8)$ | $\langle q,x,1 \rangle \langle z,null,3 \rangle$ $\langle t_1,xn,4 \rangle \langle zn,t_1,5 \rangle$ $\langle xn,z,6 \rangle \langle t_2,z,7 \rangle$ | $\langle q,x,1 \rangle \langle q,t_1,8 \rangle \langle z,null,3 \rangle$ $\langle t_1,xn,4 \rangle \langle t_1,t_1n,4 \rangle \langle zn,t_1,5 \rangle$ $\langle xn,z,6 \rangle \langle t_1n,z,6 \rangle \langle t_2,z,7 \rangle$ | $\langle q,x,1 \rangle \langle q,t_1,8 \rangle \langle z,null,3 \rangle$ $\langle t_1,xn,4 \rangle \langle t_1,t_1n,4 \rangle \langle zn,t_1,5 \rangle$ $\langle xn,z,6 \rangle \langle t_1n,z,6 \rangle \langle t_2,z,7 \rangle$ |
| | $\mathbb{R}_{out}(8)$ | $\langle z,null,3 \rangle \langle t_1,xn,4 \rangle$ $\langle zn,t_1,5 \rangle \langle xn,z,6 \rangle$ $\langle t_2,z,7 \rangle \langle q,t_1,8 \rangle$ | $\langle z,null,3 \rangle \langle t_1,xn,4 \rangle \langle t_1,t_1n,4 \rangle$ $\langle zn,t_1,5 \rangle \langle xn,z,6 \rangle \langle t_1n,z,6 \rangle$ $\langle t_2,z,7 \rangle \langle q,t_1,8 \rangle$ | $\langle z,null,3 \rangle \langle t_1,xn,4 \rangle \langle t_1,t_1n,4 \rangle$ $\langle zn,t_1,5 \rangle \langle xn,z,6 \rangle \langle t_1n,z,6 \rangle$ $\langle t_2,z,7 \rangle \langle q,t_1,8 \rangle$ |

图5 单链表程序的到达定值

Fig.5 The reaching-definition information of a single linked list program

和参数 h 指向同一个位置的等价类表示的内存位置的集合。

\mathbb{R}_{alias} 算法首先初始化一个内存位置集合 $H_{set} \subset H$ 为空 (第1行), 然后判断某语句的入口程序点的到达定值抽象 $D_{set} \subset D$ 是否为空, 若为空, 则退出返回 h 本身 (第2行); 否则循环处理 D_{set} 集合中每个元素, 假设为 $\langle h_1, h_2, t \rangle$, 如果 $\mathbb{H}_{entry}(h)$ 和 $\mathbb{H}_{entry}(h_1)$ 不相同, 也就是它们的指针变量对应的内存位置不相同, 继续处理 D_{set} 中的下一个元素 (第4-5行); 否则, 如果 h_1 不是由域访问表达式指向的, 有两种情况:

1. h 是域访问表达式指向的, 则应该先找出和域访问表达式的指针变量指向相同位置的内存位置, 要求 $\mathbb{H}_{entry}(h_2)$ 不为空且 $\mathbb{H}_{hasField}(h_2)$ 为 false, 则 h 和 h_2 指向相同的内存位置。然后再递归找和 $h_2 // \mathbb{H}_{field}(h)$ 指向相同位置的内存位置 (第6-9行)。

2. h 是指针变量指向的, 并且 $\mathbb{H}_{entry}(h_2)$ 不为空且 $\mathbb{H}_{hasField}(h_2)$ 为 false, 则将 h_2 加入到 H_{set} 中 (第11行)。

如果 h 和 h_1 是同一个域访问表达式对应的内存位置, 并 h_2 是由指针变量指向的内存位置, 则用 h_2 替代 h , 即将 h_2 加入到 H_{set} 中 (第13-15行)。最后在循环结束后判定 H_{set} 是否为空, 为空则返回 h 本身 (第17行)。

3.1.3 示例

在这里通过下面的程序片段来说明到达定值分析的主要特性。它是对单链表的插入操作, 在单链表 x 的每个节点

后插入节点 z 。

```

[q = x]1;
while [q!=null]2 do {
    [z=malloc(type)]3;
    [t1 = q->next]4;
    [z->next = t1]5;
    [q->next = z]6;
    [t2 = q->next]7;
    [q = t2->next]8;
}

```

图5表述了到达定值分析利用 worklist 迭代算法 [4] 的迭代过程, 计算每次迭代中每个程序点的到达定值信息。

在 Iterator1 中, 在语句 1 中, 内存位置 q 和 x 右值别名, 这样出口程序点到达定值中增加 (gen) 定值实体 $\langle q,x,1 \rangle$; 到达语句 4 时, 因为 q 和 x 右值别名, 由别名函数可以找到 $\mathbb{R}_{alias}(\mathbb{R}_{in}(4), qn) = \{xn\}$, 即由 q 的别名 x 替代, 增加 $\langle t_1,xn,4 \rangle$; 到达语句 5 时, 增加 $\langle zn,t_1,5 \rangle$; 到达语句 6 时, 同理语句 4, 增加 $\langle xn,z,6 \rangle$; 到达语句 7 时, 由别名函数可以找到 $\mathbb{R}_{alias}(\mathbb{R}_{in}(7), qn) = \{z\}$, 故增加 $\langle t_2,z,7 \rangle$; 到达语句 8 时, 由别名函数可以找到 $\mathbb{R}_{alias}(\mathbb{R}_{in}(8), t_2n) = \{t_1\}$, 由于语句是对 q 的定值, 将所有到达此程序点对 q 的定值注销 (kill), 而增加 $\langle q,t_1,8 \rangle$ 。

经过一次迭代计算后, 分析没有到达稳定状态, 继续迭代计算, 从语句 1 和 8 到达循环的头部, $\mathbb{R}_{in}(2) = \mathbb{R}_{out}(1) \cup \mathbb{R}_{out}(8)$, 这样 q 可能 (may) 和 x , t_1 右值别名, 同理在定值实

体中对 q 的引用时用 x, t_1 去替代, 如到达语句 4 时, 将所有到达此程序点对 t_1 的定值实体注销, 而增加定值实体 $\langle t_1, xn, 4 \rangle \langle t_1, t_1 n, 4 \rangle$ 。

经过数次迭代计算后, 每个程序点的到达定值没有变化, 迭代算法到达稳定的状态, 到达定值分析结束, 得到如图 5 中 Iterator3 中所示的每个程序点的到达定值信息。

3.2 引用定值链

在到达定值信息的基础上, 可以计算引用定值链。引用定值链连接对一个内存位置的引用的语句到所有可能沿某条路径流经到该语句并对该内存位置定值的语句。

3.2.1 推导方程式

函数 $\mathbb{R}_{ud}: L \rightarrow \mathcal{P}(H \times L)$ 得到某语句引用的内存位置和所有可能沿某条路径到达该语句并对该内存位置定值的语句的标号之间的序对。

对于任一标号为 l 的语句 s , 根据语句入口程序点的到达定值信息, 有:

$$\mathbb{R}_{ud}(l) = \{ \langle h, l_1 \rangle \mid h \in \mathbb{R}_{uses}(s^l) \wedge \langle h, _ \rangle, l_1 \in \mathbb{R}_{in}(l) \} \quad (1)$$

其中, 函数 $\mathbb{R}_{uses}: S \rightarrow \mathcal{P}(H)$ 得到某语句中所有引用的域访问表达式对应的内存位置。

对于引用语句, \mathbb{R}_{uses} 函数定义如公式(2), 而其它类型的语句的 \mathbb{R}_{uses} 函数都为空。

$$\mathbb{R}_{uses}([p = q -> f]^l) = \{ h \mid f \mid h \in \mathbb{R}_{alias}(\mathbb{R}_{in}(l), q) \} \quad (2)$$

3.2.2 示例

根据图 5 中每条语句入口程序点已经得到的到达定值信息, 利用公式(1)可以求得如图 6 的 \mathbb{R}_{ud} 函数的信息; 再利用公式(2), 可以求得如图 6 中的引用定值链。

| l | $\mathbb{R}_{uses}(l)$ | $\mathbb{R}_{ud}(l)$ |
|-----|------------------------|--|
| 1 | ϕ | ϕ |
| 2 | ϕ | ϕ |
| 3 | ϕ | ϕ |
| 4 | $xn, t_1 n$ | $\langle xn, 6 \rangle \langle t_1 n, 6 \rangle$ |
| 5 | ϕ | ϕ |
| 6 | ϕ | ϕ |
| 7 | $xn, t_1 n$ | $\langle xn, 6 \rangle \langle t_1 n, 6 \rangle$ |
| 8 | zn | $\langle zn, 5 \rangle$ |

图 6 单链表程序的引用定值链

Fig.6 The use-definition chains of a single linked list program

3.3 定值引用链

在引用定值链信息的基础上, 可以推导得到定值引用链。定值引用链连接对一个内存位置的定值的语句到所有可能沿某条路径流经到对该内存位置引用的语句。

3.3.1 推导方程式

函数 $\mathbb{R}_{du}(l): L \rightarrow \mathcal{P}(H \times L)$, 得到某语句定值的内存位置和可能沿某条路径到达所有引用该内存位置的语句的标号之间的序对。

利用引用定值链, 求定值引用链的推导方程式为:

$$\forall l_1, l_2 \in L, h \in H, \langle h, l_2 \rangle \in \mathbb{R}_{ud}(l_1) \Rightarrow \langle h, l_1 \rangle \in \mathbb{R}_{du}(l_2) \quad (3)$$

3.3.2 示例

根据图 6 得到的引用定值链, 可以利用公式(3)可以求得程序的定值引用链, 如图 7 所示。

| l | $\mathbb{R}_{ud}(l)$ | $\mathbb{R}_{du}(l)$ |
|-----|--|---|
| 1 | ϕ | ϕ |
| 2 | ϕ | ϕ |
| 3 | ϕ | ϕ |
| 4 | $\langle xn, 6 \rangle \langle t_1 n, 6 \rangle$ | ϕ |
| 5 | ϕ | $\langle zn, 8 \rangle$ |
| 6 | ϕ | $\langle xn, 4 \rangle \langle xn, 7 \rangle \langle t_1 n, 4 \rangle \langle t_1 n, 7 \rangle$ |
| 7 | $\langle xn, 6 \rangle \langle t_1 n, 6 \rangle$ | ϕ |
| 8 | $\langle zn, 5 \rangle$ | ϕ |

图 7 单链表程序的定值引用链

Fig.7 The definition-use chains of a single linked list program

4 实验结果和分析

笔者在 SUIF[10]框架上, 实现了过程内的定值引用算法, 并做了相关的测试, 详细的程序和数据可见[3], 测试用例如表 1 所示, 它们大部分来自 Olden[5]中的测试程序。

表 1 测试用例的数据结构

Table 1 Data structures of some benchmark programs

| Program | Description | Data Structures |
|---------|------------------------------------|----------------------|
| BH | N-body simulation | Leaf-linked tree |
| EM3D | Simulation of electromagnetic wave | Bipartite |
| Power | Power system optimization | Hierarchical tree |
| Tsp | Traveling salesman problem | Balanced binary-tree |
| Treeadd | Tree addition | Binary tree |
| Insert | List insert | Linked-list |

选择这些测试用例的原因是它们创建和访问各种类型的动态链状数据结构, 如链表、树状结构等。表 2 给出了经过过程内定值引用链分析的统计数据, 分别给出了程序的行数、包含的过程体数量、分析过程中所耗的总时间、经过分析找出的所有定值引用序对的数量和文章[6]中算法给出的定值引用序对的数量。其中符号“/”表示文章[6]中没有对此程序进行测试。

表 2 过程内分析的统计数据

Table 2 Statistics after intraprocedural analysis

| Program | Lines | Procedures | Time(s) | DEF/USE | DEF/USE [6] |
|---------|-------|------------|---------|---------|-------------|
| BH | 1977 | 36 | 2.98 | 4 | 5 |
| EM3D | 352 | 15 | 0.17 | 7 | 8 |
| Power | 772 | 19 | 0.45 | 0 | 0 |
| Tsp | 532 | 14 | 0.41 | 15 | / |
| Treeadd | 99 | 3 | 0.01 | 0 | 0 |
| Insert | 48 | 3 | 0.01 | 5 | / |

可以看到最后两列对应程序的定值引用序对数目并不相同。这是因为更精确的别名分析,可以移除虚假的定值引用链。由于本文中采用的算法,分析得到了更精确的别名信息,移除了不再有效的到达定值,提高了定值引用链的精度,所以定值引用序对数量比[6]中的略少,从而也表明本文的算法更优越。

5 相关工作比较

过程内的定值引用链的分析技术,早已众所周知[9],过程间的分析技术也被提出[12],而扩展到带有指针或动态分配数据结构的程序中定值引用链的研究,早期的有[7, 8, 6]。

Pande 和 Landi 等[7]首次在一指针的 C 语言程序中,提出了一种过程间的定值引用链的算法。算法首先利用别名信息计算过程间的到达定值,然后利用到达定值的信息建立定值引用链,然而该算法忽略了对动态分配的数据结构的分析。

Altucher 和 Landi[8]运用一种新的命名策略来提高动态分配的内存位置的定值引用信息的精度。它被用来计算动态分配内存位置的扩展的一定别名(extended must aliases),然后利用此别名信息来提高定值引用链的精度,但是该算法将动态分配的结构体看成是一个整体对象,没有区分结构体中的各个不同域。

Hwang 等[6]通过逆向的数据流迭代来计算每条语句向上暴露的引用(upward exposed uses),然后再计算动态链状数据结构指针程序的过程间的定值引用链。该方法产生多级的域访问路径表达式,为了确定两个多级的域访问路径表达式是否指向同一个内存位置,必须进行访问表达式的比较,算法的时空开销较大。

本文采用正向的数据流迭代算法来计算到达定值信息,并在此基础上,推导求得每条语句的引用定值链和定值引用链。笔者受[13]的启发,在到达定值的分析过程中,抽象值记录赋值语句等号左,右两边的表达式和语句标号的三元组,不但保存内存位置的定值信息,而且保存了内存位置的右值别名信息。再根据语句入口程序点的到达定值信息,找到和指针变量右值别名的等价类表示,用它去替代指针变量对应的内存位置在定值实体中的位置,并向下传递别名。从而将到达定值分析和别名信息的获取在一个分析过程中完成,大大提高分析的效率。

6 结束语

提出了一种面向动态链状数据结构的过程内指针定值引用链的算法。该算法首先用正向的数据流迭代算法得到动态链状数据结构的每条语句入口和出口程序点的到达定值信息,然后利用语句入口程序点的到达定值信息和别名信息,计算语句的引用定值链;最后根据语句中的引用定值链信息,推导得到定值引用链。本文不足之处在于只处理过程内的定值引用信息,没有完成过程间的定值引用算法。对过程间定值引用链算法的研究将是我们接下来的主要工作内容。

References

- [1] E. Duesterwald, R. Gupta and M.L. Soffa. A demand-driven analyzer for data flow testing at the integration level [C]. In IEEE International Conference on Software Engineering, Berlin, Germany, March 1996: 575-586.
- [2] M.Wolfe. High Performance Compilers for Parallel Computing[M], Addison-Wesley, 1995.
- [3] Yu Zhang et al.. Pplsr:Parallel Programming Language with Shared Resource Specification[EB/OL]. <http://sug.ustcsz.edu.cn/pplsr>. 2009.
- [4] B. Steensgaard .Points-to Analysis in Almost Linear Time[C]. In Proceedings of the 23rd ACM Symposium on the Principles of Programming Languages, St. Petersburg Beach, Florida, 1996
- [5] Martin.C. Carlisle . Anne Rogers.Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines [EB/OL].<http://www.martincarlisle.com/olden.html>.1996.
- [6] Yaun-Shin Hwang, Joel Saltz. Interprocedural definition use chains of dynamic pointer-linked data structures[M]. Scientific Programming, IOS Press. 2003.
- [7] H.D. Pande, W.A. Landi and B.G. Ryder .Interprocedural def-use associations for c systems with single level pointers[C], IEEE Transactions on Software Engineering . May, 1994, 20(5): 385-402.
- [8] R. Altucher and W. Landi. An extended form of must alias analysis for dynamic allocation[C]. In Proceedings of the 22nd ACM Symposium on the Principles of Programming Languages, San Francisco, California, January 1995: 74-84.
- [9] A.V. Aho, R. Sethi and J.D. Ullman. Compilers: Principles, Techniques, and Tools .Addison-Wesley, 1986.
- [10] R.Wilson, R.French, C.Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers[R]. In ACM SIGPLAN Notices, December 1994, 29(12) : 31-37.
- [11] F Nielson, HR Nielson, C Hankin .Principle of program analysis[M].1999:41-44.
- [12] M.J. Harrold and M.L. Soffa. Efficient computation of interprocedural definition-use chains[C]. ACM Transactions on Programming Languages and Systems. 1994, 16(2):175-204.
- [13] B Hackett and R Rugina. Region-Based Shape Analysis with Tracked Locations[C]. In Proceedings of the 32nd ACM symposium on Principles of programming languages, Long Beach, California, January , 2005:310-323.
- [14] L.O. Andersen .Program Analysis and Specialization for the C Programming Language[D]. PhD thesis, DIKU, University of Copenhagen, 1994.