第41卷第2期 中国科学技术大学学报 Vol. 41, No. 2
2011年2月 JOURNAL OF UNIVERSITY OF SCIENCE AND TECHNOLOGY OF CHINA Feb. 2011

文章编号:0253-2778(2011)02-0164-09

一种动态共享数据结构的并发访问控制分析方法

张 伟[1,2],张 昱[1,2],汪 晨[1,2],付小朋[1,2]

(1.中国科学技术大学计算机科学技术学院,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)

摘要:共享数据的并发访问控制是并行编程的关键之一,而对动态共享数据结构的细粒度并发访问控制更是其中的难点.针对操作动态数据结构的指针程序,提出一种并发访问控制的分析方法,该方法采用形状图对程序运行时的动态数据结构和指针变量关系建立抽象,通过形状图推导完成对共享单元的访问控制分析.程序员只需要声明指针及指向单元的共享性和数据结构形状特征,编译器利用本文中的方法分析得到共享单元的访问控制点,并添加使用互斥锁进行访问控制的代码.

关键词:并行编程;程序分析;访问控制;形状图

中图分类号:TP312 **文献标识码**:A doi:10.3969/j.issn.0253-2778.2011.02.011

An analysis approach to concurrent access control for shared mutable data

ZHANG Wei[1,2], ZHANG Yu[1,2], WANG Chen[1,2], FU Xiaopeng[1,2]

(1. *School of Computer Science & Technology*, *University of Science & Technology of China*, *Hefei* 230027, *China*;
2. *Software Security Lab.*, *Suzhou Institute for Advanced Study*, *University of Science & Technology of China*, *Suzhou* 215123, *China*)

Abstract: Concurrent access control for shared data is extremely important for parallel programming. And it is challenging to perform fine-grained concurrent access control for mutable-shared data. To address these problems, a novel approach was presented to achieve concurrent access control for concurrent programs with pointers. A shape graph was used to establish the abstract relationship between dynamic data structures and pointer variables in the run time, inferring the synchronization points. Our solution only requires programmers to provide some necessary sharing and shape information of the pointers and the memory cells for program analysis, which produces the synchronized points for each shared memory cell. Then, the compiler generates the mutual-lock-based code in terms of these synchronized points.

Key words: concurrency programming; program analysis; concurrent access control; shape graph

0 引言

超线程、多核等并行体系结构的迅猛发展,对并行软件的研究提出了更多的需求.对共享资源的有效并发访问控制是并行编程的关键之一.在现今的并行编程实践中,程序员主要使用锁来控制对共享资源的并发访问,程序员需要决定锁的粒度并对加锁和解锁操作直接编程.使用粗粒度的锁,编程简单但并发度低;而使用细粒度的锁虽能提高并发度,但编程困难且易出错.当程序含有涉及指针操作的动

收稿日期:2010-04-30;**修回日期**:2010-05-26

基金项目:国家自然科学基金(90718026,60928004)资助.

作者简介:张伟,男,1983年生,硕士生.研究方向:并行程序分析技术.E-mail:wzhang17@mail.ustc.edu.cn

通讯作者:张昱,博士/副教授.E-mail:yuzhang@ustc.edu.cn

态共享数据结构(如链表)时,对它们的细粒度并发访问控制更是其中的难点.

为降低并行编程的难度,正在研究的高效能并行编程语言 X10[1-2],Chapel[3] 和 Fortress[4] 等都引入原子区(形如 atomic{…})这种语言构造使得基于共享内存的并行编程不必使用显式的低级同步. 采用原子区确实简化了对共享资源访问控制的编程[5],但是由于原子区仅标识哪段代码需要被原子隔离地执行,而这种原子隔离性的保证却需要由编译运行系统来承担. 编译运行系统为此需要分析识别程序中的共享数据单元以及这些单元形成的数据结构特征和操作特征等,再使用锁[6-7]或事务内存技术[8-9]来达到对共享单元的有效并发访问控制. 当前,这两类实现原子区的相关技术仍在研究中,并且已经遇到一些难以解决的问题. 例如,如何分析出指针指向的数据结构形状,从而结合共享数据结构特征给出行之有效的访问控制方式;如何降低事务内存系统的运行开销,系统调用和 I/O 操作可否出现在事务中、事务可否嵌套、事务与非事务代码之间的隔离性等.

为了平衡程序员和编译器两者的负担,笔者所在的课题组正在探索一种新的并发访问控制编程模型[10],其主要思路是:程序员不用直接管理对共享数据的并发访问控制,而只需要通过语言提供的语言构造来声明程序中的共享资源及其使用特征;编译器根据程序员提供的信息通过程序分析识别共享单元及其访问控制点,然后为程序自动插入共享单元并发访问控制的代码. 基于上述思想本课题组已完成对静态整型数据的访问控制分析和实现,包括生产者消费者的例子.

本文研究动态共享数据结构的并发访问控制编程模型. 程序员只需要声明指针及其指向单元的共享性和数据结构形状特征,就能按串行的思维对这些数据结构进行编程. 我们设计了一套基于形状图的程序分析方法来分析程序中需要进行并发控制的动态共享单元及其访问控制点,并给出了利用分析结果生成基于锁的访问控制代码的方法. 所提出的方法已能有效解决单链表的并发访问控制.

本文的贡献主要有以下几点:

(Ⅰ) 定义一种简单有效的形状图,用于抽象程序运行时的动态数据结构和指针变量关系,并给出一组形状图推导规则用于推导程序的运行对形状图状态产生的变化.

(Ⅱ) 结合语言的维持语义和形状图,设计一种推导线程需要维持的共享数据(包括共享指针变量、动态共享数据结构中的节点)以及维持起始点的方法.

(Ⅲ) 利用上述的维持分析结果,编译器可以为程序添加基于锁的访问控制代码,从而使程序在并行执行时能够正确同步.

1 语言定义

为了便于形式讨论本文的分析方法,这里给出一个强调指针的类 C 并行语言的抽象语法,如图 1 所示.

(Prog):	prog $::= s_1 \parallel \cdots \parallel s_n$
(Stmts):	$s \in S \quad s \ ::= [e_0 = e_1]^l \mid \{s\} \mid s_0 ; s_1 \mid [e = \mathrm{malloc(type)}]^l$
	$\mid [\mathrm{free}(e)]^l \mid \mathrm{if}\ [b]^l\ \mathrm{then}\ s_0\ \mathrm{else}\ s_1 \mid \mathrm{while}\ [b]^l s$
(Exp)	$e \in E \quad e \ ::= \mathrm{null} \mid x \mid e \rightarrow f$
(BExp)	$b \in B \quad b \ ::= \mathrm{true} \mid \mathrm{false} \mid x \mid x != \mathrm{null} \mid \cdots$
(Var)	$x \in V$
(Field)	$f \in F = \{f_1, \cdots, f_n\}$
(Lab)	$l \in L$

图 1 语言的抽象语法

Fig. 1 Abstract syntax of the language

一个程序 prog 由 n 个并行执行的线程组成,每个线程执行语句 s_i. 程序中的数据类型 type 都是结构体类型,因此所有声明的变量都是指向结构体的指针变量. 程序员可以声明指针指向的数据结构形状(图 1 中略去了这些声明性语言构造),如指向单链表. 语言中禁止使用取地址(&)、强制类型转换和指针算术运算. malloc 和 free 被看成是语言预定义的函数,并且满足程序安全的基本要求. 语言级的赋值语句以及 if,while 中的条件测试均为原子执行的原子命令.

源程序中的多级域访问表达式假定全被规范化为一级的域访问,也就是域访问表达式 $x \rightarrow f$ 中的 x 是指针变量,f 为域名,并且只允许域访问表达式在等式的一边出现. 从而,在下面的程序分析中只需考虑三种可能的指针赋值语句:$p = q$, $p = q \rightarrow f$, $p \rightarrow f = q$. 另外,条件表达式也通过规范化变成只存在对指针变量的非空判断.

本文的分析是过程内的,若源程序包含函数调用,在没有函数递归调用的情况下,可以通过将该函数内联到调用点来免去过程间分析.

1.1 共享单元

程序中的数据单元分为静态和动态两种,前者

是声明变量所对应的单元,后者是通过 malloc 动态分配的单元. 在程序执行中可能被多个线程访问的数据单元称为共享单元. 在图 1 的简化语言中,静态数据单元由声明为共享的指针变量、声明为指向共享的局部指针变量以及其他局部指针变量组成;动态数据单元由动态的共享单元和动态的线程私有单元组成. 本文默认全局指针变量为共享变量,局部指针变量为私有变量. 动态数据单元只有在被共享的指针变量指向,或者是被其他共享单元的指针域指向时,才能作为共享单元被多个线程访问.

1.2 共享单元的维持语义

在程序中,若涉及对共享单元的读写,我们希望这些读写操作被视为在无干扰的环境下执行,即其他线程在执行过程中不会改变这些共享单元的状态. 为了达到这个目标,我们需要设计对共享单元加锁、解锁的方法. 该方法从单个线程的角度考虑对共享单元的维持. 所谓共享单元的维持是指线程 T 在执行某段代码期间要访问某共享单元 s,并且禁止其他线程在此期间访问 s,这种对共享单元访问的保护行为称作维持. 这段代码执行期的开始点和结束点分别称作共享单元 s 的维持开始点和结束点. 例如,在以下程序片段

```
[p=H]¹;  //在语句 1 前维持 H 指向的共享单元
while [p! =null]²{
  [q=p]³;  //在语句 3 前继续维持 p 所指向的共享单元
  [p=q→next]⁴;  //在语句 4 前开始维持 q→next 指向的
                    共享单元
}
```

中,假设 H 是共享指针变量,p,q 为局部指针变量,由于 H 是共享指针变量,因此 H 指向的单元为共享. 语句 1 第一次引用 H,导致 H 指向的共享单元维持开始,又因为语句 1 和 3 通过赋值使局部指针变量 p,q 也指向了 H 指向的单元,使得对该单元的最后一次引用在语句 4 后结束. 同理,语句 4 使 p 指向 q→next 指向的单元,该单元由于被共享单元的域指向,因此也为共享单元,它的维持从本次循环的语句 4 执行前开始,到下一次循环的语句 4 执行后结束.

基于上述的这种指针变量与动态单元之间的引用关系,使程序员能按串行思维编写推理程序,语言规定如下的维持语义:对于程序中的共享单元,从指向它的指针变量第一次被引用开始,直到所有指向它的指针变量不再被引用结束,在这段执行过程中该共享单元被维持.

上述维持语义描述在程序执行中每个被访问共享单元的维持开始和结束情况,访问控制粒度较细,不会明显降低并发度.

为了完成基于维持语义的访问控制,需要知道程序在其运行时动态数据单元的演变和访问情况,并且区分哪些是可能被多个线程访问的共享单元,同时分析得到这些共享单元在程序运行时维持的开始和结束点. 为解决这些问题,本文引入形状图来帮助分析推导.

2 形状图的定义与推导

形状图用来抽象程序中动态分配的数据单元以及这些单元之间的指向关系. 根据不同语句对形状图产生的变换来模拟程序执行中动态数据的具体形态. 最终通过形状图提供的信息分析动态共享单元及其维持.

2.1 形状图的定义

$G=(N,E,T)$表示一个形状图,其中 N 为节点集合,E 为边集合, T 为节点与其类型的映射关系集合.

n_N 表示空节点,n_D 表示悬空节点. 对于每个图 G 的节点集合 N 初始时都包含这两个节点.

$K=\{k_A,k_S\}$表示节点类型集合,k_A 表示抽象节点,k_S 表示结构节点.

$T=\{(n,k)\mid n\in N,k\in K\}$表示节点与其类型的序对集合,节点在创建时会对应一种类型.

$V=V_S\cup V_L$ 为指针变量集合,V_S 表示共享指针变量集合,V_L 表示局部指针变量集合.

$E=E_P\cup E_F$ 表示边集合,E_P 为指向边集合,E_F 为域边集合.

$E_P=\{(v,n)\mid v\in V,n\in N\}$表示指向边集合,$(v,n)$表示指针变量 v 指向节点 n.

$E_F=\{(n_1,f,n_2)\mid n_1,n_2\in N,f\in F\}$表示域边集合,其中 F 是域名集合,(n_1,f,n_2)表示节点 n_1 的域 f 指向节点 n_2.

$\mathbb{G}$ 表示所有形状图的集合,$\mathbb{N}$ 表示所有节点的全集,$\mathbb{E}$ 表示所有边的全集.

结构节点和抽象节点分别对应内存中某种结构类型的一个动态单元和满足某种性质的零个或多个动态单元.

2.2 形状图变换规则

抽象节点代表零个或多个节点,在分析过程中

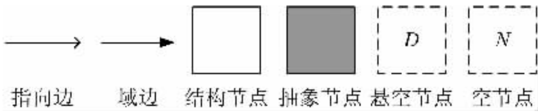

图 2　形状图的节点和边示例

Fig. 2　Example of nodes and edges in shape graph

用于抽象某个指针指向的单元的特征，以便进一步推导. 本文仅考虑对单链表的推导，因此抽象节点均代表单链表中的某段(包含零个或多个连续的单链表节点)，其他动态结构的推导可以在本文的基础上进行扩展定义. 当条件表达式对某个指向单链表的指针进行非空判断时，如果该指针为空则表示该指针指向的抽象节点不包含任何结构节点，因此可将该抽象节点转换成空节点如图 3(a)所示；否则表示该抽象节点至少含有一个结构节点，因此将其从抽象节点中分离出来，替代原来抽象节点所在的位置，如图 3(b)所示.

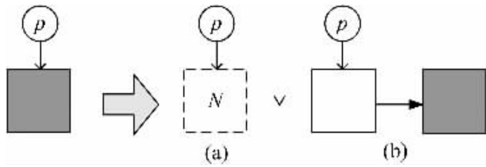

图 3　抽象节点展开示例

Fig. 3　Unfolding an abstract node

函数 $\mathrm{unfold}_1(G,n)=G'$ 表示在形状图 G 中的抽象节点 n 为空时，所得到的展开后的形状图 G'. 假设 $G=(N,E,T)$，$n\in N$，$(n,k_A)\in T$，$G'=(N',E',T')$，则

$$N'=N-\{n\},\ T'=T-\{(n,k_A)\mid(n,k_A)\in T\}$$
$$E'=E-\{(v,n)\mid(v,n)\in E\}\cup\{(v,n_N)\mid(v,n)\in E\}-\{(n',f,n)\mid(n',f,n)\in E\}\cup\{(n',f,n_N)\mid(n',f,n)\in E\}$$

函数 $\mathrm{unfold}_2(G,n)=G'$ 表示在形状图 G 中的抽象节点 n 为非空时，所得到的展开后的形状图 G'. 假设 $G=(N,E,T)$，$n\in N$，$(n,k_A)\in T$，$G'=(N',E',T')$，则

$$n_1\notin N,\ N'=N\cup\{n_1\},$$
$$T'=T-\{(n,k_A)\}\cup\{(n_1,k_A)\}\cup\{(n,k_S)\}$$
$$E'=E-\{(n,f,n_2)\mid(n,f,n_2)\in E\}\cup\{(n_1,f,n_2)\mid(n,f,n_2)\in E\}\cup\{(n,f,n_1)\}$$

展开时先创建新的抽象节点 n_1，并修改节点 n 的类型为结构节点，最后修改与 n 有关的各条指向边和域边.

当形状图中存在相邻的一个或多个没有任何指针变量指向的非空节点时，为了使形状图更具一般性，则将这多个节点合并成一个抽象节点. 为了定义合并规则，首先定义以下几个辅助谓词和函数

$$\frac{n\in N\quad\forall v\in V\quad(v,n)\notin E}{\mathrm{unref}((N,E,T),n)}$$

谓词 $\mathrm{unref}(G,n)$ 表示形状图 G 中的节点 n 没有指针变量指向

$$\frac{(n_1,f,n_2)\in E\quad\mathrm{unref}(G,n_1)\quad\mathrm{unref}(G,n_2)}{\mathrm{two}(G,n_1,n_2)}$$

谓词 $\mathrm{two}(G,n_1,n_2)$ 表示形状图 G 中相邻节点 n_1 和 n_2 都没有指针变量指向.

函数 $\mathrm{fold}_L(G,n)=G'$ 表示向左合并两个相邻的节点，假设 $G=(N,E,T)$，$G'=(N',E',T')$

$$\exists n_0,\ n_1\in N,\ (n_0,f,n_1)\in E,\ \mathrm{two}(G,n_1,n),$$
$$N'=N-\{n_1\},$$
$$T'=T-\{(n_1,_),(n,_)\}\cup\{(n,k_A)\}$$
$$E'=E-\{(n_1,f,n),(n_0,f,n_1)\}\cup\{(n_0,f,n)\}$$

函数 $\mathrm{fold}_R(G,n)=G'$ 表示向右合并两个相邻的节点，假设 $G=(N,E,T)$，$G'=(N',E',T')$

$$\exists n_0,\ n_1\in N,\ (n_0,f,n_1)\in E,\ \mathrm{two}(G,n,n_0),$$
$$N'=N-\{n_0\},$$
$$T'=T-\{(n_0,_),(n,_)\}\cup\{(n,k_A)\}$$
$$E'=E-\{(n,f,n_0),(n_0,f,n_1)\}\cup\{(n,f,n_1)\}$$

函数 $\mathrm{fold}(G,n)=G'$ 采用递归定义，不断对 n 左右相邻的节点进行合并，直到没有满足合并条件的节点时返回合并后的形状图 G'.

let $G_1=\mathrm{fold}_L(G,n)$ in let $G'=\mathrm{fold}_R(G_1,n)$ in
　if $G'==G$ then G' else $\mathrm{fold}(G',n)$

函数 $\mathrm{tran}(G,s)=G'$ 表示形状图 G 经过一条语句 s 后变换成 G'，假设 $G=(N,E,T)$，$G'=(N',E',T')$，根据语句不同有如下规则：

(Ⅰ) $p=q$

let $N_0=\{n\mid(p,n)\in E\}$ in let $\{n\}=N_0$ in
　let $E_1=E-\{(p,n)\}\cup\{(p,n')\mid(q,n')\in E\}$ in
　$G'=\mathrm{fold}((N,E_1,T),n)$

(Ⅱ) $p=q\to f$

let $N_0=\{n\mid(p,n)\in E\}$ in let $\{n\}=N_0$ in
　let $E_1=E-\{(p,n)\}\cup\{(p,n_2)\mid(q,n_1)\in E,(n_1,f,n_2)\in E\}$ in
　$G'=\mathrm{fold}((N,E_1,T),n)$

（Ⅲ）$p \rightarrow f = q$

$N' = N$, $T' = T$

$E' = E - \{(n_1, f, n_2) \mid (p, n_1) \in E, (n_1, f, n_2) \in E\} \cup \{(n_1, f, n_2) \mid (p, n_1) \in E, (q, n_2) \in E\}$

赋值语句对形状图产生的变化是根据赋值后的指向关系修改图中各类边的指向. 例如，$p = q \rightarrow f$ 语句会从边集合中删除 p 原来的指向边，并添加 p 指向 $q \rightarrow f$ 节点的边. 另外，由于指针 p 被重新定值，因此可能导致原先指向的节点没有指针变量指向，所以调用 fold 函数进行一次合并操作.

（Ⅳ）$p = \text{malloc(type)}$

$n_1 \notin N$,

let $N_0 = \{n \mid (p, n) \in E\}$ in

　let $\{n\} = N_0$ and let $N_1 = N \cup \{n_1\}$ in

　　let $E_1 = E - \{(p, n)\} \cup \{(p, n_1)\}$ in

　　　let $T_1 = T \cup \{(n_1, k_S)\}$ in

　　　　$G' = \text{fold}((N_1, E_1, T_1), n)$

（Ⅴ）free(p)

$N' = N - \{n \mid (p, n) \in E\}$,

$T' = T - \{(n, k_S) \mid (p, n) \in E, (n, k_S) \in T\}$

$E' = E - \{(p, n) \mid (p, n) \in E\} \cup \{(p, n_N)\}$

分配语句会创建新的结构节点，改变 p 的指向边，并调用合并函数. 释放语句则删除被释放的节点，并且要求被释放的节点类型是结构节点否则报错，为了避免指针悬空，删除后会将 p 指向空节点. 此外 free 前会先检查是否有其他指针指向该节点，如果有则给出警告.

（Ⅵ）(p! =null)

case true:

let $N_0 = \{n \mid (p, n) \in E, n \in N, (n, k_A) \in T\}$ in

　if $\exists n_0 \in N_0$ in

　　then $G' = \text{unfold}_2(G, n_0)$

　　else $G' = G$

case false:

　与 case true 基本相同，只是将 unfold_2 改成 unfold_1.

由于整型的各种判断不影响形状图的变化，因此将这些判断忽略. 条件判断由于存在真假两种分支，因此在形状图变换时不同的条件分支处理也不同. 非空判断主要是针对指针指向抽象节点的情况，根据其是否非空而选择 2.2 中的不同展开规则来处理，而指针指向其他类型节点将不改变形状图的形态.

3 维持分析

利用形状图变换规则分析某个线程执行的程序代码，可以得到其不同程序点处的形状图，从而得到不同程序点处动态数据单元和指针之间的指向关系. 在此基础上利用语句中指针对动态数据的访问关系，就可以推导出线程运行时需要维持的动态共享单元及其维持的起始和结束位置.

维持分析包括：①共享推导. 识别哪些动态数据单元是共享单元；②维持推导. 每个共享单元维持开始与结束的位置，以及在这些位置可以通过哪些指针访问该单元.

3.1 共享推导

动态数据单元只有在被共享的指针变量指向，或者是被其他共享单元的指针域指向时，才能成为共享单元. 因此可以根据形状图中的指向关系来确定哪些节点是共享的.

$$\frac{v \in V_S}{\text{shared}(v)}$$

$$\frac{\text{shared}(v) \quad (v, n) \in E \quad n \in N}{\text{shared}((N, E, T), n)}$$

$$\frac{\text{shared}((N, E, T), n) \quad (n, f, n') \in E \quad n' \in N}{\text{shared}((N, E, T), n')}$$

谓词 shared(v)表示指针变量 v 为共享指针变量，谓词 shared(G, n)表示图 G 中节点 n 为共享节点.

假如存在 shared(v)和(v, n)，其中 v 表示共享指针变量，n 表示某个结构节点或抽象节点，(v, n)表示由 v 指向 n 的边，则可以推导出节点 n 为共享单元.

假如存在 shared(G, n)和(n, f, n')，其中 n 表示某个已确定的共享节点，(n, f, n')表示由 n 通过域 f 指向 n' 的域边，则可以推导出节点 n' 为共享单元.

3.2 维持推导

依据共享单元的维持语义，若语句 s 入口处的形状图中某个属性为共享的节点在语句 s 中被指针引用，但是该节点尚未被维持，则认为在语句 s 的入口处是该节点的维持开始. 当某个已被维持的共享节点在执行语句 s 后不再被其他指针引用或者该节点不再有共享属性时，则认为在语句 s 的出口处是该节点的维持结束.

S 为语句集合，L 为语句标签集合，如图 1 定义所示.

$\mathbb{H}\subseteq N$ 表示形状图中已被维持的节点集合，这些节点一定是结构节点或抽象节点. $\mathbb{H}$ 表示节点集合的幂集.

P 为指向节点的指针访问路径集合.

$\mathbb{R}=\mathscr{P}(P)$ 表示访问路径的幂集，用于表示多个节点的访问路径集合.

谓词 lastrefer(v, l)表示变量 v 在语句 l 中是上一次定值的最后一次引用. 该谓词是用于判断某个指针变量指向的节点能否被释放的条件之一. 使用已有的变量到达定值分析[11]可以得到上述谓词的结果，文中只作为判断条件使用，不再给出具体的定义.

函数 $f_{\text{start}}:\mathbb{G}\times\mathbb{H}\times S\to\mathbb{R}\times\mathbb{H}$ 用于推导节点维持的开始. 它根据给定的形状图 G 和维持节点集合 H，推导执行语句 s 时新增的维持节点集合(即节点维持的开始)，并返回这些节点的指针访问路径集合. 根据 s 的不同 f_{start} 定义为

$f_{\text{start}}(G,H,[p=q]^l)=$
　let $N_0=\{n|(q,n)\in E,\text{shared}(G,n),n\notin H\}$ in
　　$(\{q|N_0\neq\varnothing\},\ H\cup N_0)$

$f_{\text{start}}(G,H,[p\to f=q]^l)=$
　let $N_0=\{n|((q,n)\in E,\text{shared}(G,n),n\notin H)\}$ in
　　let $N_1=\{n|(p,n)\in E,\text{shared}(G,n),n\notin H)\}$ in
　　$(\{p|N_0\neq\varnothing\}\cup\{q|N_1\neq\varnothing\},H\cup N_0\cup N_1)$

$f_{\text{start}}(G,H,[p=q\to f]^l)=$
　let $N_0=\{n|(q,n)\in E,\text{shared}(G,n),n\notin H\}$ in
　　let $N_1=\{n|(q,n_1)\in E,(n_1,f,n)\in E,\text{shared}(G,n),n\notin H\}$ in
　　$(\{q|N_1\neq\varnothing\}\cup\{q\to f|N_2\neq\varnothing\},\ H\cup N_0\cup N_1)$

$f_{\text{start}}(G,H,[p=\text{malloc(type)}]^l)=(\varnothing,H)$

$f_{\text{start}}(G,H,[\text{free}(p)]^l)=$
　let $N_0=\{n|(p,n)\in E,\text{shared}(n),n\in H\}$ in
　　$(\varnothing,H-N_0)$

$f_{\text{start}}(G,H,[(p!\ =\text{null})]^l)=$
　let $N_0=\{n|(p,n)\in E,\text{shared}(G,n),n\notin H\}$ in
　　$(\{p|N_0\neq\varnothing\},H\cup N_0)$

函数 $f_{\text{end}}:\mathbb{G}\times\mathbb{H}\times S\to\mathbb{R}\times\mathbb{H}$ 用于推导节点维持的结束. 它根据给定的形状图 G 和维持节点集合 H，推导语句 s 执行后需要释放的节点集合(即节点维持的结束)，并返回这些节点的指针访问路径集合. 其定义为

$f_{\text{end}}(G,H,[p=q]^l)=$
　let $N_0=\{n|n\in H,\ \neg\text{shared}(G,n)\}$ in
　　let $N_1=\{n|(q,n)\in E,n\in H,(v,n)\in E,\text{lastrefer}(v,l)\}$ in
　　　$(\{p|N_0\neq\varnothing\}\cup\{q|N_1\neq\varnothing\},H-N_0-N_1)$

$f_{\text{end}}(G,H,[p\to f=q]^l)=$
　let $N_0=\{n|n\in H,\ \neg\text{shared}(G,n)\}$ in
　　let $N_1=\{n|(q,n)\in E,n\in H,(v,n)\in E,\text{lastrefer}(v,l)\}$ and
　　　$N_2=\{n|(p,n)\in E,n\in H,(v,n)\in E,\text{lastrefer}(v,l)\}$ in
　$(\{p\to f|N_0\neq\varnothing\}\cup\{q|N_1\neq\varnothing\}\cup\{p|N_2\neq\varnothing\},H-N_0-N_1-N_2)$

$f_{\text{end}}(G,H,[p=q\to f]^l)=$
　let $N_0=\{n|n\in H,\neg\text{shared}(G,n)\}$ in
　　let $N_1=\{n|(q,n)\in E,n\in H,(v,n)\in E,\text{lastrefer}(v,l)\}$ and
　　　$N_2=\{n|(q,n_1)\in E,(n_1,f,n)\in E,n\in H,(v,n)\in E,\text{lastrefer}(v,\ l)\}$ in
　$(\{p|N_0\neq\varnothing\}\cup\{q|N_1\neq\varnothing\}\cup\{q\to f|N_2\neq\varnothing\},H-N_0-N_1-N_2)$

$f_{\text{end}}(G,H,p=\text{malloc(type)})=$
　let $N_0=\{n|n\in H,\neg\text{shared}(G,n)\}$ in
　　let $N_1=\{n|(p,n)\in E,\text{shared}(p),n\in H\}$ in
　　　$(\{p\},H-N_0-N_1)$

$f_{\text{end}}(G,H,\text{free}(p))=(\varnothing,H)$

$f_{\text{end}}(G,H,[(p!\ =\text{null})]^l)=$
　let $N_0=\{n|(p,n)\in E,n\in H,(v,n)\in E,\text{lastrefer}(v,l)\}$ in
　　$(\{p|N_0\neq\varnothing\},H-N_0)$

本文定义的 malloc 和 free 函数对共享单元的分配及其锁初始化和共享单元的释放及其锁释放操作是由我们内建的 malloc 和 free 函数自身实现的. 因此只要修改节点的维持属性. 另外，当 free 某共享节点时，则要求该节点处在被维持的状态，因此释放前需要进行判断.

函数 $F_A:\mathbb{G}\times\mathbb{H}\times\mathbb{R}\times\mathbb{R}\times S\to\mathbb{G}\times\mathbb{H}\times\mathbb{R}\times\mathbb{R}$ 是维持分析推导的全局函数，具体定义为

$F_A(G,H,M_S,M_E,s)=$
　let $(M_S{}',H_1)=f_{\text{start}}(G,H,s)$ in
　　let $G'=\text{tran}\ (G,s)$ in
　　　let $(M_E{}',H')=f_{\text{end}}(G',H_1,s)$ in
　　　　$(G',H',M_S{}',M_E{}')$

形状图的维持分析包含三个阶段：①根据语句

s 入口点的形状图 G 和维持节点集合 H，通过函数 f_{start} 获得新增维持节点的指针路径集合 M_S'，并返回更新后的维持节点集合 H_1；②通过函数 tran 对形状图 G 进行变换得到语句 s 执行后的形状图 G'；③通过函数 f_{end} 对变换后的形状图 G' 推导，获得需要释放节点的指针访问路径集合 M_E'，并返回更新后的维持节点集合 H'.

3.3　维持的数据流分析

利用数据流分析[11]方法，可以得到程序每一点的维持信息.

我们将程序中每条语句和条件表达式当成数据流中的基本块并给其标签，下面 2 个函数用来获得给定程序的起始基本块的标签以及基本块之间的流向关系.

Init：$S \rightarrow L$ 表示程序开始点的标签.

flow：$S \rightarrow L \times L$ 表示语句在正向数据流中的标签序列.

函数 $\mathbb{H}_{in}, \mathbb{H}_{out}: L \rightarrow \mathscr{P}(\mathbb{G} \times \mathbb{H} \times \mathbb{R} \times \mathbb{R})$ 分别表示计算数据流中每条语句入口和出口处的维持信息数据流值. 维持信息数据流值是一个四元组集合，每个元组由形状图、维持节点集合、维持开始和结束节点的指针访问路径集合组成.

维持分析数据流方程

$$\mathbb{H}_{in}([s]^l) = \begin{cases} (\mathrm{Init}G, \varnothing, \varnothing, \varnothing), \text{ if } l = \mathrm{Init}(S) \\ \qquad \text{otherwise} \\ \mathbb{H}_{in}[l] = \bigcup \mathbb{H}_{out}(l'), \\ \qquad (l', l) \in \mathrm{flow}(S) \end{cases}$$

$$\mathbb{H}_{out}([s]^l) = \bigcup \{F_A(G, H, M_S, M_E, s)) \mid (G, H, M_S, M_E) \in \mathbb{H}_{in}(l)\}$$

InitG 表示并行程序初始的形状图，初始时每个共享指针变量各指向一个抽象节点. 因为每个并行块开始时不能确定共享指针是否被其他线程指向某个内存区域，因此认为它指向某个堆上的空间，用抽象节点表示.

本方法采用正向数据流分析，语句入口的维持信息数据流值由它所有前驱语句出口处的维持信息合并得到. 语句出口处的维持信息数据流值则通过维持分析推导函数 F_A 对该语句入口处维持信息幂集中的每个元素分别进行推导获得.

数据流分析中最复杂的是对循环结构进行推导，因为循环执行是个动态过程，存在某些节点需要经过跨迭代的维持和释放. 因此对循环结构的分析会进行多次迭代推导，直到满足前后两次迭代分析结束产生的数据流值相同为止. 形状图的合并规则将没有指针变量指向的节点合并成一个抽象节点，使得被展开的图形可以再度收缩，从而保证了迭代推导的终止.

3.4　实例分析

图 4 是一个单链表逆置的程序片段以及对该程序的形状图推导过程. 程序中 H 为全局共享指针变量，p，q 为局部指针变量. 节点中的二元组第一元表示节点是否共享，第二元代表节点是否处在维持状态.

初始时全局共享指针变量 H 指向一个抽象节点，循环语句的条件判断导致每次进入循环迭代都会将抽象节点展开. 语句 2 对 H 指向的节点引用使得该节点开始维持，直到语句 4 执行后，由于没有共享指针指向该节点，因此不具有共享性，维持结束. 此外语句 4 又对 $q \rightarrow$next 指向的节点引用，因此维持开始，并且根据形状图的推导可以得出该节点是在语句 4 第二次迭代执行后维持结束.

第三次迭代结束时使用合并规则，推导产生的图形与第二次迭代结束时产生的图形相同，因此可以结束对循环的分析. 推导方法将每次迭代产生的节点维持开始和结束位置的指针访问路径集合保存，以便添加锁代码时使用，具体情况见表 1 所示.

表 1　指针访问路径集合(维持开始和结束位置)

Tab. 1　Set of pointer accessing path

语句	Iter 1		Iter 2		Iter 3	
	M_S	M_E	M_S	M_E	M_S	M_E
2	$\{H\}$	∅	∅	∅	∅	∅
3	∅	∅	∅	∅	∅	∅
4	$\{q \rightarrow \mathrm{next}\}$	$\{q\}$	$\{q \rightarrow \mathrm{next}\}$	$\{q\}$	$\{q \rightarrow \mathrm{next}\}$	$\{q\}$
5	∅	∅	∅	∅	∅	∅
6	∅	∅	∅	∅	∅	∅

3.5　内存泄漏检查

形状图不仅有助于分析动态数据的维持，也可以根据图中节点和指针的状况检查程序中潜在的问题. 例如，通过检查在程序执行中是否存在没有入边的结构节点或抽象节点可以发现是否存在内存泄漏

$$\mathrm{leak}((N, E, T), n) = \forall v \in V, n' \in N, \neg(v, n) \in E \wedge \neg(n', f, n) \in E$$

谓词 leak(G, n)表示判断形状图中的节点 n 是否存在内存泄漏. 可以在形状图推导指针赋值语句前使用，判断赋值是否导致内存泄漏.

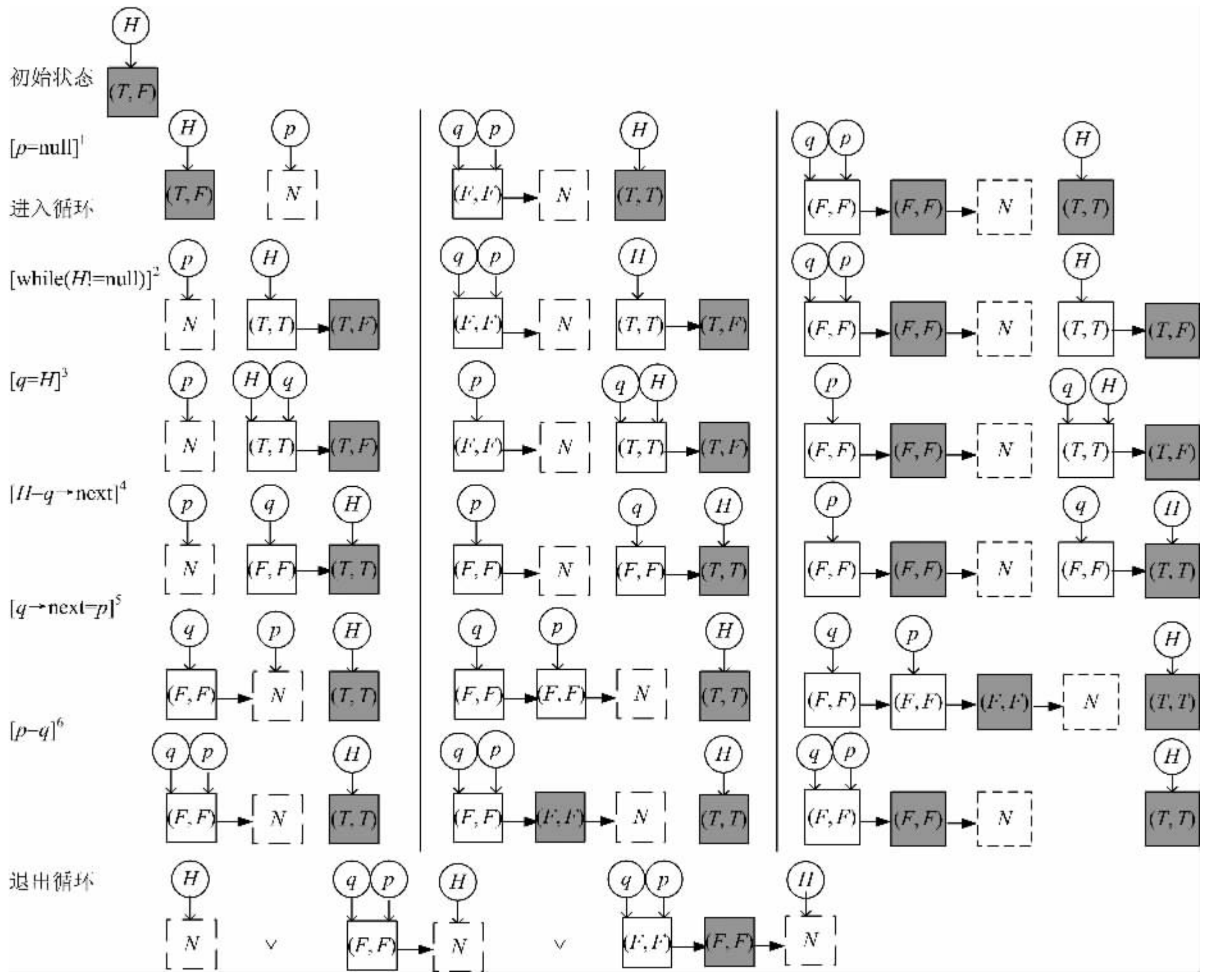

图 4　形状图推导过程

Fig. 4　Example of shape graph transformation

4　访问控制

维持分析能够获得每个共享单元的维持区间，并将每个单元维持开始和结束时的指针访问路径保存在每个标签语句的 M_S 和 M_E 中，根据这些信息编译器可以为程序添加基于锁的访问控制代码. 算法如下：

```
for([s]l∈S) do
  AS=allMS(l)
  AE=allME(l)
  /* Acquire lock operation */
  for(x∈AS) do
    for((l′, l) flow(s)) do
      add lock(x) between l′ and l
  /* Release lock operation */
  for(x∈AE) do
    for((l, l′ ) flow(s)) do
      add unlock(x) between l and l′
```

为了保证程序在各种情况下都能正确并发执行，需要将所有图形分析得到的指针路径集合合并. all$M_S(l)$和 all$M_E(l)$表示将$\mathbb{H}_{out}(l)$返回的四元组集合中对应所有 M_S 和 M_E 集合做一个合并操作，得到两个新集合 A_S 和 A_E，表示在该语句位置所有可能的加锁与解锁指针路径集合.

该算法顺序遍历每条语句$[s]^l$，并获得该位置合并后的指针访问路径集合 A_S 和 A_E，将集合中的所有访问路径作为加锁和解锁操作的参数. 加锁的位置是在该语句和其前驱语句之间，解锁的位置是在该语句和其后继语句之间. 对于存在多个入边或出边的语句$[s]^l$ 则需要在每条边都添加对应的锁操作. 图 5 是依据表 1 中维持分析结果得到的带有锁的程序代码.

lock( )和 unlock( )用来表示加锁和解锁操作，

```
1    p=null;
2    lock(H)
3    while (H! =null) {
4        q=H;
5        lock(q→next)
6        H=q→next;
7        unlock(q)
8        q→next=p;
9        p=q;
10   }
```

图5 自动加锁代码示例

Fig. 5 auto add lock and unlock for program

并且规定当作用在空单元时等于不做任何操作.程序在循环前对 H 指向的链表头节点加锁,直到语句6执行后 H 指向下一个节点解锁,并且此时指向原来节点的指针变量是 q.而语句6由于 H 需要访问链表的下一个节点,因此又添加了一条加锁语句,该节点的解锁是在下一次迭代语句6执行后.循环结束时由于 H 指向了空,因此语句5在最后一次迭代中的加锁为空操作,不会影响后面的程序.

当程序中的条件分支过多时,该算法会导致在同一个程序点上的形状图集合膨胀,给自动加锁的操作带来问题.本文下一阶段将进一步研究如何对多个形状图进行简化的方法,从而尽量减少锁操作.

5 结论

本文定义了以指针访问关系为依据的维持语义,利用形状图来描述程序中动态数据和指向关系,并给出过程内动态数据状态和共享性的分析方法.该方法可以找出动态数据被并发访问时的维持信息,然后编译器根据这些信息为数据访问进行加锁和解锁操作.利用形状图的推导方法还能检查程序中如内存泄漏等潜在问题.目前的分析只局限于过程内,不支持函数调用,形状图分析只能推导较为简单的链表结构.我们正在研究如何利用形状图推导更复杂的数据结构以及对多个形状图进行简化的方法,从而使本文的自动加访问控制的方法更加完善.

参考文献(References)

[1] Saraswat V. Report on the experimental language X10. Version 2.01, Jan. 2010 [EB/OL]. http://x10-lang.org/.

[2] Lee J K, Palsberg J. Featherweight X10: a core calculus for async-finish parallelism[C]// Proceedings of 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Bangalore, India: ACM Press, 2010: 25-36.

[3] Cray Inc. Chapel language specification 0.785 [EB/OL]. http://chapel.cs.washington.edu/.

[4] Allen E, Chase D, Hallett J, et al. The Fortress Language Specification Version1.0 [S]. Sun Microsystems, Inc., 2008.

[5] Rossbach C J, Hofmann O S, Witchel E. Is transactional programming actually easier? [C]// Proceedings of 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Bangalore, India: ACM Press, 2010: 47-56.

[6] Zhang Y, Franco J B M, Gao G R. Atomic section: concept and implementation[C]// Proceedings of Mid-Atlantic Student Workshop on Programming Languages and Systems. Newark, USA, 2005: 1-10.

[7] Cherem S, Chilimbi T, Gulwani S. Inferring locks for atomic sections[C]// Proceedings of ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation. Tucson, USA: ACM Press, 2008: 304-315.

[8] Larus J, Rajwar R. Transactional Memory. Synthesis Lectures on Computer Architecture[M]. California, USA: Morgan & Claypool Publishers, 2007.

[9] Cascaval C, Blundell C, Michael M, et al. Software transactional memory: why is it only a research toy? [J]. Queue, 2008, 6(5): 46-58.

[10] Zhang Y, Wang C, Zhang W, et al. A parallel programming language with resource holding declaration[R]. University of Science and Technology of China, 2010.

[11] Nielson F, Nielson H R, Hankin C. Principle of Program Analysis[M]. 2nd Ed, Springer, 1999.