过程

1. 令 b 为 C 之前的程序点;
2. 令 e 为 C 之后的程序点;
3. 令 F 为包含所有与 P 相关的字段的集合;
4. $M\leftarrow\{null\}$;
5. for all $stmt\in C$,如果 $stmt$ 为以下的形式 $exp\rightarrow f=rhs$,其中 $f\in F$
6. 令 i 为 $stmt$ 之前的程序点;
7. $M\leftarrow M\cup\{Simplify(EquivExp(exp,i,b))\}$;
8. end for
9. $y\leftarrow Simplify(EquivExp(y,e,b))$;
10. $Q\leftarrow ComputePartition(y,M,b,e)$;
11. return $SynPrip(y,Q,P)$;

算法 2 在已知一个数据结构(根节点为 r)的被修改节点集合 M 的前提下,递归地计算其划分.计算方法如下:首先,通过函数 *FirstReachable* 找到 M 中从 r 直接可达的节点集合.所谓直接可达的节点,指的是从 r 到目标节点之间不经过任何其他在 M 中的节点.我们将这些节点放入集合 N 中.如果 N 是空集,表明 M 中没有任何可以从 r 到达的节点,此时,以 r 为根节点数据结构的划分为$\langle\varnothing,\varnothing,\{r\}\rangle$,并且它自己为自己的子结构;如果 N 不为空,N 中的节点为分割点,我们将他们放入集合 P 中.同时,从 r 到 N 的节点满足一个片段的定义,因此形成一个片段,用二元组$\langle r,N\rangle$表示.我们将它放入集合 G 中.第 8 行~第 21 行递归地计算出子结构的划分.对于 N 中的每一个节点,我们计算出他们后继指针字段所指向的节点.从这些节点开始,我们继续计算他们的划分,并把它们加入到其父节点的划分中.

算法 2. *ComputePartition*.

输入:r:目标根节点;M:被修改节点集合;b,e:程序开始和结束时的程序点;

输出:r 的划分.

过程

1. $N\leftarrow FirstReachable(r,M)$;
2. **if** $N=\varnothing$, then
3. return $\langle\varnothing,\varnothing,\{r\}\rangle$;
4. end if
5. $P\leftarrow\varnothing$, $G\leftarrow\varnothing$, $S\leftarrow\varnothing$;
6. $P\leftarrow P\cup N$;
7. $G\leftarrow G\cup\{(r,N)\}$;
8. for each $node\in N$
9. for each $node$ 的往下指向的字段 link
10. $l\leftarrow Simplify(EquivExp(node@b\rightarrow link,e,b))$;
11. if $l\neq null$, then
12. $Q'\leftarrow ComputePartition(l,M,b,e)$;
13. $P\leftarrow P\cup Q'.P$;
14. $G\leftarrow G\cup Q'.G$;
15. $S\leftarrow S\cup Q'.S$;
16. end if
17. end for
18. end for

19. return ⟨*P*,*G*,*S*⟩;

算法 3 在第 3 行中首先找到离叶子节点最近的分割点,即,它们与叶子节点中间没有其他的分割点.接着,我们使用 *UnfoldMatch* 函数(实现了展开与匹配技术),并根据 *P* 的定义来合成由分割点所指向的数据结构的性质 *p*.这些性质存储在集合 *N* 中,并且将这些分割点从 *Q.P* 中移除,并重复以上操作.如果 *Q.P* 为空,意味着离根节点最近的分割点已经被处理,并且最终的数据结构由一个片段和若干个子结构组成(这些子结构由 *M* 中最后剩下来的节点所指向).此时,我们将可以根据 *M* 中的节点判断最后得到的 *r* 最后一步的划分 *Q*.最后,我们使用 *Compose* 函数,根据 *Q*、子结构的性质 *N* 和 *P* 的定义,计算出 *r* 的性质.如果这个过程 *Compose* 函数无法直接计算出结果,则需要提供额外的信息辅助分析过程.

算法 3. *SynProp*.

输入:*r*:新的递归数据结构的根节点;*Q*:*r* 的划分;*P*:归纳性质的定义;

输出:*r* 关于 *P* 所满足的性质.

过程

1. 令 *N* 为关于子结构的性质的集合,初始为空;
2. while $Q.P\neq\varnothing$
3. 令 *M* 为包含 *Q.P* 中与叶子节点之间不存在其他在 *Q.P* 中节点的节点集合;
4. for each $n\in M$
5. $p\leftarrow UnfoldMatch(P,n)$;
6. $N\leftarrow N\cup\{p\}$;
7. $Q.P\leftarrow Q.P\backslash\{n\}$;
8. $Q.S\leftarrow Q.S\backslash\{n\}$
9. end for
10. if $Q.P=\varnothing$,则
11. $Q.P\leftarrow M$;
12. $Q.G\leftarrow\langle r,M\rangle$;
13. $Q.S\leftarrow M$;
14. end if
15. end while
16. 返回 *Compose*(*r*,*Q*,*N*,*P*);

4 过程间分析

本节讨论函数调用对于性质的影响.这部分主要包含两个问题.

(1) 如何确定一个函数局部堆内存与全局堆内存之间的关系;

(2) 如何获取对于调用者和被调用者来说都较为精确的函数摘要.

针对第 1 个问题,我们对目标函数做一定的限制,即,只会访问其参数所指向的结构及其子结构的函数.这样的限制使得我们只需要考虑函数对于其局部堆内存的影响,而不要考虑其他部分堆内存的变化.根据我们的统计,多数递归结构的处理函数都具备这样特点,即:子过程往往只会递归地修改子结构,而不会反向地修改父结构.我们设计了算法来检查目标函数是否满足这个条件.

针对第 2 个问题,我们提出了一种性质导向的程序摘要生成和实例化技术.对于一个特定的归纳性质,我们在现有技术[20]的基础上定制一个结合了自下向上和自上向下两种分析模式的过程来精化函数摘要.这个过程较为灵活和精确,能够以较低的计算复杂度得到可重性较高的程序摘要.

4.1 过程间分析的总体过程

分析框架结合了自上而下和自下而上两个分析过程.首先,根据函数的调用关系图,优先分析具有前置条件

的函数.遇到函数调用时,如果遇到被调用的函数已经分析完毕,或者有用户提供的函数摘要时,直接实例化.如果被调用函数没有可用的摘要,框架会暂停当前的分析过程,创建一个新的任务先分析被调用函数.根据函数的调用关系会形成一个任务栈,如果栈顶的任务无法进行,说明缺乏必要的前置条件或无法实例化摘要,整个任务栈将会被终止.

如果自上而下的过程无法完全实现自动化,需要用户为关键函数提供前置条件.特别的,对于不满足两类性质变化模式的情况,需要由用户提供其函数摘要.这个过程中,我们的分析框架可以对目标函数进行自下而上的分析,提供相关的信息给用户,帮助其准确给出必要的前置条件或函数摘要.

4.2 过程局部堆内存

在考虑函数调用对堆内存所产生的影响时,很重要的一点是确定一次函数调用能够访问到哪些内存.一个被调用的函数能访问到的堆内存被称为函数局部堆内存,在进行过程间分析时,需要确定对函数局部堆内存的修改给全局堆内存带来的副作用.

在我们的分析方法中,我们重点关注函数是不是只会修改其实际参数指向的那部分子结构的堆内存.为了更好地理解这一点,我们以图 4 中的程序为例.

函数 f 是处理二叉树的递归函数,其中,*bt* 是至少包含 3 个字段的结构体:*left* 表示左分支,*right* 表示右分枝,*parent* 表示父节点.为了简单起见,没有显示数据字段.这个函数根据不同的条件处理 3 类情况.如果图 4 中第 1 个条件满足,则处理左分支;如果第 2 个条件满足,则处理右分支.这是典型的递归函数的模式.但是当前两个条件都不满足时,函数处理其父节点.

这种情况对于程序分析来说是不愿意见到的,因为函数 f 的内存踪迹变得难以准确捕获.例如,我们将一个二叉树 x 的子树 v 传入函数 f,那么整个 x 中的所有节点都有可能被 f 访问到.因此,我们必须限制我们处理的函数类型.简单地讲,我们只处理那种按照自上而下方式处理递归数据结构的函数:一个函数中所调用的函数必须只能访问这个函数参数所指向结构的子结构,而不能反向地访问其父节点.

```
void f(bt* t)
Requires: bt(t)
Ensures: bt(t)
{
    if (...)
    {
        f(t→left);     //no problem
    }
    else if (...)
    {
        f(t→right);    //no problem
    }
    else
    {
        f(t→parent);   //in trouble
    }
}
```

Fig.4 A recursive function that is hard to analyze and reason automatically

图 4 一个难以自动化分析和验证的递归函数

4.2.1 函数内存踪迹检查

对于一个具体的性质 P,可以根据其定义进行自动化检查函数 f 是否满足这个限制.首先,根据递归性质的定义判断它依赖于哪些字段;然后分析函数中是否访问到除这些字段之外的指向当前节点类型的指针字段,非指针字段以及指向其他类型的指针字段不受影响.如果 f 中包含函数调用,则还需要检查被调用的函数是否满足此限制.这个整个检查过程以递归的方式进行下去,直到函数内不再包含其他函数调用.如果函数 f 通过检查,则 f 可以被我们的方法进行分析.

4.3 调用上下文敏感的程序摘要生成

现有的部分工作[20]尝试解决类似的问题,以在计算复杂度和可重性方面达到一个平衡.基本过程如下:首先进行一次自上而下的分析,以得到被调用函数输入参数的抽象状态(即输入参数所满足的规约).当抽象状态数超过一定的阈值时,说明被调用者要处理的情况较多,触发一次自下而上的分析.这个过程中,自上而下的分析结果就可以为自下而上的分析决定为被调用者生成什么样模板的性质.这些模板同时也指出了需要分析被调用者在哪些特定的前置条件下所满足的后置条件,从而提高分析的效率和精度.我们在这个工作的基础上,以上下文敏感的方式为递归数据结构计算的以性质为导向的程序摘要.

我们优先分析具有前置条件的函数.这也意味着我们的过程间分析总体上来说是自上而下的.因为自上而下的分析可以使我们获取被调用者的上下文信息,从而减少对于用户人工提供的前置条件的要求.具体来说,根据当前的分析结果可以得知,在函数调用前成立的性质,根据这些性质可以推断出被调用函数的前置条件.根据这些前置条件我们继续分析被调用者的函数摘要,直到计算得到其函数摘要或者其函数摘要已经存在.得到被调用者的函数摘要后,我们根据调用上下文将函数摘要进行实例化.如果这个过程中包含递归的,我们会假设前置条件中成立的性质在后置条件中依然成立,作为其函数摘要.然后根据这个函数摘要计算其后置条件,如果计算得出的后置条件与假设的后置条件不冲突,那么将这两个后置条件的并集作为其摘要.

从调用者和被调用者的角度考虑程序摘要的结果往往是不同的:一方面,自上而下(从调用者到被调用者)的分析过程只考虑被调用者可能出现的调用上下文,而忽略哪些不可能成立的上下文,因而会得到不太全面的结果;另一方面,自下而上的分析(先分析被调用者再分析调用者)往往需要考虑被调用可能需要处理的所有情况,或者说只会给出程序运行需要满足的最弱前置条件,而无法考虑到被调用时所处的上下文,因而难以实例化.对比来讲,前者计算起来需要更小的开销但是由于缺乏通用性所以难以被重用,而后者容易被重用但是需要大量的计算.我们通过下面这个例子来说明两者之间的区别.

例 3:图 5 中的程序 *leftRotate* 将其参数 *x* 指向的有序二叉树向左进行旋转,并且在程序结束时返回新的二叉树(用 *y* 指向).

```
Node *leftRotate(Node *x)
Requires: bt(x)
Ensures: bt(x)
{
    if (x==null)
        return x;
    Node *y=x→right;
    if (y!=null) {
        Node *T2=y→left;
        y→left=x;
        x→right=T2;
    }
    return y;
}
```

Fig.5 A program that rotates a binary tree to the left

图 5 将二叉树进行左旋的函数

一方面,我们以分析程序结束时 *y* 的两颗子树的高度差这个性质为例,从函数 *leftRotate* 本身的角度来看,它的前置条件仅仅要求 *x* 是一棵二叉树节点,传入的二叉树 *x* 可以是各种形状,甚至可以是 *null*.此时,想通过分析得到所有情况下 *y* 的两棵子树的高度差就较为复杂.

但如果考虑到函数 *leftRotate* 实际被调用时的上下文(即图 6 中的程序 *insert*),传入的二叉树的左右子树高度差只有两种可能性存在.在 *insert* 程序中,函数调用 *getBalance*(*node*)用于获取 *node* 的左子树与右子树的高度差.在函数 *insert* 中,*leftRotate* 被调用了 3 次.其中:第 1 次和第 3 次调用时,传入的二叉树形状属于表格 1 列出的第 2 类情况,第 2 次调用时传入的二叉树形状属于第 2 类情况.这取决于 *insert* 程序中的几个 if 条件判断以及另外一个函数 *rightRotate* 的影响.

```
Node* insert(Node* node,int key)
Requires: bt(node),−2<=height(node→left)-height(node→right)<=2
Ensures: bt(node),−2<=height(node→left)−height(node→right)<=2
{
    if (node==NULL)
        return newNode(key);
    if (key<node→key)
        node→left=insert(node→left,key);
    else
        node→right=insert(node→right,key);
    int balance=getBalance(node);
    if (balance>1 && key<node→left→key)
        return rightRotate(node);
    if (balance<−1 && key>node→right→key)
        return leftRotate(node);                    //第1次调用
    if (balance>1 && key>node→left→key) {
        node→left=leftRotate(node→left);            //第2次调用
        return rightRotate(node);
    }
    if (balance<−1 && key<node→right→key) {
        node→right=rightRotate(node→right);
        return leftRotate(node);                    //第3次调用
    }
    return node;
}
```

Fig.6 A program that inserts a node while ensuring its balance

图 6 向二叉树中插入节点并使之保持平衡的函数

表 1 中列出了两种在程序开始时 *x* 的高度与程序结束时 *y* 的高度之间的对应关系,其中,函数 *height* 为表示二叉树高度的递归函数.有两种情况.

1) 如果程序开始时 *x* 的右子树的高度比其左子树的高度多 1,那么在程序结束时 *y* 的左子树的高度比其右子树的高度多 1;
2) 如果程序开始时 *x* 的右子树的高度比其左子树的高度多 2,并且 *x* 的右子树的右子树的高度比 *x* 的右子树的左子树的高度多 1,那么在程序结束时 *y* 的左子树的高度和其右子树的高度相同.

Table 1 Program abstract of function *leftRotate* w.r.t. the property *height*

表 1 函数 *leftRotate* 关于性质 *height* 的函数摘要

程序开始点	程序结束点
$height(x\rightarrow right)-height(x\rightarrow left)=1$	$height(y\rightarrow left)-height(y\rightarrow right)=1$
$height(x\rightarrow right)-height(x\rightarrow left)=2\wedge height(x\rightarrow right\rightarrow right)-height(x\rightarrow right\rightarrow left)=1$	$height(y\rightarrow left)=height(y\rightarrow right)$

如果我们不知道函数 *leftRotate* 所处的调用上下文,我们就难以分析出调用者 *insert* 所需要的摘要,也就难以将这种摘要实例化成证明函数 *insert* 时所需的的重要断言.值得指出的是,这个函数摘要难以自动化获取的主要原因在于性质 *height* 的变化不满足前文提到的两种模式.此时,需要用户给出精确的函数摘要供分析框架进行实例化.

另一方面,如果对于性质 *bt* 来说,由于其变化满足前文提到的两种模式,因此关于它的函数摘要就容易自动分析出来.首先,我们可以通过计算得出函数 *leftRotate* 每次被调用之前的上下文,通过合并之后可以得出其前置条件为 $bt(x)$,即,参数 *x* 指向一个二叉树.根据这个前置条件,我们对函数 *leftRotate* 进行分析,由于性质 *bt* 的变化符合两类变化模式,所以可以得出其后置条件为 $bt(y)$,即,返回值 *y* 指向一个二叉树.这就是函数 *leftRotate* 关于性质 *bt* 的函数摘要.

5 案例分析

二叉树是一种常用的数据结构,尤其是红黑树被多个库作为 Set 和 Map 的具体实现,如 Java 中的 TreeSet

和 TreeMap 以及 C++ STL 中的 std::set 和 std::map 等.

我们关注二叉树的以下几个性质,分别是节点集合(*nsbt*)、二叉树的谓词(*isbt*)和树的高度(*height*),它们的定义见表 2.其中,二叉树片段的节点集合(*nsbtseg*)和二叉树片段的谓词(*isbtseg*)为辅助性的定义,分别用来辅助分析节点集合和谓词两个性质.

Table 2 Some Inductive properties of binary search trees

表 2 二叉树中部分归纳性质的定义

性质	定义
nsbt(节点集合)	$nsbt(x)\triangleq(x=null)?\varnothing:nsbt(x\to l)\cup\{x\}\cup nsbt(x\to r)$
nsbtseg(片段的节点集合)	$nsbtseg(x,\{y\})\triangleq(x=null\vee x=y)?\varnothing:nsbt(x\to l,\{y\})\cup\{x\}\cup nsbt(x\to r,\{y\})$
isbt(二叉树)	$isbt(x)\triangleq(x=null)?\text{true}:isbt(x\to l)\wedge isbt(x\to r)\wedge nsbt(x\to l)\cap\{x\}=\varnothing\wedge nsbt(x\to r)\cap\{x\}=\varnothing\wedge nsbt(x\to l)\cap nsbt(x\to r)=\varnothing$
isbtseg(二叉树片段)	$isbtseg(x,\{y\})\triangleq(x=null\vee x=y)?:\text{true}:isbtseg(x\to l,\{y\})\wedge isbtseg(x\to r,\{y\})\wedge nsbtseg(x\to l,\{y\})\cap\{x\}=\varnothing\wedge nsbtseg(x\to r,\{y\})\cap\{x\}=\varnothing\wedge nsbtseg(x\to l,\{y\})\cap nsbtseg(x\to r,\{y\})=\varnothing$
height(树的高度)	$height(x)\triangleq(x=null)?0:\max(height(x\to l),height(x\to r))+1$

可以发现:节点集合这个性质满足我们描述的第 2 类的要求,即,局部的变化可以反映到整体的变化.如果左子树的节点集合增加了一个元素 *y*,并且 *y* 与当前节点 *x* 不同、同时,*y* 不属于右子树中的节点集合时,我们可以判断整体的节点集合也增加了一个元素 *y*.另外,谓词性质和高度性质并不满足第 2 类的要求.当一个节点的左子树或右子树不是二叉树时,我们只能说这个节点所代表的数据结构并不是一棵二叉树,并不能得出额外有用的性质.而当树的某个子树的高度增加或者减少时,我们也并不能断言整个子树的高度变化了多少.

旋转是处理所有类型的平衡二叉树中的一个关键操作.我们以左旋为例.图 7(a)为一个完整的左旋函数的代码.在这个例子中,*x* 是需要旋转的目标节点.如果 *x* 为 *null*,则直接返回 *null*.如果 *x* 不为 *null*,且 *x* 的右子节点($x\to right$)不为空,则进行旋转,否则直接返回 *x*.程序中一共有两个分支和 3 种基本情况:第 1 种情况是当输入的参数 *x* 为 *null* 时,直接返回 *null*,如图 7(b)所示;第 2 种情况是当 *x* 不为 *null* 但是 $x\to right$ 为 *null* 时,此时直接返回 *x*,如图 7(c)所示;最后一种情况是当 *x* 不为 *null* 并且 $x\to right$ 也不为 *null* 时,可以顺利完成一次旋转,并返回新树的根节点 *y*,如图 7(d)所示.因此,我们可以将 *leftRotate* 函数分解为如图 7 中所示的 3 种不同前置条件下的顺序程序.

我们以二叉树节点集合性质 *nsbt* 为例,它是满足我们将的第 2 种分类的,即,局部的变化可以映射到全局的变化.首先使用分割与拼接技术对图 7(d)进行分析.我们首先能得到程序运行结束时二叉树 *y* 的一个划分.

$$\langle\{y,y\to left\},\{\langle y,\{y\}\rangle,\langle y\to left,\{y\to left\}\rangle\},\{y\to left\to left,y\to left\to right,y\to right\}\rangle.$$

在这个划分的基础上,我可以计算出 *y* 的节点集合是由 5 个互补相交的集合的并集组成的,即:

$$\{y\}\cup\{y\to left\}\cup nsbt(y\to left\to left)\cup nsbt(y\to left\to right)\cup nsbt(y\to right).$$

如果用最弱前置条件的计算方法来计算,我们可以知道以下这个节点集合在程序开始前的等价表达式为

$$\{x\to right\}\cup\{x\}\cup nsbt(x\to left)\cup nsbt(x\to right\to left)\cup nsbt(x\to right\to right).$$

而这完全等价于程序开始前 *x* 的节点集合.因此,我们就可以知道程序结束时 *y* 的节点集合等于程序开始前 *x* 的节点集合.无论 *leftRotate* 函数在哪里被调用,它总会返回一个节点集合性质没有发生改变的二叉树,比如在图 6 中的 *insert* 函数中调用多次,其参数和返回值都是 *node* 这个二叉树的子结构,而子结构节点集合这个性质没有改变这个事实就决定了整个 *node* 二叉树的节点集合也没有发生改变.这个过程考虑到了 *leftRotate* 函数在 *insert* 函数中被调用的上下文,从而为我们就可以为 *leftRotate* 函数生成一个利于实例化的程序摘要(同时也是后置条件):$nsbt(y)=nsbt(x)@0$.此处的@0 表示的是在程序开始处的表达式,这个表达式的具体用法见文献[1].

```
Node *leftRotate(Node *x)
{
    if (x==null)
        return x;
    Node *y=x→right;
    if (y!=null) {
        Node *T2=y→left;
        y→left=x;
        x→right=T2;
        return y;
    } else
        return x;
}
```

(a) 源程序

```
Node *leftRotate(Node *x)
{
    return x;
}
```

(b) 前置条件:$x=null$

```
Node *leftRotate(Node *x)
{
    Node *y=x→right;
    return x;
}
```

(c) 前置条件:$x\neq null \wedge right=null$

```
Node *leftRotate(Node *x)
{
    Node *y=x→right;
    Node *T2=y→left;
    y→left=x;
    x→right=T2;
    return y;
}
```

(d) 前置条件:$x\neq null \wedge right\neq null$

Fig.7 Decomposition of the function *leftRotate*

图 7 *leftRotate* 函数的分解

6 实现和实验结果

我们将本文中提出的方法实现为验证工具 Accumulator 中的一个模块(更多技术细节请参考工具的主页面 http://seg.nju.edu.cn/scl.html).这个工具使用 ANTLR 作为前端解析工具,接收一个 C 语言的子集(不支持 union 和指针算术运算等特性),使用 Z3 作为后端的公式求解器.它支持霍尔式的程序证明过程,并带有若干个自动化和半自动化的分析技术,如数据流分析框架和最弱前置条件计算等.

给定程序、程序的前置条件以及待验证性质的归纳定义,我们能给出典型递归数据结构上特定操作完成后的关于待验证性质的公式.这些公式反映了这些操作的语义信息,并且可以被作为验证条件(verification conditions),计算其最弱前置条件,然后使用 Z3 求解器确定它们是否可以被程序的前置条件推导得出.如果能够被推导得出,就可以作为程序的后置条件.实验结果见表 3.

Table 3 Operations and properties of typical recursive data structures

表 3 典型数据结构上的操作及性质

数据结构	操作	性质	时间
单链表	*search, nsert, append, prepend, delete, rotate, reverse, delete_insert, concat, delete_all*	*nsslist, isslist, length*	<1s
双链表	*search, insert, append, prepend, delete, rotate, reverse, swap, delete_insert, delete_all, concat*	*nslist, islist, length*	<1s
二叉查找树	*search, leftmost, insert, delete, rotate*	*nsbt, isbt, hight*	<3s

可以看到:我们的方法对单链表和双链表在多个典型操作的 3 个性质(节点集合、谓词和长度)在 1s 内完成分析,对于二叉查找树在多个典型操作下的 3 个性质(节点集合、谓词和高度)可以在 3s 内完成分析.分析的结果以断言的形式在我们的工具中完成了这些性质的证明,达到了我们预期的效果.对于二叉查找树的分析需要较多的时间,主要在于函数调用所带来的自上而下和自上而下两次分析的开销.而单链表中的程序的函数调用较少,因此效率较高.

7 总结和相关工作

本文提出了一种自动化分析递归数据结构归纳性质的框架.工作的主要目标是为霍尔式的程序证明自动生成关于具体性质的程序语义摘要的断言,从而方便程序员去证明程序的相关性质.我们从 3 个方面进行研究.

1) 根据对递归数据结构上归纳性质的变化模式进行了总结,将其分为两个容易自动化处理的类别,使之在后续过程中简化分析流程;
2) 提出了一种叫做分割与拼接的过程内分析技术,它在经典的展开与匹配技术的基础上扩展了其应用场景,使之能处理更多程序;
3) 提出了一种调用上下文敏感的过程间分析技术来为程序生成容易被调用者实例化的函数摘要,主要解决传统的自下而上的过程间分析方法中函数摘要难以直接被实例化的问题.

我们提出的方法中的算法都是直接用于顺序程序的,这主要是基于霍尔式程序证明的基本过程.对于带有分支、循环和函数调用的程序,我们分别使用对应的预处理进行处理,从而得到方法能够直接处理的顺序程序.对于才有递归的程序,首先找出递归的环,将其断开(即忽略其中一个函数的调用效果),然后迭代地求整个环中的不动点.另外,函数摘要的生成和实例化过程是相互作用和相互利用结果的.

案例分析和实验结果表明,我们的方法对于单链表、双链表和二叉树的常见操作的 3 个归纳性质的分析结果还是满足预期的.我们未来还将对更多较为复杂的递归树结构、操作以及其更为复杂性质进行分析.

相关工作方面,我们使用 Scope Logic[1]来描述我们分析的结果,同时,在第 3 节在描述分析算法时用到的相关函数(如 *Simplify* 和 *EquivExp*)也参考文献[1].另外,我们的方法还在文献[11,12]提出的展开与匹配技术的基础上进行扩展,使其适应具体问题.我们还利用了文献[19,20]中自上而下与自上而下结合的过程间分析算法,并进行了适合具体问题的定制(如针对归纳性质的实例化过程).

利用程序分析技术为程序验证做好准备工作,是相关领域的热点问题.文献[5–7]使用形状分析来分析堆内存中的数据结构的形状信息,为程序在关键点上生成合适的断言,这些断言起到承上启下的作用:它们既作为之前程序片段的后置条件,又作为之后程序片段的前置条件.这些断言帮助将整个证明过程串联起来,而难点恰恰在于生成的断言既不能太强(否则难以被整个程序的前置条件推导得出),又不能太弱(否则难以推出整个程序的后置条件).抽象解释[8]是另一种经典且被采用较多用来分析与处理堆内存的理论.文献[26]使用抽象解释的分析结果来自动地构造霍尔式程序证明过程.文献[10]基于符号执行的技术获取程序的相关性质,并使用分离逻辑[2]来描述这些性质.文献[11,12]提出了自然证明(natural proof)技术证明带有指针和递归数据结构的程序,其关键在于将人们在平时证明中用到的一些技巧应用到自动化过程中.文献[15]中利用最强后置条件达成一定程度的自动化,与 Dijkstra 提出的最弱前置条件[27]颇为相似.文献[25]对多年来的相关工作进行了综述性介绍.

文献[16]中的工作的主要优点在于能够处理较多类型的数据结构,包括图和哈希表.为此,其中使用了高阶逻辑来提高表达能力.但是,他们进行证明的过程较为复杂,使用了特别的决策过程和子表达式替换等.我们的工作则聚焦递归树结构本身,因此更容易被学习和使用.文献[17]关注程序中的可达性性质和模块化验证问题,并且通过加入对于堆中的别名和路径等的限制达到完备性的目标.我们的工作与之不同点在于,我们关注于特定的分析技术而非对于程序完整的功能正确性验证.我们认为:对于复杂程序的证明,需要依赖于多种分析和验证技术的综合使用和相互补充.

文献[21]中的工作与我们的非常类似,也关注性质导向的分析.他们的方法基于谓词解释并且尝试证明程序没有违反内存安全性的问题以及递归数据结构谓词性质的满足.我们的方法能够处理一般的归纳性质,不仅仅局限于谓词性质.另外,还能够给出如果一个性质发生变化,以及变化的形式是如何的.文献[22,23]中切点(cutpoint)的概念用来描述局部堆内存与全局堆内存之间的关系,与我们进行内存踪迹检查的方法是类似的.文献[23]中也限制了对于非子结构部分内存的访问,从而在减少计算量的同时确保分析结果的正确性.但他们提出的方法比我们更为形式化.我们方法的优点在于直观而有效.

References:

[1] Zhao J, Li X. Scope logic: An extension to hoare logic for pointers and recursive data structures. In: Proc. of the Theoretical Aspects of Computing (ICTAC 2013). Berlin, Heidelberg: Springer-Verlag, 2013. 409−426. [doi: 10.1007/978-3-642-39718-9_24]

[2] Reynolds JC. Separation logic: A logic for shared mutable data structures. In: Proc. of the 17th Annual IEEE Symp. on Logic in Computer Science. IEEE, 2002. 55−74. [doi: 10.1109/LICS.2002.1029817]

[3] Sagiv M, Reps T, Wilhelm R. Parametric shape analysis via 3-valued logic. ACM Trans. on Programming Languages and Systems (TOPLAS), 2002,24(3):217−298. [doi: 10.1145/514188.514190]

[4] Deutsch A. Interprocedural may-alias analysis for pointers: Beyond *k*-limiting. ACM SIGPLAN Notices, 1994,29(6):230−241. [doi: 10.1145/773473.178263]

[5] Kreiker J, Seidl H, Vojdani V. Shape analysis of low-level C with overlapping structures. In: Proc. of the Verification, Model Checking, and Abstract Interpretation. Berlin, Heidelberg: Springer-Verlag, 2010. 214−230. [doi: 10.1007/978-3-642-11319-2_17]

[6] Berdine J, Calcagno C, Cook B, *et al*. Shape analysis for composite data structures. In: Proc. of the Computer Aided Verification. Berlin, Heidelberg: Springer-Verlag, 2007. 178−192. [doi: 10.1007/978-3-540-73368-3_22]

[7] Holik L, Lengal O, Rogalewicz A, *et al*. Fully automated shape analysis based on forest automata. In: Proc. of the Computer Aided Verification. Berlin, Heidelberg: Springer-Verlag, 2013. 740−755. [doi: 10.1007/978-3-642-39799-8_52]

[8] Cousot P, Cousot R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of the 4th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages. ACM Press, 1977. 238−252. [doi: 10.1145/512950.512973]

[9] Bouajjani A, Dragoi C, Enea C, *et al*. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In: Proc. of the Verification, Model Checking, and Abstract Interpretation. Berlin, Heidelberg: Springer-Verlag, 2012. 1−22. [doi: 10.1007/978-3-642-27940-9_1]

[10] Berdine J, Calcagno C, O'hearn PW. Symbolic execution with separation logic. In: Proc. of the Programming Languages and Systems. Berlin, Heidelberg: Springer-Verlag, 2005. 52−68. [doi: 10.1007/11575467_5]

[11] Pek E, Qiu X, Madhusudan P. Natural proofs for data structure manipulation in C using separation logic. In: Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation. ACM Press, 2014. 46. [doi: 10.1145/2594291.2594325]

[12] Qiu X, Garg P, Stefanescu A, *et al*. Natural proofs for structure, data, and separation. ACM SIGPLAN Notices, 2013,48(6): 231−242. [doi: 10.1145/2491956.2462169]

[13] Chin WN, David C, Nguyen HH, *et al*. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Science of Computer Programming, 2012,77(9):1006−1036. [doi: 10.1016/j.scico.2010.07.004]

[14] Lee O, Yang H, Yi K. Automatic verification of pointer programs using grammar-based shape analysis. In: Proc. of the Programming Languages and Systems. Berlin, Heidelberg: Springer-Verlag, 2005. 124−140. [doi: 10.1007/978-3-540-31987-0_10]

[15] de Boer F, Bonsangue M, Rot J. Automated verification of recursive programs with pointers. In: Proc. of the Automated Reasoning. Berlin, Heidelberg: Springer-Verlag, 2012. 149−163. [doi: 10.1007/978-3-642-31365-3_14]

[16] Zee K, Kuncak V, Rinard M. Full functional verification of linked data structures. ACM SIGPLAN Notices, 2008,43(6):349−361. [doi: 10.1145/1375581.1375624]

[17] Itzhaky S, Banerjee A, Immerman N, Lahav O, Nanevski A, Sagiv M. Modular reasoning about heap paths via effectively propositional formulas. In: Proc. of the 41st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. ACM Press, 2014. 385−396. [doi: 10.1145/2535838.2535854]

[18] De Moura L, Bjørner N. Z3: An efficient SMT solver. In: Proc. of the Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer-Verlag, 2008. 337−340. [doi: 10.1007/978-3-540-78800-3_24]

[19] Zhang X, Mangal R, Naik M, Yang H. Hybrid top-down and bottom-up interprocedural analysis. ACM SIGPLAN Notices, 2014, 49(6):249−258. [doi: 10.1145/2594291.2594328]

[20] Nystrom E, Kim HS, Hwu WM. Bottom-Up and top-down context-sensitive summary-based pointer analysis. In: Proc. of the Static Analysis. 2004. 165−180. [doi: 10.1007/978-3-540-27864-1_14]

[21] Itzhaky S, Bjørner N, Reps TW, Sagiv M, Thakur AV. Property-Directed shape analysis. In: Proc. of the CAV. 2014. 35−51. [doi: 10.1007/978-3-319-08867-9_3]

[22] Rinetzky N, Bauer J, Reps T, Sagiv M, Wilhelm R. A semantics for procedure local heaps and its abstractions. ACM SIGPLAN Notices, 2005,40(1):296−309. [doi: 10.1145/1047659.1040330]

[23] Rinetzky N, Sagiv M, Yahav E. Interprocedural shape analysis for cutpoint-free programs. In: Proc. of the SAS. 2005. 284−302. [doi: 10.1007/11547662_20]

[24] Hoare CA. An axiomatic basis for computer programming. Communications of the ACM, 1969,12(10):576−580. [doi: 10.1145/1562764.1562779]

[25] Hoare T. The verifying compiler: A grand challenge for computing research. Journal of the ACM (JACM), 2003,50(1):63−69. [doi: 10.1007/978-3-540-45213-3_4]

[26] Seo S, Yang H, Yi K. Automatic construction of Hoare proofs from abstract interpretation results. In: Proc. of the Programming Languages and Systems. 2003. 230−245. [doi: 10.1007/978-3-540-40018-9_16]

[27] Dijkstra EW. Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM, 1975,18(8): 453−457. [doi: 10.1007/978-1-4612-6315-9_14]

[28] Tang Z, Wang H, Li B, Zhai J, Zhao JH, Li X. Node-Set analysis for linked recursive data structures. In: Proc. of the 2015 IEEE Int'l Conf. on Software Quality, Reliability and Security (QRS). IEEE, 2015. 59−64. [doi: 10.1109/QRS.2015.19]

[29] Kjolstad F, Dig D, Acevedo G, Snir M. Transformation for class immutability. In: Proc. of the 33rd Int'l Conf. on Software Engineering. ACM Press, 2011. 61−70. [doi: 10.1145/1985793.1985803]

汤震浩(1989－),男,江苏南通人,博士生,主要研究领域为软件工程,程序分析,程序验证.

李彬(1988－),男,博士生,主要研究领域为软件工程,程序分析,程序验证.

翟娟(1988－),女,博士,CCF 专业会员,主要研究领域为软件工程,程序分析,程序验证,程序合成.

赵建华(1971－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为形式化方法,软件工程,程序设计语言.