软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2017,28(4):786−803 [doi: 10.13328/j.cnki.jos.005195]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

通过抽象程序证明复杂具体程序*

李 彬, 汤震浩, 翟 娟, 赵建华

(计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)

通讯作者: 赵建华, E-mail: zhaojh@nju.edu.cn

摘 要: 描述了证明抽象程序和具体程序满足一致性关系的方法.抽象程序使用抽象数据结构(ADTs),如 set,list,map 及其上的操作.具体程序使用类 C 语言中的类型.抽象程序和具体程序一致性证明需要用户给出抽象变量和具体变量的关系、抽象程序程序点和具体程序程序点的对应关系.基于对应关系,抽象程序和具体程序一致性证明可以分解,从而容易并可能自动证明.

关键词: 程序证明;一致性;抽象程序;精化;分解

中图法分类号: TP311

中文引用格式: 李彬,汤震浩,翟娟,赵建华.通过抽象程序证明复杂具体程序.软件学报,2017,28(4):786−803. http://www.jos.org.cn/1000-9825/5195.htm

英文引用格式: Li B, Tang ZH, Zhai J, Zhao JH. Verification of concrete programs with respect to abstract programs. Ruan Jian Xue Bao/ Journal of Software, 2017,28(4):786−803 (in Chinese). http://www.jos.org.cn/1000-9825/5195.htm

Verification of Concrete Programs with Respect to Abstract Programs

LI Bin, TANG Zhen-Hao, ZHAI Juan, ZHAO Jian-Hua

(State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

Abstract: This paper presents an approach to prove that a concrete program correctly implements its corresponding abstract program. Here, an abstract program uses some abstract data types such as set, list and map, and abstract operations upon those data types. A concrete program uses the types in the C-like language. The approach presented in the paper requires to specify correspondences between the abstract program and the concrete program, including correspondences between program points and correspondences between variables. Based on the correspondences, the verification task can be divided into small subtasks that can be easily and mostly automatically verified.

Key words: program verification; consistency; abstract program; refinement; decomposition

复杂的程序常常含有指针以及递归数据结构,这类程序的证明是困难的.即使提供了断言(assertion)和不变式(invariant),目前的工作,如文献[1−3],仍然难以验证动态操纵堆的性质.验证工具如 VeriFast[1]和 Bedrock[2]需要用户写低层次的辅助定理(low-level lemmas)和证明策略(proof tactics)去指导验证.它们要求用户熟悉程序(code)和规约语言(specification),需要用户提供合适的不变式(invariant)——这是非常困难的工作.

为了简化证明复杂程序的难度,本文提出了利用抽象程序辅助证明具体程序的方法.抽象程序是指使用抽象数据结构(ADTs),如 set,list,map 及其上的操作的程序.具体程序即真正的程序,比如 C 语言程序.利用抽象程

* 基金项目: 国家自然科学基金(61632015, 61561146394); 国家重点基础研究发展计划(973)(2016YFB1000802)

Foundation item: National Natural Science Foundation of China (61632015, 61561146394); National Basic Research Program of China (973) (2016YFB1000802)

收稿时间: 2016-06-20; 修改时间: 2016-09-08, 2016-11-26; 采用时间: 2016-12-04; jos 在线出版时间: 2017-01-24

CNKI 网络优先出版: 2017-02-20 13:43:27, http://www.cnki.net/kcms/detail/11.2560.TP.20170220.1343.002.html

序,本文的方法将复杂具体程序的证明分成两步:首先证明抽象程序满足规约(本文不关注抽象程序的证明);然后证明具体程序和抽象程序满足一致性.这种分层次证明可以简化证明难度,这体现在:

(1) 抽象程序的证明比较容易.抽象程序使用抽象数据类型描述程序,从而抽离了实现细节,比如指针等;

(2) 一些复杂经典的算法(如图算法)在书中使用抽象程序的形式描述,它们已被严格地验证了正确性.程序员只需实现这些算法,然后证明具体实现和抽象算法满足一致性即可;

(3) 在软件开发过程中,程序员需要将高层的设计(high-level design)转化为底层实现,程序员常常使用抽象程序理解和描述算法,程序员有必要证明底层实现符合高层的设计.有时候根据不同的环境,具体程序会有多个实现版本,分层证明之后,程序员只需要证明 1 次抽象程序,然后证明多个版本的具体程序和抽象程序满足一致性,从而减少证明负担;

(4) 本文将展示证明抽象程序和具体程序满足一致性关系是相对简单的.通常,抽象程序和具体程序的一致性证明可以被分解,分解后往往变成了对抽象数据类型的操作,从而可能实现自动证明.

本文主要关注抽象程序和具体程序的一致性关系证明.图 1 显示了证明抽象程序与具体程序一致性关系的框架.

- 首先,需要用户给出抽象程序和具体程序,然后给出抽象程序变量和具体程序变量的等式关系以及抽象程序程序点和具体程序程序点的对应关系;
- 之后验证义务生成器会在具体程序程序点上生成验证义务;
- 最终,通过程序证明工具证明具体程序上的验证义务是否成立:如果成立,那么具体程序和抽象程序满足一致性关系;否则,无法确定.

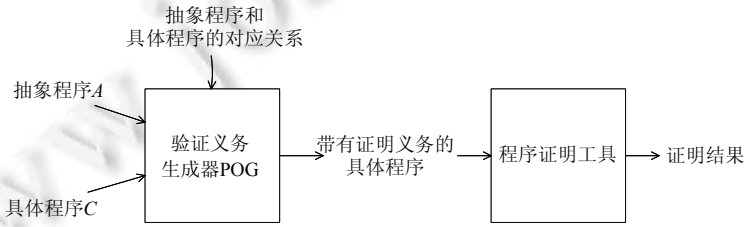


Fig.1 Procedure of our approach

图 1 方法过程

1 一个例子

考虑证明图 2 所示的程序(*Ci* 表示程序点).

```
typedef struct {Node*link; int D;} Node;
 Node*second;
Node*first;
 Node*tmp;
C1:SL_IsSList(first)
 second=null ;
C2:
while (first!=null) {
  C3:
   tmp=first->link;
   first->link=second;
   second=first;
   first=tmp;
   C4:
}
C5:SL_DataSet(second)=SL_DataSet(first)@C1
```

Fig.2 Reversing list

图 2 翻转链表

程序点 $C1$ 给出前置条件,$C5$ 给出后置条件.$SL_IsSList(first)$表示 $first$ 指向一个单链表.$SL\_IsSList$ 定义如下:

$$SL\_IsSList(x{:}\boldsymbol{P}(\boldsymbol{Node})){:}\boldsymbol{bool}\triangleq x=NULL?\text{true}:SL\_IsSList(x\to Link).$$

$SL_DataList(first)$从 $first$ 指向的单链表中生成数据域 D 的 set.$SL\_DataSet$ 定义如下:

$$SL\_DataSet(x{:}\boldsymbol{P}(\boldsymbol{Node})){:}\boldsymbol{Set}(\boldsymbol{int})\triangleq x=NULL?\{\}:x\to D+SL\_DataSet(x\to Link).$$

在程序点 $C5$,$SL\_DataSet(second)=SL\_DataSet(first)@C1$ 表示 $second$ 指向的数据域的集合等于程序点 $C1$ 处 $first$ 指向的数据域的集合.

翻转链表程序中存在循环,为了证明图 2 中的性质,验证工具需要用户提供合适的循环不变式,但这一工作往往是困难的.在这个证明中,需要下面的不变式.

$$\begin{aligned}&SL\_DataSet(first)+SL\_DataSet(second)=SL\_DataSet(first)@C1\\&SL\_NodeSet(first)\cap SL\_NodeSet(second)=\varnothing\\&SL\_IsSList(first)\\&SL\_IsSList(second)\end{aligned}.$$

$SL\_NodeSet$ 定义如下:

$$SL\_NodeSet(x{:}\boldsymbol{P}(\boldsymbol{Node})){:}\boldsymbol{Set}(\boldsymbol{P}(\boldsymbol{Node}))\triangleq x=NULL?\{\}:x+SL\_DataSet(x\to Link).$$

1.1 利用抽象程序证明具体程序

在软件开发过程中,程序员常常使用抽象程序理解和描述算法.图 3 给出了翻转链表的抽象程序(Ai 表示程序点).对应地,本文将最终的实现称作具体程序(如图 2 所示).

```
List firstList;
List secondList;
A1:
  secondList=empty;
A2:
while (firstList!=empty) {
  A3:
    secondList=head(firstList)+secondList;
    firstList=tail(firstList);
  A4:
}
A5:
```

Fig.3 Abstract program of reversing list

图 3 翻转链表的抽象程序

抽象程序和具体程序之间存在对应关系.这体现在如下几个方面.

- 抽象程序中的变量实现为具体程序中的变量.

$$firstList=SL\_DataList(first),\ secondList=SL\_DataList(second).$$

$SL\_DataList$ 定义如下:

$$SL\_DataList(x{:}\boldsymbol{P}(\boldsymbol{Node})){:}\boldsymbol{List}(\boldsymbol{int})\triangleq x=NULL?[]:x\to D+SL\_DataList(x\to Link).$$

- 抽象程序中的语句实现为具体程序中的语句.

程序点 Ai~Aj 之间的语句对应程序点 Ci~Cj 之间的语句.根据语句的对应关系,可以将整个程序的一致性关系证明分解为若干语句的一致性证明义务,从而减少证明负担.

根据本文的方法,要证明翻转链表抽象程序(如图 3 所示)的循环体($A3$~$A4$)与其具体程序(图 2 所示)的循环体($C3$~$C4$)满足一致性关系,只需证明图 4 中的性质.为了证明图 4 所示的性质,只需满足下面的条件:

$$SL\_IsSList(first) \tag{1}$$

$$SL\_IsSList(second) \tag{2}$$

$$SL\_NodeSet(first)\cap SL\_NodeSet(second)=\varnothing \tag{3}$$

```
Pre:
   tmp=first->link;
 first->link=second;
   second=first;
 first=tmp;
Pos:
  SL_DataList(first)=tail(SL_DataList(first))@pre
  SL_DataList(second)=head(SL_DataList(first))@pre+SL_DataList(second)@pre
```

Fig.4 Proof of consistency relationship of reversing list

图 4 翻转链表一致性关系的证明

相对于证明具体程序而言,证明一致性关系不再需要下面的不变式.

$$SL\_DataSet(first)+SL\_DataSet(second)=SL\_DataSet(first)@C1 \quad (4)$$

公式(1)~公式(3)是关于指针的形状(shape)性质,目前有大量形状分析(shape abalysis)的工作[4−9].利用这些形状分析结果,图 4 中的证明可以达到自动化的证明.比如,可以使用文献[2,10,11]的方法达到自动化的证明.

2 程序一致性关系

本节介绍程序语法、程序的精化关系以及一致性关系.

2.1 程序语法

具体程序就是C语言语法.因此,这里只给出抽象程序的语法.抽象程序使用抽象数据结构(ADTs),如 *set*,*list*,*map* 及其上的操作.抽象程序语法如下.

$$type ::= set \mid list \mid map \mid BiRelation$$

$$st ::= e1 := e2 \mid st;st \mid \textbf{if}\ (e)\ st\ \textbf{else}\ st \mid \textbf{while}\ (e)\ st.$$

BiRalation 是两个集合的笛卡尔乘积,*map* 是数学上的函数.表 1 给出了 ADT 上的常见操作.

Table 1 Operators

表 1 操作

语法	语义
*S*1+*S*2/*S*1+*x*	集合并
*S*1-*S*2/*S*1−*x*	集合差
*S*1**S*2	集合交
x isin S	检查 *x* 是否在 *S* 中
random S	随机取一个元素
#S	取 *S* 的大小
*S*1×*S*2	笛卡尔乘积
*L*1+*L*2/*L*1+*x*	序列拼接
x isin L	检查 *x* 是否在 *L* 中
first(*L*)/*last*(*L*)	*L* 第 1 个元素/最后一个元素
tail(*L*)	去掉第 1 个元素后的序列
~L	逆转 *L*
#L	取 *L* 的大小
L(*index*)	取第 *index* 个元素
remove L(*index*)	删除第 *index* 个元素后的序列
x insert L(*index*)	*x* 添加到 *L* 的 *index* 位置之后的序列
*M*1+*M*2	函数覆盖
*M*1**M*2	函数复合
dom/*ran M*	取定义域/值域
#M	取 *M* 的大小
~M	翻转 *M*
id(*S*)	*S* 的恒等关系
S limit M/*M limit S*	限制
S limit−*M*/*M limit*−*S*	限制减
M[*x*]	*ran*(*x limit M*)

表 1 中,S 表示集合,L 表示序列,M 表示 *Map*,x 表示元素.

2.2 精化关系

程序 B 是程序 A 的精化,意味着 B 应该满足任何 A 满足的规约[12].即:在任何上下文中,B 可以替换 A.使用连接不变式(coupling invariant)I,精化条件可以表达成下面的形式[12].

$$wp(A,\top)\wedge I\Rightarrow wp(B,\neg wp(A,\neg I)) \tag{5}$$

连接不变式 I 描述 A 和 B 变量之间的关系.$wp(B,Q)$表示当程序 B 终止并且满足后置条件 Q 时,程序需要满足的最弱前置条件.精化条件(5)表示:对于任何初始条件满足连接不变式 I 的 B 的执行,存在一个 A 的天使(angelic)执行,使得 I 在 A 和 B 的最终状态仍然成立.如果程序 A 是确定性的(deterministic),精化条件(5)可以简化为

$$wp(A,\top)\Rightarrow wp(\textit{assume } I;B;A;\textit{assert } I;\top) \tag{6}$$

公式(6)要求 A 和 B 的状态空间不相交,A 和 B 的初始和最终状态满足 I.本文描述的抽象程序全部满足确定性.

2.3 一致性关系

精化关系要求具体程序能够满足抽象程序的所有前后置条件,但在实际过程中,抽象程序往往具有确定的前后置条件.因此,本文提出了比精化关系更弱的一致性关系.

为了描述抽象程序和具体程序的一致性关系,首先需要描述一致性标准(consistency criterion).一致性标准是一个二元组(S,E).S 是一个程序规约,表示为前置条件 P 和后置条件 Q 组成的二元组(P,Q).E 是抽象变量与具体变量的等式关系,E 的形式如下:

$$Abv=\exp(Cv1,Cv2,\ldots,Cvn).$$

Abv 表示抽象变量,*Cv* 表示具体变量.抽象变量等于一组具体变量的表达式.E 可以看作满足如上等式关系的连接不变式.

定义 1(一致性关系). 令 $S=(P,Q)$是一个程序规约,*ProgA* 是一个满足 $P\{ProgA\}Q$ 的抽象程序,$C=(S,E)$是一个一致性标准,一个具体程序 *ProgC* 是 *ProgA* 关于 C 的一致性实现,标记为 $ProgC<_c ProgA$,假如

$$M(P)\{ProgC\}M(Q)$$

成立,其中,M 是一个映射函数,$M(P)$和 $M(Q)$分别是通过一致性标准中的变量等式关系 E,将 P 和 Q 中的抽象变量替换为对应具体变量表达式之后的公式.

根据一致性的定义,如果 P 和 Q 是一致性标准 C 中给出的规约,并且 $ProgC<_c ProgA$,那么抽象程序 *ProgA* 满足 $P\{ProgA\}Q$,具体程序 *ProgC* 满足 $M(P)\{ProgC\}M(Q)$.

一致性关系要弱于精化关系,这体现在:*ProgC* 和 *ProgA* 是在特定的一致性标准 C 下满足一致性关系,即,*ProgC* 只需要满足特定的规约:$M(P)$和 $M(Q)$,其中,P 和 Q 是一致性标准 C 中给定的规约.直观上,一致性关系可以看作:具体程序在特定的前置条件下实现了抽象程序;而精化关系要求具体程序在任何前置条件下都实现了抽象程序.本文之所以提出一致性关系,是因为在实际过程中,抽象程序往往具有确定的前后置条件,具体程序只需在该前置条件下实现抽象程序即可满足要求.基于一致性关系,复杂程序的证明可以分成两步:首先证明抽象程序满足特定规约(本文不关注抽象程序的证明),然后证明具体程序是抽象程序在这个特定规约下的一致性实现.这样做的优点在于可以减少证明负担.

(1) 在第 2 步证明过程中,可以利用第 1 步证明的结果,从而减轻证明负担;

(2) 因为抽象程序和具体程序具有对应关系,抽象程序和具体程序的一致性证明可以分解,从而减轻了证明负担.

3 抽象程序和具体程序的对应关系

本文的方法需要用户建立抽象程序和具体程序的对应关系.主要有变量关系和程序点关系.下面分别介绍.

3.1 变量关系

变量关系由两部分构成.

(1) 抽象变量与具体变量的等式关系 E,它满足如下形式:

$$Abv=\exp(Cv1,Cv2,\ldots,Cvn).$$

Abv 表示抽象变量,Cv 表示具体变量.抽象变量等于一组具体变量的表达式,这种表示形式方便将抽象变量替换为具体变量表达式,这对于在具体程序点上生成验证义务是有用的.

(2) 具体变量数据结构内的不变式,如指针指向单链表、数组是有序的,等等.

下面用一个例子来讲解如何建立抽象变量和具体变量的等式关系.

如图 5 所示,抽象程序 *reach* 实现了从给定顶点 *v*0 找到它所有可达顶点的集合 *reachableSet* 的过程(*Ai* 和 *Ci* 表示程序点).抽象程序中,抽象集合 *nds* 描述节点集合,*nds* 上的二元关系 *edges* 表示边;在具体程序中,使用节点指针数组 *LinkedListNode***A*[*n*]表示图.

```
Set nds;
 BiRelation edges;
Set reachableSet;
Set waiting;
A1:
  reachableSet={v0};
A2:
 waiting={v0};
A3:
 while (waiting!=empty){
    A4:
      u=random waiting;
      waiting=waiting−{u};
      traversed=empty;
      Iterator it=edges{u}.iterator();
    A5:
     while (#traversed<#edges{u}){
        A6':
          v=it.next();
        A6:
        if (!(v isin reachableSet)){
           A7:
             reachableSet=reachableSet+{v};
            waiting=waiting+{v};
           A8:
        }
        A9:
          traversed=traversed+v;
        A10:
    }
    A11:
}
A12:
```

(a) 抽象程序

```
int n;
  LinkedListNode*A[n];
 bool[n] reachable;
  LinkedListNode* toBeExplored
C1:
 for (int i=0; i<n; i++){
     reachable[i]=false;
}
  reachable[v0]=true;
C2:
    toBeExplored=alloc(LinkedListNode);
 toBeExplored−>node=v0;
 toBeExplored−>next=null;
C3:
 while (toBeExplored!=null){
    C4:
      int i=toBeExplored−>node;
       toBeExplored=toBeExplored−>next;
       LinkedListNode*index=A[i];
    C5:
     while (index!=null){
       C6:
       if (reachable[index−>node]==false){
         C7:
          reachable[index−>node]=true;
           LinkedListNode*tmp=toBeExplored;
           toBeExplored=alloc(LinkedListNode);
          toBeExplored−>node=index−>node;
          toBeExplored−>next=tmp;
         C8:
       }
       C9:
        index=index−>next;
       C10:
    }
    C11:
}
C12:
```

(b) 具体程序

Fig.5 Reach program

图 5 可达集程序

首先,使用一些抽象符号将具体变量抽象成集合、序列等,这些抽象符号由逻辑描述语言提供.本文使用文献[13]中的抽象符号.抽象变量 *nds* 和具体变量之间的等式关系如下:

$$nds=0\ldots n-1.$$

然后,使用 ADT 的运算符和 lamda 表达式构造复杂的抽象数据类型,比如二元关系、*Map* 等.

这里使用笛卡尔乘积×构造二元关系,使用 lamda 表达式构造二元关系的集合.抽象变量 *edges* 和具体变量的等式关系如下:

$$edges=\lambda(x)(\{x\}\times SL\_DataSet(A[x]))\ \text{in}\ (0\ldots n-1).$$

3.2 程序点关系

本文的方法需要用户给出抽象程序程序点和具体程序程序点的对应关系.

抽象和具体的语句之间有一定的对应关系,但是抽象程序和具体程序结构之间差异较大,比如一个抽象语句可能被实现为具体程序中的一个循环语句,因此,自动建立语句之间的映射关系是比较困难的.但是程序员手动建立语句之间的对应关系是容易的,因此,本文的方法要求程序员手动建立语句之间的映射关系.

虽然抽象和具体的语句之间有一定的对应关系,但是经常会出现一些语句之间不能直接对应的情况.图 6 给出了语句无法直接对应的一个例子(*Ai* 和 *Ci* 表示程序点).

```
Set explored;
List stack;
Map L,R;
Set LR={Left,Right};
Map <nds,LR> CHK;
A1:
explored={}
 stack=[root];
 parent=null;
A2:
while (parent!=NULL||head(stack)!=
       NULL&&!(head(stack) isIn explored)){
      A3:
      if (head(stack)==NULL||head(stack) isIn explored){
            A4:
            if (CHK(parent)==Right){
                  A5:
                   stack=tail(stack);
                   parent=head(tail(stack))
                  A6:
            }
            else {
                  A7:
                   stack=tail(stack);
                   stack=R(parent)+stack;
                   CHK(parent)=Right;
                  A8:
            }
            A9:
      }
      else {
            A10:
             parent=head(stack);
             stack=L(parent)+stack;
               explored=explored+parent;
             chk(parent)=Left;
            A11:
      }
      A12:
}
A13:
```

(a) 抽象 DFS 程序

```
Struct Node{Node*l;Node*r;bool chk;bool mk;}

C1:
t=root;
p=NULL;
C2:
while ((p!=NULL)||(t!=NULL&&t->mk!=true)){
      C3:
      if (t==NULL||t->mk==true){
            C4:
            if (p->chk){
                  C5:
                  q=t; t=p; p=p->r; t->r=q;
                  C6:
            }
            else {
                  C7:
                  q=t; t=p->r; p->r=p->l; p->l=q;
                   p->chk=true;
                  C8:
            }
            C9:
      }
      else {
            C10:
               q=p; p=t; t=t->l; p->l=q;
            p->mk=true; p->chk=false;
            C11:
      }
      C12:
}
C13:
```

(b) 具体 Schorre-Waite 程序

Fig.6

图 6

图 6(b)展示了 Schorr-Waite 算法,它是一种垃圾回收的算法.一般的垃圾回收算法(图 6(a)所示)使用深度优先遍历(DFS)搜索,使用一个栈记录当前访问路径.Schorr-Waite 程序不使用栈,仅使用图本身的指针实现回溯.

表 2 给出了图 6 中的变量对应关系,其中,*STACK* 定义如下:

$$STACK(x{:}\boldsymbol{P}(\boldsymbol{Node})){:}\boldsymbol{List}(\boldsymbol{P}(\boldsymbol{Node}))\triangleq x=NULL?[]:x+STACK(x\rightarrow chk?x\rightarrow r{:}x\rightarrow l).$$

Table 2 Correspondences between some variables of Schorr-Waite

表 2 Schorr-Waite 部分变量对应关系

抽象变量	对应的具体变量表达式
nds	$NS0$
explored	$\{x\mid x\in NS0\wedge x\rightarrow mk=\text{true}\}$
stack	$[t]+STACK(p)$
L	$(\lambda(x)((x,x\rightarrow l))\text{ in }NS0)$
R	$(\lambda(x)((x,x\rightarrow r))\text{ in }NS0)$

图 6 中,整个抽象程序对应整个具体程序,但是抽象程序 *A*5~*A*6 之间的语句与具体程序 *C*5~*C*6 之间的语句不存在对应关系:因为 *C*5~*C*6 临时修改了图结构,使得抽象程序点 *A*6 成立的性质

$$L=L@A5 \tag{7}$$

$$R=R@A5 \tag{8}$$

其所对应的具体性质(9)、性质(10)在具体程序点 *C*6 上不成立.

$$(\lambda(x)((x,x\rightarrow l))\text{ in }NS0)=(\lambda(x)((x,x\rightarrow l))\text{ in }NS0)@C5 \tag{9}$$

$$(\lambda(x)((x,x\rightarrow r))\text{ in }NS0)=(\lambda(x)((x,x\rightarrow r))\text{ in }NS0)@C5 \tag{10}$$

这时,采用语句对应方式导致抽象程序和具体程序无法分解证明.但是我们注意到:尽管 *A*5~*A*6 与 *C*5~*C*6 不存在完全的对应关系,但是语句之间仍然存在部分的对应关系,这体现在大部分 *A*6 处成立的性质,如性质(11)、性质(12).

$$explored=explored@A5 \tag{11}$$

$$stack=tail(stack)@A5 \tag{12}$$

其对应的具体性质(13)、性质(14)在 *C*6 处仍然成立.

$$\{x\mid x\in NS0\wedge x\rightarrow mk=\text{true}\}=\{x\mid x\in NS0\wedge x\rightarrow mk=\text{true}\}@C5 \tag{13}$$

$$[t]+STACK(p)=tail([t]+STACK(p))@C5 \tag{14}$$

为了获取抽象性质到具体性质的映射关系,本文从程序点的角度来确定对应关系.这样操作更灵活,可以避免一些语句无法完全映射的问题.

这里需要指出的是:虽然图 6 的具体程序是抽象程序的精化,但是因为具体程序临时修改了图结构,导致语句无法分解对应,精化关系证明无法分解.比如,因为下面的连接不变式在图 6 所示的 *A*6 和 *C*6 处不成立:

$$L=(\lambda(x)((x,x\rightarrow l))\text{ in }NS0),$$

$$R=(\lambda(x)((x,x\rightarrow r))\text{ in }NS0),$$

所以图 6 所示的 *A*5~*A*6 之间的抽象语句与 *C*5~*C*6 之间的具体语句不满足精化关系.

抽象程序程序点和具体程序程序点的对应关系能够帮助程序员减轻证明负担,并且建立这种对应关系对于程序员是容易的.但并不是任何程序点的对应关系都是可行的,下面给出合适的程序点对应关系需要满足的约束.

- 合适的程序点对应关系.

合适的抽象程序程序点和具体程序程序点之间的对应关系满足下面的条件.

(1) 抽象程序的一个程序点最多只能和一个具体程序点对应,但是多个抽象程序点可能对应到同一个具体程序点.

(2) 抽象程序的入口和出口必须分别对应于具体程序的入口和出口.

(3) 如果抽象程序中某个循环语句的内部有一个程序点和具体程序点对应,那么,

a) 抽象的循环语句必然和某个具体的循环语句对应;且

b) 抽象循环语句 body 之前/之后的程序点分别对应于这个具体循环语句的 body 之前/之后的程序点;

c) 抽象循环语句之前/之后的程序点分别对应于具体程序中的程序点.

(4) 如果抽象程序中的某个 if 语句内部的程序点和某个具体程序点对应,那么,

a) 这个 if 语句必然和具体程序中的某个 if 语句所对应;

b) 抽象的 if 语句的两个分支之前/之后的程序点分别对应于相应具体 if 语句的两个分支之前/之后的程序点;

c) 抽象 if 语句之前/之后的程序点分别对应于具体程序中的程序点.

(5) 对于抽象程序中两个任意的程序点 *AP* 和 *AQ*,如果 *AP dominate AQ* 且 *AQ reverse-dominate AP*,那么它们对应的具体程序点 *CP* 和 *CQ* 在具体程序中也有 *CP dominate CQ* 且 *CQ reverse-dominate CP*.

4 验证义务

如果用户已经建立了抽象变量和具体变量之间的等式关系,并已建立具体数据结构必须满足的性质,即数据结构不变式,并且已经给出抽象程序程序点和具体程序程序点的对应关系,那么本文的方法能够在相应的具体程序点上生成验证义务.

4.1 基本验证义务

首先假设抽象程序已经是被证明的,证明过程中的性质已被加入到各个抽象程序点上.为了证明抽象程序和具体程序满足一致性关系,需要证明下面的验证义务.

- 关于一致性标准 *C* 的基本验证义务.

对于任意对应的程序点 *AP* 及其所对应的具体程序点 *CP*,必须证明如下性质.

(1) 具体数据结构必须满足的性质,即数据结构不变式,在 *CP* 上成立;

(2) 对于 *AP* 上的性质 *P*,在 *CP* 上必须证明 *M*(*P*)成立,其中,*M*(*P*)是通过一致性标准 *C* 中的等式关系 *E* 和程序点对应关系,将 *P* 中抽象变量、抽象程序点替换为对应具体变量、具体程序点之后的公式.

直观地,基本验证义务就是将抽象程序点上的证明性质映射到对应的具体程序点上证明.

定理 1. 令 *S*=(*P*,*Q*)是一个程序规约,*C*=(*S*,*E*)是一个一致性标准,*ProgA* 和 *ProgC* 分别是抽象程序和具体程序.如果 *ProgA* 满足 *P*{*ProgA*}*Q*,并且 *ProgC* 满足关于一致性标准 *C* 的基本验证义务,那么 $ProgC <_c ProgA$.

证明:因为 *P*{*ProgA*}*Q* 成立,根据一致性关系的定义,要证明 $ProgC <_c ProgA$,只需证明 *M*(*P*){*PorgC*}*M*(*Q*)成立.*P* 和 *Q* 分别是抽象程序入口和出口处的性质.因为程序点的映射必须满足合适的程序点对应关系,所以抽象程序的入口和出口必须分别对应于具体程序的入口和出口.又因为关于 *C* 的基本验证义务成立,所以 *M*(*P*)和 *M*(*Q*)分别在具体程序的入口和出口处成立,即 *M*(*P*){*PorgC*}*M*(*Q*)成立.因此,$ProgC <_c ProgA$. □

4.2 简化证明义务

抽象程序点上可能有太多的证明过程中的性质,其中,部分的证明义务对于证明抽象程序和具体程序的一致性是冗余的.本节将介绍简化证明义务的两个规则.

- 非循环语句的简化证明义务规则.

令抽象程序点 *Ai*,*Aj* 分别与 *Ci*,*Cj* 对应,*Ai* 在 *Aj* 之前.*Ast* 表示 *Ai* 和 *Aj* 之间的语句,*Cst* 表示 *Ci* 和 *Cj* 之间的语句.*Ast* 中不包括循环语句,为了证明抽象语句 *Ast* 和具体语句 *Cst* 满足一致性关系,只需在具体程序点 *Cj* 上证明如下性质:

$$M(sp(S,\overrightarrow{abv}=\overrightarrow{abv@Ai})).$$

$\overrightarrow{abv}$ 表示抽象语句 *Ast* 中的变量序列,$\overrightarrow{abv@Ai}$ 是变量序列 $\overrightarrow{abv}$ 在程序点 *Ai* 的值.在程序点 *Ai* 处,$\overrightarrow{abv}=\overrightarrow{abv@Ai}$ 恒成立.*M* 是抽象程序到具体程序的映射函数.*M*(*P*)是将性质 *P* 中的抽象变量替换为对应具体变

量,抽象程序点替换为对应具体程序点后的性质.

sp 表示最强后置条件(strongest postconditions), $sp(Ast,\overrightarrow{abv}=\overrightarrow{abv@Ai})$ 表示当程序 *Ast* 的初始条件满足 $\overrightarrow{abv}=\overrightarrow{abv@Ai}$ 时,*Ast* 终止时满足的最强的后置条件.当抽象语句 *Ast* 中没有循环语句时, $sp(Ast,\overrightarrow{abv}=\overrightarrow{abv@Ai})$ 的计算是确定的.

定理 2. 令抽象程序点 *Ai*,*Aj* 分别与 *Ci*,*Cj* 对应,*Ai* 在 *Aj* 之前.*Ast* 表示 *Ai* 和 *Aj* 之间的语句,*Cst* 表示 *Ci* 和 *Cj* 之间的语句,*S*=(*P*,*Q*)是任何程序规约,*E* 是变量对应关系,*C*=(*S*,*E*)是一致性标准.如果 *Ast* 和 *Cst* 能够应用非循环语句的简化证明义务规则并且简化证明义务成立,那么 $Cst<_cAst$.

证明:这里直接使用一个精化方法的结论.如果下面公式可以被证明,那么 *Ast* 和 *Cst* 满足精化关系[14]:

$$M(wp(Ast,\text{true})\wedge\vec{v}=\overrightarrow{v@Ai})\{Cst\}M(sp(Ast,\vec{v}=\overrightarrow{v@Ai})) \tag{15}$$

其中, $\vec{v}$ 是 *Ast* 中的变量序列, $\overrightarrow{v@Ai}$ 是变量序列 $\vec{v}$ 在程序点 *Ai* 的值.*wp*(*Ast*,true)用于检查语句 *Ast* 是否终止.当语句 *Ast* 不包含循环语句时,公式(15)可以简化为

$$M(\vec{v}=\overrightarrow{v@Ai})\{Cst\}M(sp(Ast,\vec{v}=\overrightarrow{v@Ai})) \tag{16}$$

$M(\vec{v}=\overrightarrow{v@Ai})$ 在 *Ci* 处恒成立.如果 *Ast* 和 *Cst* 能够应用非循环语句的简化证明义务规则并且简化证明义务成立,那么公式(16)成立,*Ast* 和 *Cst* 满足精化关系,因此,*Ast* 和 *Cst* 一定也满足关于一致性标准 *C* 的一致性关系 $Cst<_cAst$. □

推论 1. 如果抽象语句 *Ast* 和具体语句 *Cst* 可以应用非循环语句的简化证明义务规则,并且简化证明义务可证,那么 *Ast* 和 *Cst* 满足精化关系.

4.2.1 While 循环语句

本节考虑 while loop 语句的简化证明义务.

表 3 描述了抽象循环语句和具体循环语句,其中,抽象程序点 *Ai*,*Aj*,*Ak*,*Ap* 分别对应具体程序点 *Ci*,*Cj*,*Ck*,*Cp*. 注意,*Ci* 不一定是具体循环语句直接前驱处的程序点.

Table 3 Program points correspondence of while-statement

表 3 while 语句程序点对应关系

Abstract while statement	Concrete while statement
...	...
Ai:	*Ci*:
while (*AbCon*)	...
Aj:	*Cm*:
Ast_1	while (*ConCon*)
Ak:	*Cj*:
Ap:	Cst_1
...	*Ck*:
	Cp:
	...

- while 语句的简化证明义务规则.

如果抽象循环语句的程序点和具体循环语句的程序点满足表 3 所示的对应关系,为了证明抽象循环语句和具体循环语句满足一致性关系,只需证明下面的条件成立.

(1) 在具体程序点 *Cm* 处, $M(\overrightarrow{abv}=\overrightarrow{abv@Ai})$ 成立;

(2) 在具体程序点 *Cj* 处,*M*(*AbCon*)成立;

(3) 抽象循环体 Ast_1 和具体循环体 Cst_1 满足精化关系;

(4) 在具体程序点 *Cp* 处,*M*(¬*AbCon*)成立.

M(*P*)是将 *P* 中的抽象变量替换为对应具体变量、抽象程序点替换为对应具体程序点后的公式.

定理 3. 令 *Ast* 和 *Cst* 分别表示抽象循环语句和具体循环语句,*S*=(*P*,*Q*)是任何程序规约,*E* 是变量对应关系,*C*=(*S*,*E*)是一致性标准.如果 *Ast* 和 *Cst* 能够应用 while 语句的简化证明义务规则并且简化证明义务成立,那么

$Cst<_cAst$.

证明:如果能够证明 *Ast* 和 *Cst* 满足精化关系,那么 $Cst<_cAst$.对于任何前置条件 *P* 和后置条件 *Q* 使得 $P\{Ast\}Q$ 成立,只需证明 $M(P)\{Cst\}M(Q)$成立即可.前置条件 $M(P)$在 *Ci* 处成立,因为 $M(\overrightarrow{abv}=\overrightarrow{abv@Ai})$ 在 *C* 处成立,所以 $M(P)$在 *Cm* 成立.令 *INV* 表示任何使得公式(17)成立的不变式.

$$(P\Rightarrow INV)\wedge(INV\wedge AbCon\{Ast_1\}INV)\wedge(INV\wedge\neg AbCon\Rightarrow Q) \tag{17}$$

因为 Ast_1 和 Cst_1 满足精化关系,因此公式(18)成立.

$$(M(P)\Rightarrow M(INV))\wedge(M(INV)\wedge M(AbCon)\{Cst_1\}M(INV))\wedge(M(INV)\wedge M(\neg AbCon)\Rightarrow M(Q)) \tag{18}$$

因为在 *Cm* 处 $M(P)$成立,在 *Cj* 处 $M(AbCon)$成立,在 *Cp* 处 $M(\neg AbCon)$成立,并且公式(18)成立,因此,$M(P)\{Cst\}M(Q)$成立,*Ast* 和 *Cst* 满足精化关系.因此,$Cst<_cAst$. □

推论 2. 如果抽象语句 *Ast* 和具体语句 *Cst* 可以应用循环语句的简化证明义务规则,并且简化证明义务可证,那么 *Ast* 和 *Cst* 满足精化关系.

推论 3. 令 $S=(P,Q)$是任何程序规约,*E* 是变量对应关系,$C=(S,E)$是一致性标准.如果抽象语句 Ast_1 和 Ast_2 与具体语句 Cst_1 和 Cst_2 可以分别应用简化证明义务规则(非循环语句简化规则或者循环语句简化规则),并且简化证明义务可证,那么 $Cst_1;Cst_2<_cAst_1;Ast_2$.

综上,对变量对应关系 *E*,任何程序规约 $S=(P,Q)$,一致性标准 $C=(S,E)$,如果抽象语句 *Ast* 和具体语句 *Cst* 可以应用非循环语句简化规则或者循环语句简化规则,并且简化证明义务成立,那么 $Cst<_cAst$.

5 案例研究

本节使用我们的方法验证 reach 程序(如图 5 所示)和 Schorre-Waite 程序(如图 6 所示).

5.1 reach程序

图 5 给出了抽象 reach 程序及其具体实现.抽象程序 reach 实现了从给定顶点 *v*0 找到它所有可达顶点的集合 *reachableSet* 的过程.表 4 给出了 reach 抽象程序和具体程序的变量对应关系,其中,*SL_DataSet* 和 *SL_DataSegment* 定义分别如下所示:

$$SL\_DataSet(x:\boldsymbol{P}(\boldsymbol{Node})):\boldsymbol{Set}(\boldsymbol{int})\triangleq x=NULL\,?\,\{\}:\ x\rightarrow node+SL\_DataSet(x\rightarrow next),$$

$$SL\_DataSegment(x:\boldsymbol{P}(\boldsymbol{Node}),y:\boldsymbol{P}(\boldsymbol{Node})):\boldsymbol{Set}(\boldsymbol{int})\triangleq$$

$$(x=NULL)?\{\}:((x=y)?\{\}:p+SL\_NodeSetSeg(x\rightarrow next,y)).$$

reach 抽象程序和具体程序的程序点如图 5 所示,图 5 中的抽象程序点 *Ai* 对应具体程序点 *Ci*.

Table 4 Correspondences between variables of *reach*

表 4 *reach* 程序的变量对应关系

抽象变量	对应的具体变量表达式
nds	0… ñ1
edges	$edges=\lambda(x)(\{x\}\times SL\_DataSet(A[x]))$ in (0... ñ1)
reachableSet	$\{i\mid 0\leqslant i<n\wedge reachable[i]=\text{true}\}$
waiting	*SL_DataSet* (*toBeExplored*)
traversed	*SL_DataSegment*(*A*[*i*],*index*)
v	*index*→*node*

在给出程序点对用关系之后,本文的方法可以在具体程序上生成验证义务(目前,通过手动方式在具体程序上生成验证义务).因为抽象程序和具体程序的映射很细,因此这里可以完全应用简化证明义务规则.图 7 给出了 reach 具体程序上的验证义务,为了简化表示,图 7 使用 *v*′表示 $M(v)$,$M(v)$表示抽象变量 *v* 对应的具体变量表达式.

在具体程序上生成验证义务后,可以使用证明工具证明这些性质.因为生成了细粒度的验证义务,所以证明这些验证义务是较为容易的.具体程序上验证义务的详细证明过程,可以参考我们的工具网址:http://seg.nju.edu.cn/scl/download.html.

```
C1:
 for (int i=0; i<n; i++){
      reachable[i]=false;
}
  reachable[v0]=true;
 C2: reachableSet′={v0}∧waiting′=waiting′@C1
   toBeExplored=alloc(LinkedListNode);
 toBeExplored→node=v0;
 toBeExplored→next=null;
C3: waiting′={v0}∧reachableSet′=reachableSet′@C2
 while (toBeExplored!=null){
      C4: waiting′!={}
       int i=toBeExplored→node;
       toBeExplored=toBeExplored→next;
       LinkedListNode*index=A[i];
   C5: waiting′=waiting′@C4−{i}∧traversed′={}∧
        reachableSet′=reachableSet′@C4
      while (index!=null){
         C6: #traversed<#edges{u}
        if (reachable[index→node]==false){
         C7: !(index→node isin reachableSet′)
           reachable[index→node]=true;
           LinkedListNode*tmp=toBeExplored;
            toBeExplored=alloc(LinkedListNode);
            toBeExplored→node=index→node;
            toBeExplored→next=tmp;
          C8: reachableSet′=reachableSet′@C7+{index→node}@C7∧
            waiting′=waiting@C7+{index→node}@C7∧
            traversed′=traversed′@C7
        }
        C9:
        index=index→next;
         C10: traversed′=traversed′@C9+{index→node}@C9∧
                reachableSet′=reachableSet′@C9∧
                waiting′=waiting@C9
     }
     C11: !(#traversed<#edges{u})
}
C12: waiting′={}
```

Fig.7 Proof obligations of reach

图 7 reach 程序的验证义务

5.2 Schorre-Waite程序

图 6(a)展示了一般的垃圾回收算法,它使用深度优先遍历(DFS)搜索,使用一个栈记录当前访问路径.图 6(b)展示了 Schorr-Waite 算法,它不使用栈,仅使用图本身的指针实现回溯.表 5 给出了 Schorre-Waite 抽象程序和具体程序的变量对应关系,其中,*NS*0 表示节点图中所有节点引用的集合.

Table 5 Correspondences between variables of Schorre-Waite

表 5 Schorre-Waite 程序的变量对应关系

抽象变量	对应的具体变量表达式
Nds	*NS*0
explored	$\{x\|x\in NS0\wedge x\rightarrow mk=\text{true}\}$
stack	$[t]+STACK(p)$
parent	*p*
CHK	$\{(x,x\rightarrow chk)\|x\in NS0\}$
L	$(\lambda(x)((x,x\rightarrow l))$ in *NS*0)
R	$(\lambda(x)((x,x\rightarrow r))$ in *NS*0)
Left	false
Right	true

STACK 定义如下:

$$STACK(x{:}\boldsymbol{P}(\boldsymbol{Node})){:}\boldsymbol{List}(\boldsymbol{P}(\boldsymbol{Node}))\triangleq x=NULL?[]:x+STACK(x\rightarrow chk?x\rightarrow r:x\rightarrow l).$$

具体变量数据结构内的不变式为

$$IsRotateList(p)\wedge \underset{x\in STACK(p)}{\wedge}(x\rightarrow mk).$$

IsRotateList 定义如下:

$$IsRotateList(x{:}\boldsymbol{P}(\boldsymbol{Node})){:}\boldsymbol{bool}\triangleq(x=NULL)?\text{true}:x+IsRotateList(x\rightarrow chk?x\rightarrow r:x\rightarrow l).$$

图 8 给出了 Schorre-Waite 具体程序上的验证义务(目前是通过手动方式在具体程序上生成验证义务).需要指出的是:图 8 中的验证义务只是关于 *explored*,*stack*,*CHK*,*parent* 变量的验证义务,没有关于 *L*,*R* 变量的验证义务.如果一致性标准 *C* 中的规约是关于 *explored*,*stack*,*CHK*,*parent* 变量的性质(不包括 *L*,*R* 变量的性质),那么图 8 的验证义务可以证明 Schorre-Waite 程序是抽象 DFS 垃圾回收算法关于 *C* 的一致性实现.图 8 使用 *INV* 表示具体变量数据结构不变式.

```
Struct Node{Node*l;Node*r;bool chk;bool mk;}
Pre:
  ForAll x in NS0()⇒x→mk=false∧x→chk=false∧root!=null∧root isIn NS0()
C1:
t=root;
p=NULL;
      C2: explored′={}∧stack′=[root]∧p=NULL∧CHK′=CHK′@C1∧INV
while   ((p!=NULL)||((t!=NULL) && t→mk!=true)){
              C3: (p!=NULL||(head(stack′)!=NULL && head(stack′) isIn explored′)
        if  (t==NULL||t→mk==true){
              C4: head(stack′)=NULL||head(stack′) isIn explored′∧stack′=stack′@C3∧
                  explored′=explored′@C3∧CHK′=CHK′@C3∧p=p@C3
              if  (p→chk){
                        C5: (p,true) isIn CHK′∧INV∧stack′=stack′@C4∧explored′=explored′@C4∧
                            CHK′=CHK′@C4∧p=p@C4
                        q=t; t=p; p=p→r; t→r=q;
                            C6: stack′=tail(stack′)@C5∧p=head(tail(tail(stack′)@C5))@C5∧
                                explored′=explored′@C5∧CHK′=CHK′@C5∧INV
              }
               else {
                         C7: not ((p,true) isIn CHK′)∧INV∧stack′=stack′@C4∧explored′=explored′@C4∧
                             CHK′=CHK′@C4∧p=p@C4
                         q=t; t=p→r; p→r=p→l; p→l=q; p→chk=true;
                           C8: stack′=(p→r)@C7+tail(stack′)@C7∧(p@C7,true) isIn CHK′∧
                               explored′=explored′@C7∧INV
              }
               C9:
        }
         else {
              C10: not (head(stack′)=NULL||head(stack′) isIn explored′)∧INV∧
                   stack′=stack′@C3∧explored′=explored′@C3∧CHK′=CHK′@C3∧p=p@C3
               q=p; p=t; t=t→l; p→l=q; p→mk=true; p→chk=false;
               C11: p=head(stack′)@C10∧stack′=((head(stack′)@C10)→l)@C10+stack′@C10∧INV∧
                    explored′=explored′@C10+head(stack′)@C10∧(head(stack′)@C10,false) isIn CHK′
        }
         C12:
}
  C13: not (p!=NULL||(head(stack′)!=NULL && head(stack′) isIn explored′)
```

Fig.8 Proof obligations of Schorre-Waite

图 8 Schorre-Waite 程序的验证义务

附录描述了 Schorre-Waite 大概的证明过程.具体程序上验证义务的详细证明过程,可以参考我们的工具网址:http://seg.nju.edu.cn/scl/download.html.

6 相关工作和讨论

6.1 精 化

精化演算(refinement calculus)方法广泛被研究[15−18],精化方法分为操作精化(operation refinement)和数据精化(data refinement).

- 操作精化是指在不改变规约的前提下,从抽象操作推导更加具体的操作.它也被称为算法设计(algorithm design);
- 数据精化[19−25]是指使用具体数据表示替换抽象数据表示.假设 a 是一组抽象变量(abstract),c 是一组具体变量(concrete),I 是公式,称作连接不变式.程序 *progA* 通过 a,c,I 数据精化到 *progC* 当且仅当对所有的不包含 c 的后置条件 A,有:

$$(\exists a.I\wedge wp(progA,A))\Rightarrow wp(progC,\exists a.I\wedge A).$$

本文的方法借鉴了精化的思想,但并非用于证明精化关系,而是用于证明一致性关系.本文的方法通过程序点之间的映射分解来证明.

6.2 程序验证

程序员首先给出待验证程序,并使用前置条件(pre-condition)、后置条件(post-condition)、断言(assertion)和不变式(invariant)标记程序.这些标记(annotation)不仅有软件规约(specification),而且包含不变式,这些不变式分解标记程序成为不含循环的 Hoare 三元组(Hoare triples).最终,利用程序语言的语义,将程序验证(program verification)规约成推导逻辑公式是否正确,这可以使用定理证明器进行推导.工具 Bedrock[2],VCC[26],AFNY[27],HAVOC[28],VeriFast[1],jStar[29]和 Smallfoot[30]等都属于这一类框架.其中一些工具虽然能够减轻程序证明负担,比如 Bedrock 支持接近自动化(mostly-automated)的程序证明[2],但它需要用户给出准确的辅助定理(low-level lemmas)和证明策略(proof tactics)来指导验证,需要用户提供合适的循环不变(loop invariant)来处理循环语句.Cao[11]开发了针对单链表的特殊策略,从而可以接近自动化的处理单链表程序.但是,当处理非单链表程序时,仍然需要用户给出准确的辅助定理和合适的证明策略和不变式.相比这些方法,本文的方法通过分层将证明负担解耦到两个部分:首先证明抽象程序,然后证明具体程序是抽象程序的一致性实现.抽象程序抽离了实现细节,比如没有指针,证明相对容易.抽象程序和具体程序的一致性证明可以被分解,分解后的证明相对容易,甚至可能自动化.

本文的方法使用 scope logic[13]作为后端 logic 工具.Scope logic 是 Hoare logic 的扩展,它可以处理指针程序.它的主要观察是可以语法构造出表达式 e 的值依赖的内存单元集合,这个内存单元集合称为表达式 e 的 memory scope.Scope logic 支持 local reasoning,支持程序点指定(program-point-specific)表达式 $e@i$.$e@i$ 表示表达式 e 在程序点 i 处的值.

7 结束语

本文提出了一个证明复杂具体程序的框架:首先证明抽象程序满足规约,然后证明具体程序和抽象程序满足一致性.这种分层次证明可以简化证明难度.本文具体介绍了证明抽象程序和具体程序满足一致性的方法.抽象程序和具体程序的一致性证明可以分解,分解后往往变成对抽象数据类型的操作,从而容易证明并可能自动证明.

References:

[1] Jacobs B, Smans J, Philippaerts P, Vogels F, Penninckx W, Piessens F. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In: Bobaru M, ed. NASA Formal Methods. Springer-Verlag, 2011. 41−55. [doi: 10.1007/978-3-642-20398-5_4]

[2] Chlipala A. Mostly-Automated verification of low-level programs in computational separation logic. In: Hall M, ed. Proc. of the Programming Language Design and Implementation. ACM Press, 2011. 234−245. [doi: 10.1145/1993498.1993526]

[3] 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. 2014,49(6):440−451. [doi: 10.1145/2594291.2594325]

[4] Distefano D, O'hearn PW, Yang H. A local shape analysis based on separation logic. In: Hermanns H, ed. Proc. of the Tools and Algorithms for the Construction and Analysis of Systems. Vienna: Springer-Verlag, 2006. 287−302. [doi: 10.1007/11691372_19]

[5] Lev-Ami T, Immerman N, Reps T, Sagiv M, Srivastava S, Yorsh G. Simulating reachability using first-order logic with applications to verification of linked data structures. In: Nieuwenhuis R, ed. Proc. of the Automated Deduction (CADE-20). Tallinn: Springer-Verlag, 2005. 99−115. [doi: 10.1007/11532231_8]

[6] Lev-Ami T, Sagiv M. TVLA: A system for implementing static analyses. In: Palsberg J, ed. Proc. of the Static Analysis. Santa Barbara: Springer-Verlag, 2000. 280−301. [doi: 10.1007/978-3-540-45099-3_15]

[7] Balaban I, Pnueli A, Zuck LD. Shape analysis by predicate abstraction. In: Cousot R, ed. Proc. of the Verification, Model Checking, and Abstract Interpretation. Paris: Springer-Verlag, 2005. 164−180. [doi: 10.1007/978-3-540-30579-8_12]

[8] Chatterjee S, Lahiri SK, Qadeer S, Rakamarić Z. A reachability predicate for analyzing low-level software. In: Grumberg O, ed. Proc. of the Tools and Algorithms for the Construction and Analysis of Systems. Braga: Springer-Verlag, 2007. 19−33. [doi: 10.1007/978-3-540-71209-1_4]

[9] Lahiri SK, Qadeer S. Verifying properties of well-founded linked lists. ACM SIGPLAN Notices, 2006,41(1):115−126. [doi: 10.1145/1111320.1111048]

[10] Lahiri S, Qadeer S. Back to the future: Revisiting precise program verification using SMT solvers. ACM SIGPLAN Notices, 2008, 43(1):171−182. [doi: 10.1145/1328897.1328461]

[11] Cao J, Fu M, Feng X. Practical tactics for verifying C programs in Coq. In: Leroy X, ed. Proc. of the 2015 Conf. on Certified Programs and Proofs. Mumbai: ACM Press, 2015. 97−108. [doi: 10.1145/2676724.2693162]

[12] Gries D, Prins J. A new notion of encapsulation. ACM SIGPLAN Notices, 1985,20(7):131−139. [doi: 10.1145/17919.806834]

[13] Zhao JH, Li XD. Scope logic: An extension to Hoare logic for pointers and recursive data structures. In: Liu ZM, ed. Proc. of the Theoretical Aspects of Computing (CICTAC 2013). Shanghai: Springer-Verlag, 2013. 409−426. [doi: 10.1007/978-3-642-39718-9_24]

[14] Back RJR. A calculus of refinements for program derivations. Acta Informatica, 1988,25(6):593−624. [doi: 10.1007/BF00291051]

[15] Back RJR. Correctness preserving program refinements: Proof theory and applications. MC Tracts, 1980,131:1−118.

[16] Morgan C, Robinson K. Specification statements and refinement. IBM Journal of Research and Development, 1987,31(5):546−555. [doi: 10.1147/rd.315.0546]

[17] Morris JM. A theoretical basis for stepwise refinement and the programming calculus. In: Sintzoff M, ed. Proc. of the Science of Computer programming. Amsterdam: Elsevier, 1987. 287−306. [doi: 10.1016/0167-6423(87)90011-6]

[18] Morgan C. The specification statement. ACM Trans. on Programming Languages and Systems (TOPLAS), 1988,10(3):403−419. [doi: 10.1145/44501.44503]

[19] Gardiner PH, Morgan CC. Data refinement of predicate transformers. Theoretical Computer Science, 1991,87(1):143−162. [doi: 10.1016/0304-3975(91)90029-2]

[20] Morgan C. Auxiliary variables in data refinement. Information Processing Letters, 1988,29(6):293−296. [doi: 10.1016/0020-0190(88)90227-X]

[21] Morris JM. Laws of data refinement. Acta Informatica, 1989,26(4):287−308. [doi: 10.1007/BF00276019]

[22] Morgan C, Gardiner PH. Data refinement by calculation. Acta Informatica, 1990,27(6):481−503. [doi: 10.1007/BF00277386]

[23] Chen W, Udding JT. Towards a calculus of data refinement. In: van de Snepscheut JLA, ed. Proc. of the Mathematics of Program Construction. London: Springer-Verlag, 1989. 197−218. [doi: 10.1007/3-540-51305-1_11]

[24] Leuschel M, Butler M. Automatic refinement checking for B. In: Lau KK, ed. Proc. of the Formal Methods and Software Engineering. Manchester: Springer-Verlag, 2005. 345−359. [doi: 10.1007/11576280_24]

[25] Burdy L, Meynadier JM. Automatic refinement. In: Proc. of the BUGM at FM. 1999. 99. https://link.springer.com/book/10.1007%2F3-540-48118-4

[26] Cohen E, Dahlweid M, Hillebrand M, Leinenbach D, Moskal M, Santen T, Schulte W, Tobies S. VCC: A practical system for verifying concurrent C. In: Berghofer S, ed. Proc. of the Theorem Proving in Higher Order Logics. Munich: Springer-Verlag, 2009. 23−42. [doi: 10.1007/978-3-642-03359-9_2]

[27] Leino KR. Dafny: An automatic program verifier for functional correctness. In: Clarke EM, ed. Proc. of the Logic for Programming, Artificial Intelligence, and Reasoning. Dakar: Springer-Verlag, 2010. 348−370. [doi: 10.1007/978-3-642-17511-4_20]

[28] Condit J, Hackett B, Lahiri SK, Qadeer S. Unifying type checking and property checking for low-level code. In: Shao Z, ed. Proc. of the Principles of Programming Languages. Savannah: ACM Press, 2009. 302−314. [doi: 10.1145/1594834.1480921]

[29] Distefano D, Parkinson JMJ. jStar: Towards practical verification for Java. ACM Sigplan Notices, 2008,43(10):213−226. [doi: 10.1145/1449955.1449782]

[30] Berdine J, Calcagno C, O'hearn PW. Smallfoot: Modular automatic assertion checking with separation logic. In: de Boer FS, ed. Proc. of the Formal Methods for Components and Objects. Amsterdam: Springer-Verlag, 2006. 115−137. [doi: 10.1007/11804192_6]

附录:Schorre-Waite 验证义务的证明过程

本节整体描述 Schorre-Waite 程序上验证义务(如图 8 所示)的证明过程.Schorre-Waite 的一致性证明分解后变成对若干基本块的证明,基本块内的语句主要是对栈(rotate list 实现)的头部进行操作,这些操作较为简单,很多工作或工具可以较为容易地进行证明.这里使用 scope logic 工具进行证明.下面给出大概的证明过程.

下面的证明使用 v'表示 $M(v)$.$M(v)$表示抽象变量 v 对应的具体变量表达式.$NS0$ 表示节点图中所有节点引用的集合.下面给出证明过程中用到的符号的定义.

$$explored=\{x\mid x\in NS0\wedge x\to mk=\text{true}\},$$

$$stack=[t]+STACK(p),$$

$$CHK=\{(x,x\to chk)\mid x\in NS0\},$$

$$INV=IsRotateList(p)\wedge\underset{x\in STACK(p)}{\wedge}(x\to mk),$$

$$STACK(x:P(Node)):List(P(Node))\triangleq x=NULL?[]:x+STACK(x\to chk?x\to r:x\to l),$$

$$IsRotateList(x:P(Node)):bool\triangleq(x=NULL)?\text{true}:IsRotateList(x\to chk?x\to r:x\to l).$$

A.1 C2处验证义务的证明过程(如图9所示)

Pre:
ForAll x in NS0()⇒x→mk=false && x→chk=false∧
root!=null∧root isIn NS0()
*C*1:
t=*root*;
p=*NULL*;
C2:
explored′={}∧stack′=[root]∧p=NULL∧CHK′=CHK′@C1∧INV

Fig.9 Proof obligations of first two lines

图 9 Schorre-Waite 程序前两行的验证义务

使用工具求解 *C*2 处的每个验证义务在 *C*1 处的最弱前置条件.

在 *C*1 处的求解得到的最弱前置条件是

$$explored'=\{\}\wedge[root]+[]=[root]\wedge\text{true}\wedge CHK'=CHK'@C1\wedge IsRotateList(NULL)\wedge\underset{x\in STACK(NULL)}{\wedge}(x\to mk),$$

其中,$CHK'=CHK'@C1$, $IsRotateList(NULL)\wedge\underset{x\in STACK(NULL)}{\wedge}(x\to mk)$ 在 *C*1 处恒成立.

通过前置条件,可以推出 $explored'=\{\}$成立.

A.2 C5和C6处验证义务的证明过程(如图10所示)

使用工具求解 *C*6 处的每个验证义务在 *C*5 处的最弱前置条件.

在 $C5$ 处的求解得到的最弱前置条件是

$$[p]+STACK(p\to r)=tail([t]+STACK(p)) \tag{19}$$

$$p\to r=head(tail(tail([t]+STACK(p)))) \tag{20}$$

$$IsRotateList(p\to r)\wedge \underset{x\in STACK(p\to r)}{\wedge}(x\to mk) \tag{21}$$

其中,$C5$ 处 $p\to chk$ 成立,所以 $STACK(p)=[p]+STACK(p\to r)$成立,所以性质(19)、性质(20)可证.又因为 INV 在 $C5$ 处成立,所以性质(21)成立.

C5: (p,true) isIn CHK′∧stack′=stack′@C4∧explored′=explored′@C4∧CHK′=CHK′@C4∧p=p@C4∧INV
$q=t$; $t=p$; $p=p\to r$; $t=r=q$;
C6: stack′=tail(stack′)@C5∧p=head(tail(tail(stack′)@C5))@C5∧explored′=explored′@C5∧CHK′=CHK′@C5∧INV

Fig.10 Proof obligations of $C5$ and $C6$

图 10 $C5$ 和 $C6$ 处的验证义务

A.3 C7和C8处验证义务的证明过程(如图11所示)

C7: not ((p,true) isIn CHK′)∧stack′=stack′@C4∧explored′=explored′@C4∧CHK′=CHK′@C4∧p=p@C4∧INV
$q=t$; $t=p\to r$; $p\to r=p\to l$; $p\to l=q$; $p\to chk$=true;
C8: stack′=(p→r)@C7+tail(stack′)@C7∧(p@C7,true) isIn CHK′∧explored′=explored′@C7∧INV

Fig.11 Proof obligations of $C7$ and $C8$

图 11 $C7$ 和 $C8$ 处的验证义务

首先,在 $C8$ 处证明 $STACK(p)=STACK(p)@C7$ 成立.因为 $C8$ 处 $p\to chk$ 成立,所以,

$$STACK(p)=[p]+STACK(p\to r) \tag{22}$$

成立.只需证明公式(23)成立.

$$[p]+STACK(p\to r)=STACK(p)@C7 \tag{23}$$

公式(23)在 $C7$ 处的最弱前置条件为

$$[p]+STACK(p\to l)=STACK(p) \tag{24}$$

在 $C7$ 处,$\neg p\to chk$ 成立,所以公式(24)成立.

下面分别证明 $C8$ 处的验证义务.

1) 将验证义务 $stack'=(p\to r)@C7+tail(stack')@C7\wedge \underset{x\in STACK(p)}{\wedge}(x\to mk)$ 中的 $STACK(P)$替换为

$$STACK(P)@C7,$$

求解它们在 $C7$ 处的最弱前置条件,它们的最弱前置条件是

$$[p\to r]+STACK(p)=[p\to r]+tail([t]+STACK(p))\wedge \underset{x\in STACK(p)}{\wedge}(x\to mk) \tag{25}$$

根据 $C7$ 处的性质,公式(25)显然成立;

2) $p=p@C7$ 在 $C8$ 处成立(通过最弱前置条件可证),因为 $p\to chk$ 成立,故$(p@C7,\text{true})\in CHK'$在 $C8$ 处成立;

3) $explored'=explored'@C7$ 通过最弱前置条件可证;

4) 在 $C8$ 处,因为 $p\to chk$ 成立,所以要想证明 $IsRotateList(p)$,只需证明 $IsRotateList(p\to r)$成立.

$IsRotateList(p\to r)$在 $C7$ 处的最弱前置条件是 $IsRotateList(p\to l)$.$IsRotateList(p\to l)$在 $C7$ 处成立.

A.4 C10和C11处验证义务证明过程(如图12所示)

C10: not (head(stack′)=NULL||head(stack′) isIn explored′)∧INV∧
stack′=stack′@C3∧explored′=explored′@C3∧CHK′=CHK′@C3∧p=p@C3
$q=p$; $p=t$; $t=t\to l$; $p\to l=q$; $p-mk$=true; $p\to chk$=false;
C11: p=head(stack′)@C10∧stack′=((head(stack′)@C10)→l)@C10+stack′@C10∧
explored′=explored′@C10+head(stack′)@C10∧(head(stack′)@C10,false) isIn CHK′∧INV

Fig.12 Proof obligations of $C10$ and $C11$

图 12 $C10$ 和 $C11$ 处的验证义务

首先,在 $C11$ 处证明 $STACK(p)=([t]+STACK(p))@C10$ 成立.

因为 $C11$ 处 $\neg p\rightarrow chk$,所以 $STACK(p)=[p]+STACK(p\rightarrow l)$成立.

所以要证明 $STACK(p)=([t]+STACK(p))@C10$,只需证明公式(26)成立.

$$[p]+STACK(p\rightarrow l)=([t]+STACK(p))@C10 \tag{26}$$

公式(26)在 $C10$ 处的最弱前置条件是 true,因此 $STACK(p)=([t]+STACK(p))@C10$ 在 $C11$ 处成立.下面分别证明 $C11$ 处的验证义务.

1) $p=head(stack)@C10$ 在 $C10$ 处的最弱前置条件是 $t=head([t]+STACK(p))$,因此 $p=head(stack)@C10$ 在 $C11$ 处成立;

2) 将 $C11$ 处的验证义务 $stack'=((head(stack')@C10)\rightarrow l)@C10+stack'@C10\wedge\underset{x\in STACK(p)}{\wedge}(x\rightarrow mk)$ 中的 $STACK(P)$替换为$([t]+STACK(p))@C10$,求解它们的最弱前置条件,其最弱前置条件是

$$[t\rightarrow l]+[t]+STACK(p)=[(head([t]+STACK(p)))\rightarrow l]+[t]+STACK(p)\wedge\underset{x\in[t]+STACK(p)}{\wedge}(t\neq x?x\rightarrow mk:\text{true}) \tag{27}$$

根据 $C10$ 处的性质,公式(27)可证;

3) 由 $p=head(stack)@C10$ 和$\neg p\rightarrow chk$ 推出$(head(stack)@C10,\text{false})\in CHK'$在 $C11$ 处成立;

4) 在 $C11$ 处,因为$\neg p\rightarrow chk$,所以要想证明 $IsRotateList(p)$,只需证明 $IsRotateList(p\rightarrow l)$成立.

$IsRotateList(p\rightarrow l)$在 $C10$ 处的最弱前置条件是 $IsRotateList(p).IsRotateList(p)$在 $C11$ 处成立;

5) 在 $C11$ 处,因为 $p\rightarrow mk$ 和 $p=head(stack)@C10$ 成立,所以 $explored'=EXPLORED(NS0-\{p\})+\{p\}$成立.其中,$EXPLORED$ 定义如下:

$$EXPLORED(S:\boldsymbol{Set}(\boldsymbol{P}(\boldsymbol{Node}))):\boldsymbol{Set}(\boldsymbol{P}(\boldsymbol{Node}))\triangleq\{x|x\in S\wedge x\rightarrow mk=\text{true}\}.$$

$EXPLORED(NS0-\{p\})+\{p\}=EXPLORED(NS0)@C10+\{p\}$在 $C10$ 处的最弱前置条件是

$$EXPLORED(NS0-\{t\})+\{t\}=EXPLORED(NS0)+\{t\}.$$

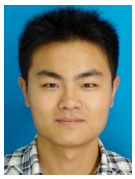
在 $C10$ 处,因为$\neg t\rightarrow mk$ 成立,所以 $EXPLORED(NS0-\{t\})=EXPLORED(NS0)$成立.

因此,$explored'=explored'@C10+head(stack')@C10$ 在 $C11$ 处成立.

综上,Schorre-Waite 的一致性证明分解后变成对若干基本块的证明,它们可以通过求解最弱前置条件,或者稍微作一些变换后求解最弱前置条件来证明.证明过程相对简单.

李彬(1988－),男,河北邯郸人,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,主要研究领域为软件工程,程序分析,程序验证,程序合成.



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

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.