Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2017-CJ-007

2017-CJ-007

自动分析递归数据结构的归纳性质

汤震浩,李彬,翟娟,赵建华

软件学报 2017

软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.005467]

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

自动分析递归数据结构的归纳性质*

汤震浩[1,2], 李 彬[1,2], 翟 娟[1,2], 赵建华[1,2]

[1](南京大学 计算机科学与技术系,江苏 南京 210023)

[2](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: 提出了一种对递归数据结构的归纳性质进行自动化分析的框架.工作分为三个主要部分.首先,它将递归数据结构的归纳性质分为两个主要类别,并提出对应的处理模式,从而帮助简化对于程序中的递归数据结构上的相关性质的分析.其次,提出了一种称为分割与拼接的技术来发现和描述递归数据结构是如何被程序修改的:递归数据结构首先被分割为若干个互不相交的片段,然后这些片段以新的方式重新拼接在一起,形成一个新的数据结构.这个技术的重点在于如何将程序原有的性质保留下来,从而为后面的分析过程所使用.最后,提出了一种调用上下文敏感的程序摘要过程间分析方法.案例分析和实验结果表明我们的分析框架可以有效地分析递归数据结构的归纳性质,并生成对程序证明过程有用的断言.

关键词: 霍尔式程序证明; 程序分析;递归数据结构;归纳性质;过程间分析

中图法分类号: TP311

中文引用格式: 汤震浩,李彬,翟娟,赵建华. 自动分析递归数据结构的归纳性质.软件学报,2018,29(6). http://www.jos.org.cn/1000-9825/5467.htm

英文引用格式: Tang ZH, Li B, Zhai J, Zhao JH. Analyzing Inductively Defined Properties for Recursive Data Structures. Ruan Jian Xue Bao/Journal of Software, 2018,29(6) (in Chinese). http://www.jos.org.cn/1000-9825/5467.htm

Automatically Analyzing Inductive Properties for Recursive Data Structures

TANG Zhen-Hao[1,2], LI Bin[1,2], ZHAI Juan[1,2], ZHAO Jian-Hua[1,2]

[1](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

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

Abstract: This paper proposes a framework facilitating the automatic analysis on inductive properties for recursive data structures. Our work has three main parts. First, it helps simplify the analysis of heap-manipulating programs by classifying inductive properties of recursive data structures into two classifications and each of them is handled with observed patterns. Second, we propose a technique called slicing and splicing to track and specify how data structures are manipulated by programs, in which data structures are first sliced into several parts and these parts are further spliced into new data structures. The key idea of this technique is to preserve the properties of original data structures, which can be used by further analysis. Third, this work presents a calling context sensitive interprocedural analysis to compute program summaries. A case study and experimental results show that our analysis framework can effectively analyze inductive properties for recursive data structures, resulting in assertions that are helpful in program verificaiton.

Key words: Hoare-style program verification; program analysis; recursive data structures; inductive properties; interprocedural analysis

霍尔式的程序证明(Hoare-style program verification)是保证代码功能正确性的重要手段.然而实现证明过

* 基金项目: 国家重点研发计划项目课题(2016YFB1000802); 国家自然科学基金(61632015,No.61561146394)
本文由形式化方法的理论基础专刊特约编辑傅育熙教授、李国强副教授、田聪教授推荐.
收稿时间: 2017-07-01; 修改时间: 2017-09-01; 采用时间: 2017-11-06; jos 在线出版时间: 2017-12-28
CNKI 网络优先出版: 2017-12-29 13:19:25, http://kns.cnki.net/kcms/detail/11.2560.TP.20171229.1318.007.html

程的自动化却受到两个主要问题的制约.这两个问题分别对应于霍尔逻辑[24] (Hoare logic)中的两个规则,即循环规则(rule of iteration)和赋值规则(axiom schema of assignment).第一,循环规则的应用依赖于循环不变式,而循环不变式的自动生成本身就是一个难题.第二,赋值规则事实上是一个简单的变量替换规则.它只适用于简单值类型的变量替换,在指针和别名存在的情况下直接替换往往会得到错误的结果.相关工作[1,2]扩展了霍尔逻辑的语义,使之能够处理指针和别名.但逻辑本身只是提供了表达能力,为了证明带有指针或别名程序的正确性,还需要进行额外的分析来获取内存布局和结构信息.典型的例子是,一些较为复杂的性质,如递归定义的谓词,往往依赖于较为复杂的其内存踪迹(memory footprint).

在尝试对程序进行自动化分析和证明的过程中,研究人员发现有一类程序较难处理,即对递归数据结构进行操作的程序.难点主要在于两个方面:1)递归数据结构往往访问较多的堆内存,内存踪迹较为复杂;2)待证明的断言往往包含用户自定义的(递归)谓词或函数.在这种情况下,要确保程序正确地操作了递归数据结构,并达到预期的功能就较为困难.

针对这个问题,很多相关工作被提出来.这些工作集中在逻辑,分析技术和证明技术等方面,分别针对程序的不同方面.文献[5-7]使用形状分析(shape analysis)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](separation logic)来描述这些性质.文献[11,12]提出了自然证明(natural proofs)技术来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所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来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]对多年来了相关工作进行了综述性的介绍.

尽管这些工作尝试解决这个自动化问题,但研究现状仍然令人不太乐观.一个主要的问题是,现有的技术较为复杂,学习成本较高,对于普通程序员来说难以使用.比如,程序员想使用抽象解释来分析一个特定性质,他们需要为这个具体的域(domain)找到一个伽罗瓦连接(Galois connection),这显然超出了大部分程序的知识范畴.而我们希望能够提出一个方法,能够帮助程序员或程序验证人员快速地分析递归数据结构的性质.具体而言有三个设计目标:1)仅仅需要程序员掌握关于程序分析和证明的基本知识,如断言、前后置条件、霍尔逻辑和以及给出待证明性质的定义等;2)提供对于递归数据结构归纳性质的自动化分析能力,其结果在使用 SMT 求解器确认之后,可以作为证明义务(proof obligation)辅助霍尔式的程序证明过程;3)能够与其它技术良好结合,既可利用其它分析技术的结果,也可以将分析结果提供给其它分析技术.总的来说,我们希望我们的方法易学、易用并且具有一定的灵活性.当然,保证这个方法能够完全的自动化无论是从理论上还是从研究现状上看都是不现实的,但是我们希望在自动化分析遇到问题时能够在程序员的简单参与下完成.

为了使我们的框架足够简洁,本文只关注递归数据结构的归纳性质,即那些能够通过归纳定义的方法给出的性质,如双链表的有序性和二叉树的高度等.本文主要从三个方面关注递归数据结构以及操作于其上的程序,并提出了一个分析框架.它具有以下特点:

- 观察并总结出递归数据结构的归纳性质两种典型的变化模式,并将这种模式应用于新性质生成的过程,可以简化分析过程和提高分析效率.
- 提出了一种称为分割与拼接(slicing and splicing)的技术来帮助我们分析递归数据结构的性质.这个技术扩展经典的展开与匹配[11,12](unfolding and matching)技术,具有良好的通用性和较高的准确度.
- 提出了一种以性质为导向的过程间分析过程,主要解决程序摘要生成与实例化不够精确的问题.这个分析过程通过自上而下的方式,根据函数调用的上下文来指导函数摘要的生成.其优点在于可以对满足两种典型变化模式的性质实现完全自动化,并在精确度和效率之间达到平衡.

本文第 1 节介绍方法的整体框架.第 2 节介绍递归数据结构的迭代定义性质的典型变化模式.第 3 节介绍

利用分割与拼接技术进行过程内分析的过程.第 4 节介绍如何生成和实例化程序摘要的过程间分析.第 5 节我们说明我们方法如何应用到一个具体程序和性质上.第 6 节介绍工具的实现和实验结果.第 7 节是相关工作和总结.

1 整体分析框架

本文提出的对递归数据结构的归纳性质的整体分析框架如图 1 所示.框架的输入包括两个部分,一部分是待证明性质的归纳定义,另一部分是带有前置条件的程序.对于程序中的分支、循环和函数调用,我们分别通过以下方式进行处理:带有分支的程序通过枚举所有可能的路径,可以得到若干个不带分支的顺序程序(sequential program,或称为直线程序),逐个分析这些顺序程序后将结果进行合并;带有循环的程序,我们需要通过其它的分析技术得到(必要时要求程序员手动输入)循环不变式,此时可以对循环的证明只需要考虑一次其循环体即可.对分支和循环的处理,我们依照的是标准的霍尔逻辑的处理方式,分别对应于分支规则和循环规则:

$$\frac{\{B \wedge P\}\ \mathrm{S}\ \{Q\}, \{\neg B \wedge P\}\ \mathrm{T}\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ \mathrm{S}\ \textbf{else}\ \mathrm{T}\ \{Q\}}\ (\text{分支规则}),\quad \frac{\{P \wedge B\}\ \mathrm{S}\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{then}\ \mathrm{S}\ \{\neg B \wedge P\}}\ (\text{循环规则}).$$

对于分支的处理方式可能会遇到的一个问题是,由于分支条件过多而导致出现过多函数分解的情况(函数的个数与分支数成指数增长的关系).因此我们具体的处理方法是,对于一个模块化程度较好的程序,单个函数的长度和分支数都在一定合理范围之内,我们直接枚举所有的分支.如果分支过多,超过一定阈值(可以由用户指定),我们将一个程序划分为若干个程序片段,对这些片段按照顺序逐个进行分析.对前面的片段的分析结果作为它后面程序片段的前置条件,进一步分析下面的程序.

Fig.1 The analysis framework

图 1 分析框架

框架中分为三个主要模块:性质划分模块,分割与拼接技术模块以及程序摘要实例化模块.其中性质类别的划分主要用来确认性质的分类,用户可以手动指定性质属于哪种类别,也可以通过其定义在 SMT 求解器的帮助下进行分析.分割与拼接技术用来提取程序对于递归数据结构的操作语义,产生带有归纳性质的断言,作为程序的后置条件和摘要.程序摘要实例化模块把被调用函数的摘要进行实例化,得到具体的函数调用的效果.另外还有三个预处理模块,分别为路径枚举模块、循环消除模块和函数内存踪迹检查模块.其中路径枚举模块将程序

的不同分支进行枚举,得到若干个顺序程序.循环消除模块通过循环不变式将对循环的分析简化为只包含一次循环体的程序,得到的结果依然是顺序程序.函数内存踪迹检查模块用来检查访问到的内存单元的踪迹,确保待分析的函数只会访问其参数所指向数据结构的子结构,从而确保分析的正确性.

值的注意的是,分割与拼接技术模块与程序摘要实例化模块是互相作用的,它们计算的分析结果可以作为彼此的输入,并得到最终的分析结果.对于一个特定的程序,我们首先获取程序的调用图(call graph),分析函数之间的调用关系.如果调用图中没有环,即没有递归函数调用.我们按照逆拓扑的顺序分析程序这些函数,确保被调用者在调用者之前被分析.如果图中有环,则在环上进行不动点计算,即首先将环断开形成链,忽略链尾部的那次函数的作用,然后按照逆拓扑顺序进行分析.最后再将链尾部的函数调用的结果考虑进去,并再次进行分析,直到每个函数的摘要都达到不动点.

2 递归数据结构和归纳性质

递归数据结构的多数性质是通过归纳定义的方式给出的.首先给出对于空指针(null)节点性质的定义,这是归纳定义的基础情况.然后给出非空节点的定义,且非空节点性质的定义又取决于这个节点的直接子节点(child)的性质,这是归纳定义的归纳部分.当一个程序在修改递归数据结构时,其迭代性质会被破坏而不再成立,或者需要重新进行证明.但基于对程序操作递归数据结构过程的观察,我们发现有一定的模式可以用来描述这些性质是如何变化的:1)当递归数据结构的一个子结构被修改时,子结构的特定性质没有发生改变而依然成立,此时根据递归定义,整个递归结构的性质仍然成立;2)当子结构被修改时,子结构上的特定性质发生了改变,但是很多情况下我们可以根据子结构上性质的变化直接推算出整体结构性质的变化.这两种模式可以帮助我们简化对相当一部分归纳性质的分析过程.

递归数据结构的基本元素为节点.节点包含数据(data)字段和连接(link)字段.数据字段用来保存数据,连接字段用来将自己和其它节点连接起来.递归数据结构有两个主要特征:1)它们可以通过归纳的方式定义出来,包含基础部分和归纳部分;2)其中的任意一个节点的多个子结构互不相交.最常见的递归数据结构包括链表(linked list)和树(tree),这两类数据结构是相关工作研究的重点.显然,有向无环图(directed acyclic graph)和一般意义上的图都不属于递归数据结构范畴,因此不在本文考虑的范围之内.为了简单起见,从现在开始,没有特别说明的情况下,我们简称递归数据结构为数据结构.

由节点 x 所指向的递归数据结构的归纳性质 P 一般通过以下形式定义:

$$P(x) \triangleq F(x, P(x_0), P(x_1), ..., P(x_n))$$

其中 $x_1, x_2, ..., x_n$ 是 x 的直接子节点, F 是一个将节点 x 的性质和其直接子节点的性质结合起来形成 x 的总体性质的函数.一般来讲, F 由两部分组成.一方面,定义了当 x 为空的基础情况;另一方面,定义了当 x 不为空的归纳定义.

例 2.1 二叉树 x 的高度性质定义如下 $height(x) \triangleq (x = null)?0 : \max(height(x \to l), height(x \to r)) + 1$.定义分为两部分:对于 *null* 节点来说,高度为 0.对于非 *null* 节点来说, x 的高度为其两棵子树高度的最大值加 1.在这个例子中,性质 P 为树的高度(*height*),函数 F 的签名为 $F(x, height(x \to l), height(x \to r))$.它通过综合 x 性质, $x \to l$ 的高度以及 $x \to r$ 的高度来定义 x 本身的高度.

2.1 归纳性质的模式

下面我们介绍,当递归数据结构被程序操作时,它们性质的变化往往所遵循的模式.假设一个程序作用于递归数据结构 x ,其中的若干个节点被程序所修改.为了达到程序功能,修改可能发生在节点的任何字段上面.例如,当程序需要去将一颗二叉树进行旋转时,往往是通过修改左右子树的指针字段来达到这样的目的.为了描述简单起见,我们假设 x_k 是 x 的直接子节点 $x_1, x_2, ..., x_n$ 中唯一被修改的节点.我们总结出以下两种情况.

2.1.1 局部性质的不变保证整体性质的不变

当 x_k 被程序修改,记修改后的节点为 x_k' ,并且 x_k' 上的性质 $P(x_k')$ 与 $P(x_k)$ 保持不变,此时根据定义,被修改之

后的数据结构 x' 上的性质 x 上的性质 $P(x'_k)$ 相对于 $P(x_k)$ 依然保持不变.对于所有通过归纳定义得到的性质来说,这一点都是成立的.这种模式可以表示为 $P(x'_k)=P(x_k) \Rightarrow P(x')=P(x)$.

2.1.2 局部性质的变化直接反映整体的变化

当 x_k 被程序修改,记修改后的节点为 x'_k,x'_k 上的性质 $P(x'_k)$ 往往会发生改变,我们可以通过分析知道它是如何变化的,假设 $P(x'_k)=P(x_k)\oplus\Delta$.并且此时 x'_k 局部上改变可以直接映射到整个数据结构 x' 上改变,即 x' 改变后的性质可以表示为 $P(x')=P(x)\oplus\Delta$.这种模式可以表示为 $P(x'_k)=P(x_k)\oplus\Delta \Rightarrow P(x')=P(x)\oplus\Delta$.

2.2 性质类别的划分

接下来我们给出判断一个性质是否满足第二种情况的方法.在我们的分析框架中,程序员使用包含用户自定义递归函数的公式来表示程序在某些程序点上的断言.我们要求这些用户自定义函数的定义是已知的.

给定一个性质的定义,我们首先分析出子结构 x'_k 的性质变化情况,假设可以表示为 $P(x'_k)=P(x_k)\oplus\Delta$.然后我们将 $P(x)$ 定义中的 $P(x_k)$ 替换为 $P(x_k)\oplus\Delta$,得到 $P(x')$ 的值.最后我们检查 $P(x')$ 是否等于 $P(x)\oplus\Delta$.如果相等,则性质 P 属于第二类,否则属于第一类.这个过程可以通过 SMT 约束求解器进行一定程度的自动化.如果性质过于复杂,或者由于 SMT 求解器的求解能力不够而无法得出结果,可以通过人工判断或由程序员直接给出.

如果一个性质不属于第二种分类,并且某个整体结构的子结构的性质发生了某种改变,无法由整体变化推导出局部变化,此时就需要人工参与分析过程.

3 过程内分析

在对递归数据结构归纳性质进行分析的过程中,往往会采用经典的展开与匹配技术,根据程序开始时的性质来获取程序结束时所产生的新的性质.在霍尔式程序证明过程中,一般采用断言的方式来描述前置和后置条件.对于操作递归数据结构的程序来说,这些断言往往为带有用户自定义递归谓词或函数的一阶谓词逻辑来描述递归数据结构的性质.因为谓词或函数是递归定义的,所以可以根据具体情况(递归数据结构中哪些节点的相关字段被修改)将其定义进行数次展开,并将其中出现多次的相同表达式替换成一个变量,直到递归函数或谓词中不存在字段被修改的子节点.然后使用 SMT 求解器去确认新得到的公式是否能被已知性质推出.然而,这种技术只能用于处理目标节点(即被修改节点)与根节点之间的距离是已知确定的情况.这是因为只有在距离是确定数目的情况下,我们才知道需要对递归谓词或函数进行多少次展开.为了解决这个问题,我们扩展了现有的展开与匹配技术来处理根节点与目标节点之间距离是未知的情况(只需保证目标节点从根节点可达即可).

例 3.1 图 2 的 `insert_sorted` 函数往参数 `h` 所指向的有序(由小到大)单链表中插入一个新节点 `n`,并保证新的链表(依然由 `h` 指向)的有序性.程序首先通过一个 `while` 循环找到一对指针 `p` 和 `q`,使得 `p->key<n->key` 且 `q->key>=n->key`.然后将 `n` 插入到 `p` 和 `q` 之间.我们需要证明在程序结束是 `h` 依然是有序的.但展开与匹配技术在这个例子中无法直接使用,因为在头结点 `h` 与目标节点 `p` 之间的距离是未知的(但是可以确定 `p` 从 `h` 可达),没有办法可以确定展开的次数.

```
Node* insert_sorted(Node* h, Node* n)
Requires: sll(h),sorted(h),n!=null
Ensures: sll(h),sorted(h)
{
    if (h == null) return n;
    Node *p = h, *q = h->next;
    while (q && q->key < n->key) {
        p = q;
        q = q->next;
    }
    p->next = n;
    n->next = q;
    return h;
}
```

Fig.2 A function that inserts a node into a sorted singly-linked list

图 2 一个向有序单链表中插入节点的程序

我们的方法通过如下方式处理上面这种情况.首先,我们找出被程序修改的所有节点,对于这个例子而言是 p 和 n.第二,我们分析出在程序结束时新的数据结构 h 的组成情况,对于这个例子而言,包含四个部分:从 h 到 p 的单链表片段,从 p 到 n 的单链表片段,从 n 到 q 的单链表片段以及从 q 到 *null*(尾部)的单链表,如图 3 所示.其中直线表示实线相邻的节点之间的边,点线表示经过若干个节点可达的边,虚线表示原来存在现在不存在的边.接着,我们使用展开与匹配技术去分析 n 和 p 的性质.此时能直接利用展开与匹配技术是因为以 n 为头结点的单链表发生修改的地方是节点 n 本身,距离为 0,因此是已知的.结合我们上一节讲到的性质分类,我们知道 *isslist*(判断是否为单链表的谓词)和 *issorted*(判断是否为有序的谓词)都属于第二类情况,同时也满足第一类的情况.因此根据程序语义可以知道 n 为一个有序单链表.同理,p 也是有序单链表.最后,由于从 h 到 p 的单链表片段的有序性依然成立(因为这个片段并没有被程序修改过),且 p 也是有序的,所以可以最终确定 h 依然是有序的(根据第一种情况的特点).

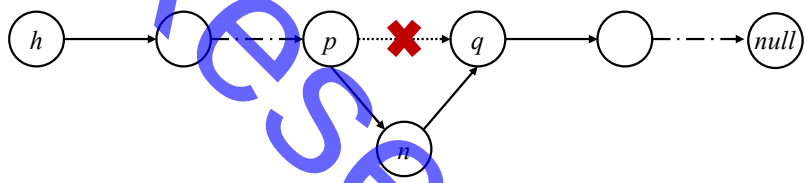

Fig.3 The process of a singly-linked list being manipulated by the program *insert_sorted*

图 3 单链表被 *insert_sorted* 程序操作的过程

可以看到,方法的基本思路是将原来的数据结构分割为若干内部没有被修改的片段,并将这些片段以新的方式或顺序拼接起来,得到新的数据结构.我们将这个方法称为分割与拼接(slicing and splicing)技术.这个技术本身也利用了展开与匹配技术,因此可以处理大部分递归数据结构的操作程序.

值的注意的是,之所以要保证这些片段内部没有被修改,是要让这些片段原来满足的性质得以保留下来.具体来讲,如果一个公式包含了递归数据结构 x 的归纳性质,当 x 中的某个节点被修改时,原本的性质被杀死,不再成立,或者还成立但是需要重新证明.此时,我们可以尝试将原有的递归性质根据其中的递归谓词或函数进行展开,将原有的性质分割到 x 的子结构上面去.当展开得到表达式中不包含字段被修改的节点的性质,展开过程就停止.这样一来展开后得到的关于子结构的性质就得以保存下来,从而可以应用到后面这些子结构重新组成一个新的递归数据结构时计算其整体性质的过程.这样的技术非常直观,也很高效,并且是采用先进 SMT 求解器辅助分析过程时. 下面我们具体介绍分割与拼接技术.

3.1 分割与拼接技术

分割与拼接技术的核心想法是在程序性质被杀死变为无效时,尽量多的保留现有的信息.保留下来的信息用来辅助生成变化之后的数据结构的性质.为了介绍分割与拼接技术,我们首先引入几个辅助性的概念.

定义 1(递归数据结构). 递归数据结构是一个二元组 $\langle N, E\rangle$,其中 N 是节点的集合,E 是边的集合.其中 N 中的任意节点的所有子结构的节点集合互不相交.

定义 2(分割点). 数据结构 $\langle N, E\rangle$ 的分割点是指 N 中相关字段被改变的节点,其中相关字段指的是对目标性质产生影响的字段.

定义 3(片段). 数据结构 $\langle N, E\rangle$ 的片段是一个二元组 $\langle N', E'\rangle$,其中 N' 是 N 的子集,使用一个二元组 $\langle p, T\rangle$ 表示,其中 p 是片段的起始节点,T 是片段的结束节点集合. N' 包含从 p 到 T 中所有可达的节点(包含 p,但不包含 T 中的节点;特别的,如果 $T=\{p\}$,片段只包含一个节点 p). E' 是 E 的子集,包含了 N' 中节点之间的边.

定义 4(子结构). 数据结构$\langle N, E\rangle$的子结构是一个二元组$\langle N'', E''\rangle$. N''为N的子集,它包含从N中的一个节点p可达的所有节点,E''包含N''中节点之间的边.

定义 5(节点集合). 节点集合是是一个递归数据结构,一个分割点,一个片段或是一个子结构所包含的节点的集合.一个数据结构的节点集合包含所有从其根节点可达的节点.一个分割点的节点集合只有一个元素,即这个节点本身.片段的节点集合包含这个片段中的节点.子结构的节点集合包含这个子结构中所有的节点.

定义 6(划分). 递归数据结构x的划分是一个三元组$\langle P, G, S\rangle$,其中P是所有分割点的集合,G是所有片段的集合,S是所有子结构的集合.它们满足以下性质:1) G和S中所有元素的节点集合互不相交;2) G和S中所有元素的节点集合的并集等于x的节点集合.

根据以上的定义,我们给出合成一个递归数据结构性质的函数$Compose(r, Q, N, P)$,它接收四个参数,分别是递归数据结构的头结点r,递归数据结构的划分Q,划分中所有子结构的性质的集合N,以及归纳性质P的定义.其中我们并没有显式地列出片段的性质,这是因为使用函数*Compose*进行计算时会将划分缩减为一个片段和若干个子结构,而这个片段的性质是不变的,可以根据r的性质直接获得.

显然,根据我们给出第 2 节的描述,我们所描述的第一类性质的合成(*compose*)的函数为:

$$P(x') = Compose(x, \langle\{x_k\}, \{\langle x, \{x_k\}\rangle\}, \{x_k\}\rangle, P(x_k'), P) = P(x),$$

第二类性质的合成函数为:

$$P(x') = Compose(x, \langle\{x_k\}, \{\langle x, \{x_k\}\rangle\}, \{x_k\}\rangle, P(x_k'), P) = P(x) \oplus \Delta.$$

数据结构的一个划分根据分割点的位置,将一个数据结构分成若干个互不相交的部分.一方面,片段和子结构中的节点并没有发生改变,因此它们上面原有的性质依然成立.另一方面,数据结构上所有的修改都发生在分割点上.根据这样的观察,我们给出分析递归数据结构归纳性质的一般方法.

给定一个递归数据结构,我们首先计算出一个顺序程序对于它的划分.其次,根据分割点在数据结构中的位置,我们以自底向上的顺序(由叶子节点到根节点,由尾部到头部)依次计算由分割点所指向的新的数据结构的性质.准确地讲,我们首先找出离叶子最近(在叶子节点和当前点之间没有其它分割点存在)的分割点,然后处理与叶子节点之间存在一个分割点的节点,直到到达根节点.当我们在处理分割点时,我们使用了传统的展开与匹配技术,因为此时被修改的节点就是分割点,两者之间的距离为 0,满足展开与匹配技术的必要条件.在这一步之后,得到的数据结构由一个片段和若干个子结构组成,而这些子结构的性质已经先被之前的过程计算出来了.最后,我们将片段的性质(没有被修改过)和子结构的性质进行拼接,形成最后得到的数据结构的性质.这个过程中包含以下三个算法.这个拼接的过程主要涉及我们在第 2 节中所讲到的性质分类,如果性质及其变化符合其中一种情况,都可以非常容易的进行合成.

算法 1 是整个框架的主要过程.其中 1 到 8 行首先找到字段被改变(只关心那些影响具体性质的字段)的节点,然后使用 *EquivExp* 函数求得节点在程序开始处的等价表达式(这个过程借助于最弱前置条件计算),接着使用 *Simplify* 函数得到他们的简化形式的表达式(主要是去除公式中判断内存是否相交的 *if-then-else* 形式).这个过程里面涉及到的技术参见文献[1,28].将得到的简化的形式的表达式放入集合 M 中.第 9 行我们通过函数*ComputePartition*(在算法 2 中介绍)计算出程序结束时新的数据结构y的划分.最后,我们使用 *SynProp* 函数来合成y的性质,这个过程主要依赖于y的根节点,y的划分和性质的定义等,这个过程在算法 3 中描述.

算法 1 分割与拼接算法

输入 C:顺序程序;x:递归数据结构根节点;P:x上的归纳性质;y:程序结束后新的数据结构

输出 程序结束后y满足的后置条件

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->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 $SynProp(y, Q, P)$;

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它在 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 到 21 行递归的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 $\langle P, G, S \rangle$;

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它的分割点.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.接着我们将这些分割点从 $Q.P$ 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它在 $Q.P$ 中节点的节点集合;
4. for each $n \in M$
5. $p \leftarrow UnfoldMatch(P, n)$;
6. $N \leftarrow N \bigcup \{p\}$;
7. $Q.P \leftarrow Q.P \setminus \{n\}$;
8. $Q.S \leftarrow Q.S \setminus \{n\}$
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

这一章我们讨论函数调用对于性质的影响.这部分主要包含两个问题:一,如何确定一个函数局部堆内存与全局堆内存之间的关系;二,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.

针对第一个问题,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,而不要考虑其它部分堆内存的变化.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.

针对第二个问题,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* 是一个处理二叉树的递归函数,其中 *bt* 是一个至少包含 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三类情况.如果第一个条件满足,则处理左分支;如果第二个条件满足,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一个函数 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它函数调用.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,我们根据当前的分析结果可以得知在函数调用前成立的性质,根据这些性质我们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 4.1 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* 是一颗二叉树节点,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* 的两颗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* 被调用了三次.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 列出的第二类情况,第 2 次调用时传入的二叉树形状属于第二类情况.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 The 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
$height(x \to right) - height(x \to left) = 1$	$height(y \to left) - height(y \to right) = 1$
$height(x \to right) - height(x \to left) = 2 \wedge$ $height(x \to right \to right) - height(x \to right \to \text{left}) = 1$	$height(y \to left) = height(y \to right)$

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*),它们的定义如表 2 所示.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)	$nsbtseg(x,\{y\}) \triangleq (x = null \vee x = y)\,?\,\varnothing : nsbt(x \to l,\{y\}) \bigcup \{x\} \bigcup nsbt(x \to r,\{y\})$
isbt (二叉树)	$isbt(x) \triangleq (x = null)\,?\,true : isbt(x \to l) \wedge isbt(x \to r) \wedge nsbt(x \to l) \bigcap \{x\} = \varnothing$ $\wedge nsbt(x \to r) \bigcap \{x\} = \varnothing \wedge nsbt(x \to l) \bigcap nsbt(x \to r) = \varnothing$
isbtseg (二叉树片段)	$isbtseg(x,\{y\}) \triangleq (x = null \vee x = y)\,?\,: true : isbtseg(x \to l,\{y\}) \wedge isbtseg(x \to r,\{y\})$ $\wedge nsbtseg(x \to l,\{y\}) \bigcap \{x\} = \varnothing \wedge nsbtseg(x \to r,\{y\}) \bigcap \{x\} = \varnothing$ $\wedge nsbtseg(x \to l,\{y\}) \bigcap nsbtseg(x \to r,\{y\}) = \varnothing$
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二类的要求,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二类的要求.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颗二叉树,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->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三种基本情况.第一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.第二种情况是,当 *x* 不为 *null* 但是 *x->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->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 中所示的三种不同前置条件下的顺序程序.

```
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: $x \neq null \wedge x \to right = null$

```
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: $x \neq null \wedge x \to right \neq null$

Fig. 7 Decomposition of the function leftRotate

图 7 leftRotate 函数的分解

我们以二叉树节点集合性质 *nsbt* 为例,它是满足我们将的第二种分类的,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].

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.实验结果如表 3 所示.

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,insert,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三个性质(节点集合、谓词和长度)在 1 秒内完成分析,对于二叉查找树在多个典型操作下的三个性质(节点集合、谓词和高度)可以在 3 秒内完成分析.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

本文提出了一个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.我们从三个方面进行研究:1)我们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)我们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)我们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,我们首先找出递归的环,将其断开(即忽略其中一个函数的调用效果),然后迭代的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三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,同时我们在第 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[11,12]提出的展开与匹配技术的技术上进行扩展,使之适应我们的具体问题.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,并进行了适合我们具体问题的定制(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 proofs)技术来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所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来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]对多年来了相关工作进行了综述性的介绍.

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,我们的方法还能够给出如果一个性质发生变化,它们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.我们方法的有点在于直观而有效.

References:

[1] Zhao J, Li X. Scope Logic: An Extension to Hoare Logic for Pointers and Recursive Data Structures. Theoretical Aspects of Computing-ICTAC 2013. Springer Berlin Heidelberg, 2013: 409-426.

[2] Reynolds J C. Separation logic: A logic for shared mutable data structures. Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on. IEEE, 2002: 55-74.

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

[4] Deutsch A. Interprocedural may-alias analysis for pointers: Beyond k-limiting. ACM Sigplan Notices. ACM, 1994, 29(6): 230-241.

[5] Kreiker J, Seidl H, Vojdani V. Shape analysis of low-level C with overlapping structures. Verification, Model Checking, and Abstract Interpretation. Springer Berlin Heidelberg, 2010: 214-230.

[6] Berdine J, Calcagno C, Cook B, et al. Shape analysis for composite data structures. Computer Aided Verification. Springer Berlin Heidelberg, 2007: 178-192.

[7] Holik L, Lengal O, Rogalewicz A, et al. Fully automated shape analysis based on forest automata. Computer Aided Verification. Springer Berlin Heidelberg, 2013: 740-755.

[8] Cousot P, Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. ACM, 1977: 238-252.

[9] Bouajjani A, Dragoi C, Enea C, et al. Abstract domains for automated reasoning about list-manipulating programs with infinite data. Verification, Model Checking, and Abstract Interpretation. Springer Berlin Heidelberg, 2012: 1-22.

[10] Berdine J, Calcagno C, O'hearn P W. Symbolic execution with separation logic. Programming Languages and Systems. Springer Berlin Heidelberg, 2005: 52-68.

[11] Pek E, Qiu X, Madhusudan P. Natural proofs for data structure manipulation in C using separation logic. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2014: 46.

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

[13] Chin W N, David C, Nguyen H H, 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.

[14] Lee O, Yang H, Yi K. Automatic verification of pointer programs using grammar-based shape analysis. Programming Languages and Systems. Springer Berlin Heidelberg, 2005: 124-140.

[15] de Boer F, Bonsangue M, Rot J. Automated verification of recursive programs with pointers. Automated Reasoning. Springer Berlin Heidelberg, 2012: 149-163.

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

[17] Itzhaky S, Banerjee A, Immerman N, Lahav O, Nanevski A, Sagiv M. Modular reasoning about heap paths via effectively propositional formulas. Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 2014: 385-396.

[18] De Moura L, Bjørner N. Z3: An efficient SMT solver. Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2008: 337-340.

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

[20] Nystrom E, Kim HS, Hwu WM. Bottom-up and top-down context-sensitive summary-based pointer analysis. Static Analysis. 2004:165-80.

[21] Itzhaky S, Bjørner N, Reps TW, Sagiv M, Thakur AV. Property-Directed Shape Analysis. In CAV 2014: 35-51.

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

[23] Rinetzky N, Sagiv M, Yahav E. Interprocedural shape analysis for cutpoint-free programs. In SAS 2005: 284-302.

[24] Hoare CA. An axiomatic basis for computer programming. Communications of the ACM. 1969 Oct 1;12(10):576-80.

[25] Hoare T. The verifying compiler: A grand challenge for computing research. Journal of the ACM (JACM). 2003 50(1):63-69.

[26] Seo S, Yang H, Yi K. Automatic construction of Hoare proofs from abstract interpretation results. Programming Languages and Systems. 2003: 230-245.

[27] Dijkstra EW. Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM. 1975 18(8): 453-457.

[28] Tang Z, Wang H, Li B, Zhai J, Zhao J, Li X. Node-Set Analysis for Linked Recursive Data Structures. InSoftware Quality, Reliability and Security (QRS), 2015 IEEE International Conference on. IEEE, 2015: 59-64.

[29] Kjolstad F, Dig D, Acevedo G, Snir M. Transformation for class immutability. In Proceedings of the 33rd International Conference on Software Engineering. ACM, 2011: 61-70.