形状图理论的定理证明

张 昱 陈意云 李兆鹏

(中国科学技术大学计算机科学与技术学院 合肥 230026)

(中国科学技术大学苏州研究院软件安全实验室 江苏 苏州 215123)

摘 要 验证操作易变数据结构的指针程序仍面临很多挑战.数据结构中严重的指针别名显著地复杂化对操作这些结构的程序的推理.为分析和验证操作易变数据结构的指针程序,文中提出了形状图逻辑.形状图是描述程序中静态声明的堆指针变量和动态分配的结构体中指针域变量的指向的一种有向图,能准确表达指针的有效性和指针之间的相等性,可用于判断两个访问表达式是否是别名.形状图逻辑是 Hoare 逻辑的一种扩展,是一种直接将形状图作为程序中指针断言集的程序逻辑.该文研究形状图的等价理论和蕴含理论以及它们的判定方法和应用.首先,把形状图及其等价规则和蕴含规则分别类比为代数项及其等式规则和重写规则,像研究代数规范的理论那样来研究形状图理论.该文定义了形状图的语法理论和语义理论,定义了形状图重写系统及其终止性、局部合流性和合流性,然后得到基于形状图重写的形状图等价判定和蕴含判定的方法.其次,提出循环不变形状图和递归函数前后形状图的自动推断方法.借助形状图理论的判定方法,该文把一个基于抽象解释的推断循环不变式的一般方法改编成推断循环不变形状图的方法.由于计算终止的递归函数总有非递归的出口,可以先通过非递归路径得到函数的后形状图的初值,然后再在递归路径上迭代求解.从而,可以像推断循环不变形状图那样来推断递归函数的前后形状图.第三,参照 Nelson-Oppen 框架,提出形状图理论和整数理论组合的一种判定方法.对易变数据结构,除了关心数据结构各节点是否连成预定的形状外,往往还关心数据在这些节点间的排列等特性,它们不能脱离易变数据结构的形状特征而单独验证.为此,所提出的组合判定方法针对这类程序的验证条件的特点,利用程序分析阶段得到的形状图对验证条件的前件中的符号断言按形状图的节点分组;然后运用整数理论为各节点推导出尽可能多的性质;最后才交由定理证明器 Z3 去自动验证.这种方式有效地避免验证条件证明过程的不终止.基于形状图逻辑以及文中的工作,我们所开发的程序验证系统原型减轻了自动定理证明器的负担,并且能验证易变数据结构上较为复杂的程序,如有序循环双向链表、二叉排序树、伸展树、树堆、二叉平衡树和 AA 树的插入和删除函数.

关键词 形状图逻辑;形状分析;程序验证;自动定理证明;循环不变式的推断

中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2016.02460

Theorem Proving for a Theory of Shape Graphs

ZHANG Yu CHEN Yi-Yun LI Zhao-Peng

(*School of Computer Science and Technology, University of Science and Technology of China, Hefei* 230026)

(*Software Security Laboratory, Suzhou Institute for Advanced Study, University of Science and Technology of China, Suzhou, Jiangsu* 215123)

Abstract Programs manipulating mutable data structures present a challenge for verification. Deep aliasing inside data structures dramatically complicates reasoning statements manipulating these structures. To analyze and verify programs manipulating mutable data structures, we proposed the shape graph logic. Shape graphs describe point-to relations of statically declared heap pointer variables and pointer fields of heap objects. They precisely express the validity of pointers and the equalities between pointers, and can be used to judge whether two access expressions

收稿日期:2014-11-14;在线出版日期:2015-05-29.本课题得到国家"八六三"高技术研究发展计划项目基金(2012AA010901)、国家自然科学基金(61170018,61229201)资助.**张 昱**,女,1972 生,博士,副教授,中国计算机学会(CCF)高级会员,主要研究方向为程序设计语言的理论和实现技术、程序验证、确定性并行编程模型与运行时系统.E-mail:yuzhang@ustc.edu.cn.**陈意云**,男,1946 年生,教授,博士生导师,中国计算机学会(CCF)高级会员,主要研究领域为程序设计语言的理论和实现技术、程序验证、软件安全等.**李兆鹏**,男,1978 年生,博士,副研究员,中国计算机学会(CCF)会员,主要研究方向为程序语言、程序验证、出具证明的编译器、自动定理证明.

are aliases or not. The shape graph logic is an extension to Hoare logic, which directly uses shape graphs as pointer assertions. This paper studies the equivalence theory and implication theory of shape graphs, as well as their decision procedures and applications. First, we investigate the shape graph theory using analogous methods to those in studying the theory of algebraic specification, where shape graphs and their equivalence rules and implication rules are analogous to algebraic terms and their equality rules and rewriting rules, respectively. Analogously, we define the syntactic theory and semantic theory of shape graphs, and then define the shape graph rewriting system as well as its termination, local confluence and confluence. Based on these definitions, we present decision methods on deciding the equality and implication of shape graphs by rewriting shape graphs. Secondly, we propose methods for automatically inferring loop-invariant shape graphs as well as the pre- and post-shape graphs of recursive functions. With the help of the decision procedures for the shape graph theory, the proposed method for inferring loop-invariant shape graphs has been adapted from a general method for inferring loop invariants based on abstract interpretation. Since a terminal recursive function has at least one non-recursive exit, the initial post-shape graphs of the function can be inferred along the non-recursive path, and then post-shape graphs can be iteratively inferred along the recursive paths. Accordingly, the pre- and post-shape graphs of recursive functions can be inferred like the method inferring loop-invariant shape graphs. Thirdly, by referencing the Nelson-Oppen combination method, we propose a decision method for the combination of the theory of shape graphs and integer theory. For mutable data structures, in addition to caring about whether nodes of a data structure form the predetermined shape, features such as data arrangement between nodes are also cared. Such features cannot be divorced from the shape feature of mutable data structures, and cannot be verified alone. Therefore, according to the characteristics of verification conditions generated from such programs, the proposed combination method uses shape graphs obtained from program analysis to group symbolic assertions by shape graph nodes; then uses integer theory to infer as many properties as possible for the nodes; and finally passes the grouped assertions and properties to the Z3 theorem prover for automatic verification. This method can effectively avoid failures to prove verification conditions. Based on the shape graph theory and the work of this paper, our prototype of program verification system alleviates the burden of automated theorem provers, and can verify programs manipulating more complex mutable data structures, such as insert functions and delete functions of sorted circular doubly linked list, binary search tree, splay tree, treap, balanced binary tree and AA tree.

Keywords　shape graph logic; shape analysis; program verification; automated theorem proving; loop-invariant inference

1　引　言

形式验证(包括抽象解释、模型检测和演绎验证等途径)是提高软件可信程度的重要方法,并在工业界已经逐步得到应用,尤其是抽象解释和模型检测.例如,空客公司在 A400M 军用运输机以及 A380 和 A350 客机的开发上,已经用形式验证取代了部分安全攸关嵌入式软件的测试.和单元测试相比,单元证明降低了投入,主要是因为它方便了维护[1].演绎验证在工业界的应用虽有限但也在拓展,达索航空公司在健壮性的形式验证方面,已有约 15%的断言是用演绎验证方式证明的[1].

在演绎验证方法中,有关自动程序验证的大部分研究按这样的途径开展:程序员提供函数前后条件和循环不变式,系统对程序采用某种正向或逆向

的演算来产生验证条件(即把对程序满足某些性质的证明转化为对一些验证条件的证明),然后用自动定理证明器来证明验证条件,如 Ynot[2]、Spec#[3]和 ESC/Java[4].有些研究依靠符号计算及其过程中的定理证明来避免验证条件的生成,如 smallfoot[5]和 jStar[6].无论用哪种策略,演绎验证方法都使用严谨的方式对软件系统进行数学推理,并且都借助定理证明软件,如定理证明器 Isabelle/HOL[7]、定理证明辅助工具 Coq①、SMT(Satisfiability Modulo Theories)求解器 Z3[8].

基于演绎推理的自动形式验证方法对自动定理证明器的能力有较高期待.首先,定理证明器在程序验证过程中频繁使用.大到验证条件的证明和循环不变式的推断,小到下标变量的别名判断和数组边界检查,都需要或可以使用定理证明器.其次,需要能够应对各种领域专用逻辑的定理证明.应用程序的代码中可能涉及多种数据类型,操作系统的代码依赖于硬件特性,为完成对这些程序的验证,需要定理证明器能够覆盖这些领域专用逻辑的定理证明.第三,需要定理证明器能处理各种理论的组合.例如面向数组类型的数组逻辑,它由下标逻辑和元素逻辑组成.对于操作数组的程序的验证,定理证明器要能证明这两种逻辑的相应理论(可能还包括等式理论和未解释函数理论)组合后形成的组合理论上的定理.组合理论的问题复杂性在于:即使两个理论分别可判定,其组合理论未必可判定[9].数组理论就是这样的例子.

本文研究操作易变数据结构(包括单/双向链表、循环单/双向链表②和二叉链表示的二叉树)的指针程序的自动验证中的一些定理证明问题.这类程序的验证非常困难.首先,难以给出表达程序性质的函数前后条件,尤其是循环不变式.因为它们涉及所操作的数据结构的结构性质(常使用带量词的命题和可达性谓词来表达)以及在数据结构中各节点内的数据的性质,而后者的描述可能紧密依赖于数据结构的结构性质,例如平衡二叉树节点的平衡因子和 AA 树节点的 level 值.其次,难以证明这类程序的验证条件,因为在验证条件中经常出现对数据结构所有节点的某种全称量化.这导致不能使用像 Nelson-Oppen 框架[9]这样仅适用于无量词理论的传统的理论组合.Z3 虽然通过使用 E-graph 匹配技术能够处理量化公式,但也难以适应复杂的量化公式.除了要用量词刻画性质以及涉及的理论组合之外,验证条件的证明可能还要用到数据结构的归纳理论.例如,对于用根节点的中序前驱(其左子树中最右下的节点)代替根节点的、删除二叉排序树根节点的函数,若要证明结果仍然是二叉排序树,则需要用到二叉排序树的两个性质:左子树中任何节点的值一定小于右子树中各节点的值,左子树中最右下节点的值大于左子树中其他节点的任何值.它们都是二叉排序树的归纳性质,但是基于演绎推理的自动定理证明器不可能从二叉排序树的归纳定义推导出这些性质,因而无法证明依赖这些性质的验证条件.

为验证操作易变数据结构的指针程序,我们提出了形状图逻辑[10].形状图是描述程序所操作的静态声明的指针型变量(简称声明指针)和动态分配的结构体中指针型域变量(简称域指针)的指向的一种有向图.它准确表达了指针的有效性(指针有效是说该指针指向已分配且尚未释放的内存块)和指针的相等性,因而可作为指针程序中指针有效性和相等性的断言.形状图逻辑就是一种直接把形状图作为程序中指针断言集的程序逻辑.它是 Hoare 逻辑的一种扩展,为指针操作语句设计了专门的推理规则,这些规则用图形方式描述指针操作语句引起的形状图上被关注部分的变化.利用形状图还可以方便地消除访问路径别名,使易变数据结构上数据性质的验证仍可用 Hoare 逻辑的规则进行推理.

本文补足文献[10]所缺少的部分理论和算法.文献[10]主要围绕程序逻辑层面,提出把形状图作为指针断言,并在此基础上设计了推理指针程序的形状图逻辑和检查形状合法性的形状系统.本文围绕作为指针断言的形状图,研究形状图的等价理论和蕴涵理论以及它们的判定方法和应用.本文的贡献有如下 3 点:

(1)提出形状图理论的一种判定方法.

本文把形状图及其等价规则和蕴涵规则分别类比为代数项及其等式规则和重写规则,像研究代数规范的理论那样来研究形状图理论.本文类似地定义形状图的语法理论和语义理论,定义形状图重写系统及其终止性、局部合流性和合流性,最后得到基于形状图重写的形状图等价判定和蕴涵判定的方法.

(2)提出循环不变形状图和递归函数前后形状

① The Coq Development Team. The Coq proof assistant reference manual(Version 8.2), 2009. URL http://coq.inria.fr

② 本文用"单/双向链表"表示非循环的单/双向链表,用"(循环)单/双向链表"表示循环和/或非循环的单/双向链表.

图的自动推断算法.

借助形状图理论的判定方法，本文把一个通过迭代推断循环不变式的一般方法[11]改编成一个推断循环不变形状图的方法；进一步设计该方法的一个变种来迭代推断递归函数的前后形状图.

(3) 提出形状图理论和整数理论组合的一种判定方法.

对易变数据结构，除了关心数据结构各节点是否连成预定的形状外，往往还关心数据在这些节点间的排列特性，它们不能脱离易变数据结构的结构特点而单独验证. 于是需要研究形状图理论和整数上线性算术理论(假定节点数据是整型)组合的判定问题. 本文针对验证条件的特点，利用程序分析阶段得到的形状图，参照 Nelson-Oppen 框架，提出形状图理论和整数上线性算术理论组合的一种判定方法.

本文第 2 节简要回顾文献[10]提出的形状图、形状图变换规则和形状图逻辑；第 3 节介绍形状图理论的判定方法；第 4 节介绍循环不变形状图的推断和递归函数前后形状图的推断；第 5 节介绍形状图理论和整数理论组合的判定方法；第 6 节是验证实例；第 7 节是和相关研究工作的比较；第 8 节是总结.

2　形状图和形状图逻辑

2.1　形状图

形状图是描述程序中静态声明指针和动态分配的结构体中域指针的指向关系的一种有向图.

形状图的顶点(也称节点)有 6 种，其图形化语法形式见图 1. 其中声明节点和结构节点分别表示静态声明指针和动态分配的结构体. null 节点和悬空节点的用意稍后会提到.

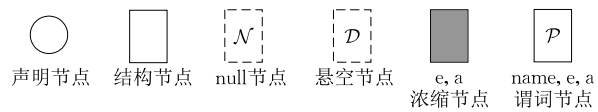

图 1　形状图的各种节点

浓缩节点是若干个结构节点和它们之间的有向边的浓缩表示，其灰色矩形下侧的表达式 e 和断言 a 分别表示被浓缩的结构节点的个数以及对 e 的取值范围的约束. 若灰色矩形下无 e 和 a，称为无约束浓缩节点，它表示被浓缩的结构节点个数任意，可以是 0 个.

谓词节点代表满足指定谓词的若干节点和它们之间的有向边，其矩形节点下侧标有谓词名 name，还可能有与浓缩节点含义一致的表达式 e 和断言 a. 指向谓词节点的有向边以及该节点下侧的 e 和 a 是该节点所代表的谓词的变元，例如图 2 给出了几种用形状图定义的数据结构，以单链表为例，图中谓词节点的变元和左侧的符号谓词 list(s, e, a)的变元是一致的.

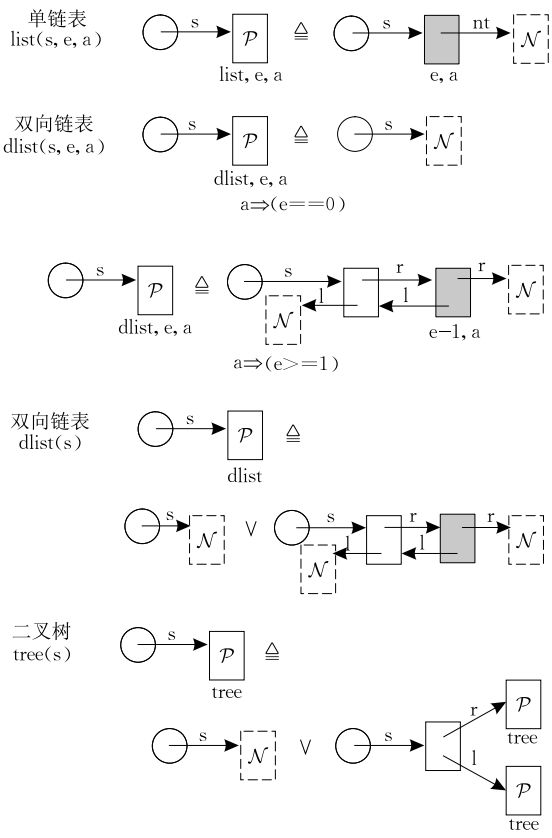

图 2　形状谓词定义的一些例子

形状图中的有向边表示声明指针和域指针的指向，指向同一个节点(悬空节点除外)的指针相等. 有向边及其连接的节点满足如下语法约束：

(1) 声明节点. 只有唯一的出边，没有入边，出边的标记就是声明指针名.

(2) 结构节点和浓缩节点. 有入边和出边，其中出边的条数与结构节点所代表的结构体的域指针个数一致，各出边的标记分别是各域指针名.

(3) null 节点、悬空节点和谓词节点. 有入边，没有出边. null 节点和悬空节点分别用来表示指向它们的有向边代表 null 指针和悬空指针.

形状图的定义：(1) 节点和有向边满足上述语法约束，各声明节点出边标记相异，且边被视为无向时则连通的图形是形状图；(2) 若形状图 $G_1, G_2, \cdots, G_n$ 的声明节点出边标记集两两相交都为空，则由逻辑合取符号 $\wedge$ 连接的 $G_1 \wedge G_2 \wedge \cdots \wedge G_n$ 也是形状图.

其中,不含符号 ∧ 的形状图 G 被称为形状子图;(3)若形状图 $G_1,G_2,\cdots,G_n$ 的声明节点出边标记集都相同,则逻辑析取符号 ∨ 连接的 $G_1 \vee G_2 \vee \cdots \vee G_n$ 也是形状图.

从文献[10]有关形状图的语义知道,一个不含析取符号并且没有浓缩节点和谓词节点的形状图是程序状态中指针型数据的图形表示,不含析取符号的一般形状图 G 则是程序状态集的图形表示. $G_1 \wedge G_2$ 中的 G_1 和 G_2 各代表程序状态的不同部分. $G_1 \vee G_2$ 中的 G_1 和 G_2 则代表不同的程序状态集. 若无特别说明,符号 G 仅表示不含符号 ∨ 的形状图.

图 2 给出基于形状图定义的单链表(域指针 nt 是下文 next 或 nxt 的简称,本文不区分这 3 个名称)、双向链表和二叉链表示的二叉树(系统还支持的循环单链表和循环双向链表的定义见文献[10]). 在图 2 中,定义式最左边的 dlist(s,e,a)等符号表示是为了便于在文中引用. 可以看出,把 dlist(s,e,a)的 2 个定义中的 e 和 a 部分略去,再用符号 ∨ 连接它们,就得到 dlist(s)的定义. 若用符号谓词表示,图 2 的二叉树定义相当于 tree(s)≜s==NULL ∨ s!=NULL ∧ tree(s→r) ∧ tree(s→l).

一个形状图的某形状子图中用点划线方框标明并满足下面条件的部分称为窗口,窗口描述形状子图上被关注的那部分,形状图的其余部分称为窗口的上下文.

(1)形状图的各节点处于窗口内或上下文中,不得与窗口的方框边界相交.

(2)窗口内各节点之间的边都位于窗口中.

(3)表达窗口中节点与上下文中节点联系的、穿越方框边界的边属于窗口,这些边的另一份副本在上下文中.

窗口 W 和上下文 $G[]$ 的匹配就是检查穿越 W 边界的边和 $G[]$ 中的副本边能否重合,重合后得到的形状图用 $G[W]$ 表示.

在形状图变换规则中出现的窗口有如下的缩写约定:对窗口中的某节点而言,若要求从窗口外指向该节点至少要有一条边(如图 3 的 p_1),则可能还有的、指向该节点的其他边(可以是 0 条)用一条标记为 p_k 的边统一表示.

窗口用来定义形状图的等价规则和蕴涵规则(见文献[10]的 2.3 节)及形状图逻辑的推理规则(见文献[10]的 3.1 节).

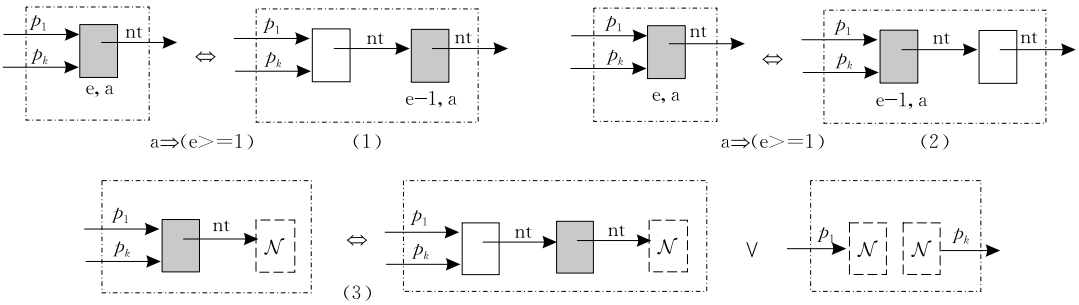

图 3 (循环)单链表的等价规则的 3 个例子

2.2 形状图的等价和蕴涵规则

等价规则是保持形状图语义等价的变换规则. 图 3 列出了在文献[10]的 2.3 节中定义的(循环)单链表的部分等价规则. 第 3 节还会列举一些等价规则.

除了等价规则外,形状图还有蕴涵规则,蕴涵规则是保持形状图语义蕴涵的变换规则. 文献[10]中 2.3 节定义的形状图蕴涵规则分成三部分.

(1)从等价规则 $W \Leftrightarrow W_1 \vee W_2$ 可得 $W_1 \Rightarrow W$ 和 $W_2 \Rightarrow W$ 两条蕴涵规则. 例如,从图 3(3)的等价规则可以得到两条蕴涵规则.

(2)若等价规则 $W_1 \Leftrightarrow W_2$ 的副条件是 $((e_1==e_2 \wedge a_1) \Rightarrow a_2) \wedge ((e_2==e_1 \wedge a_2) \Rightarrow a_1)$,则有蕴涵规则 $W_1 \Rightarrow W_2$ 和 $W_2 \Rightarrow W_1$,其副条件分别是 $((e_1==e_2 \wedge a_1) \Rightarrow a_2)$ 和 $((e_2==e_1) \wedge a_2) \Rightarrow a_1$. 例如,从图 4 的每条等价规则都可得到两条蕴涵规则.

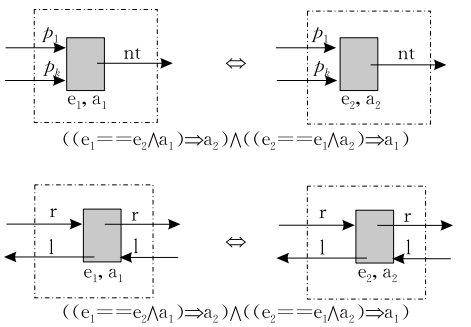

图 4 对浓缩节点的约束数据进行变换的部分等价规则

(3)将等价规则中有约束浓缩节点改成无约束浓缩节点而得到蕴涵规则. 例如,图 5 中蕴涵规则

(3)和(4)分别从等价规则(1)和(2)得到. 蕴涵方向的选择(称为定向)取决于规则的左右两边谁蕴涵谁. 以浓缩节点所代表的结构节点的个数 n 来直观解释,(3)和(4)的定向分别基于 $n==1\Rightarrow n>=1$ 和 $n>=1\Rightarrow n>=0$.

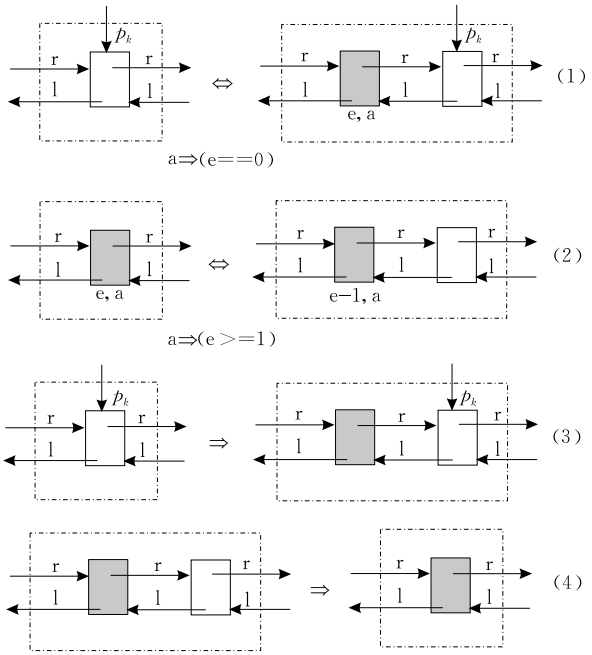

图 5　从等价规则得到的蕴涵规则

2.3　形状图逻辑的推理规则

在形状图逻辑中,有关指针操作语句的推理规则都是相应语句操作语义的图形表示. 因为推理规则体现语句执行前后形状图的变化,而形状图是程序状态的图形表示,因此形状图的变化自然就是相应语句引起的程序状态变化.

这里以指针赋值语句 $u=v$ 的部分推理规则为例,介绍形状图逻辑中指针操作语句的推理规则. 其中 u、v 代表从声明指针名出发、经若干条域指针边组成的访问路径,如 q 或 $p\rightarrow$next$\rightarrow$next.

图 6 是 u 指向 null 节点、v 指向(循环)单链表结构节点时的推理规则(v 指向浓缩或谓词节点的规则类似). 从赋值前形状图 G 到赋值后形状图 G' 的变化就是让 u 也指向 v 原来指向的节点. 图 6 花括号中的不是形状图,而只是其中被所关注的若干个窗口. 若图 6 中的 4 个窗口自左向右依次用 W_{11}、W_{12}、W_{21} 和 W_{22} 表示,则对任意能够使 $G[W_{11},W_{12}]$ 成为形状图的上下文 $G[\,]$,有推理规则

$$\{G[W_{11},W_{12}]\}u=v\{G[W_{21},W_{22}]\}$$

其中 $G[W_{11},W_{12}]$ 上的窗口 W_{11} 和 W_{12} 无重迭部分. 因此,严格说图 6 只是规则的窗口而不是规则,简单起见仍称为规则.

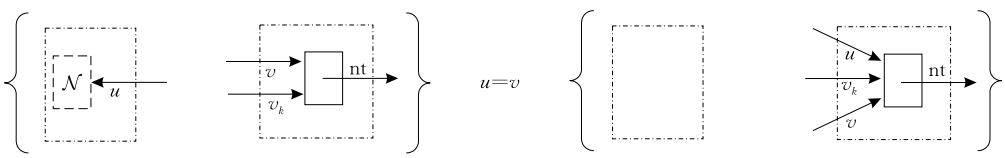

图 6　指针赋值语句的一条规则

需要注意的是,由于 u、v 代表访问路径,为便于描述推理规则,图 6 采用 2.1 节中窗口概念的一种扩展表示:将标记 u 和 v 等放在边的下方,以表示 u 和 v 等是访问路径,而不是边的标记(边的标记放在边的上方);它们要求在应用规则到某形状图时,形状图上应有相应的访问路径. 例如,使用图 6 的规则时,$G[W_{11},W_{12}]$ 要有从某声明节点到达 W_{11} 中节点的访问路径,访问路径上各边的标记顺次连接构成 u.

图 7 是 u 指向(循环)单链表结构节点、v 指向 null 节点时的推理规则(v 指向浓缩或谓词节点的规则也类似). 若赋值前只有 u 指向结构节点,则报告内存泄漏.

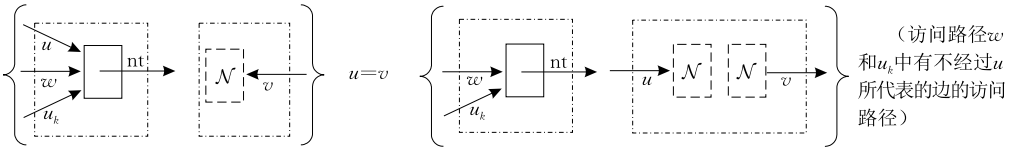

图 7　指针赋值语句的另一条规则

图 8 是两个表示单链表的形状图. 假定 head 指向的单链表至少有一个表元,程序片段

```
ptr1=head;ptr=head→next;
while(ptr!=NULL){
    ptr1=ptr;ptr=ptr→next;
}
```

的循环不变形状图就是图 8(1). 当 ptr!=NULL 为真进入循环时,利用图 3(3)的规则,把 ptr 指向的浓

缩节点展开成两种情况，其一与 ptr!=NULL 矛盾被略去，另一种情况就是图 8(2)的形状图. 把语句 ptr1=ptr 当作语句序列 dummy=ptr; ptr1=NULL; ptr1=dummy; dummy=NULL，按图 6 和图 7 的规则逐步修改图 8(2)，其中 dummy 是初值为 NULL 的虚拟局部指针变量. 引入 dummy 主要是为了避免为某些特殊的指针赋值情况设计复杂的专用规则，见文献[10]的 3.1 节. 对 ptr=ptr→next 也继续用类似的方法修改形状图，再把得到的形状图用类似图 3(3)的规则，把 ptr1 原先指向的结构节点逆向折叠进左边的浓缩节点，得到图 8(1).

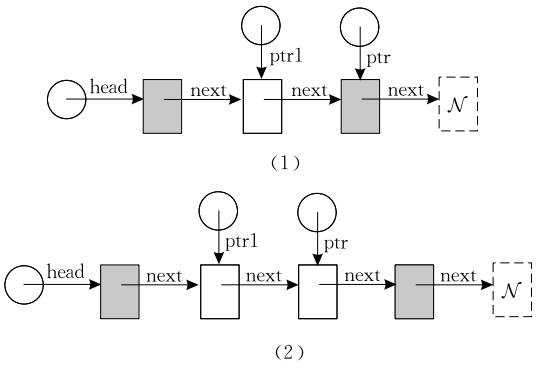

图 8 形状图的两个例子

3 形状图理论的判定方法

形状图的等价推理规则集由文献[10]中 2.3 节的形状图等价规则(不包括那些推导形状图析取 $G_1 \vee G_2$ 的规则，例如本文图 3(3))和任何等价推理系统都有的自反、对称和传递规则组成. 该规则集可用来证明形状图之间的等价性.

若等价形状图的集合 E 是封闭于可证明的，即 $\mathcal{E}\vdash G_1 \Leftrightarrow G_2$ 蕴涵 $G_1 \Leftrightarrow G_2 \in \mathcal{E}$，则把 $\mathcal{E}$ 叫做形状图的一个语法等价理论. 类似地可以定义形状图相应的语义等价理论，即 $\mathcal{E}\models G_1 \Leftrightarrow G_2$ 蕴涵 $G_1 \Leftrightarrow G_2 \in \mathcal{E}$. 由文献[10]中 2.4 节关于等价规则的性质定理，不难证明 $\mathcal{E}\vdash G_1 \Leftrightarrow G_2$ 当且仅当 $\mathcal{E}\models G_1 \Leftrightarrow G_2$. 类似地根据形状图的蕴涵规则，可以定义形状图的语法蕴涵理论和语义蕴涵理论及证明 $\mathcal{E}\vdash G_1 \Rightarrow G_2$ 当且仅当 $\mathcal{E}\models G_1 \Rightarrow G_2$.

本节介绍形状图的重写系统和这两种理论的判定方法.

3.1 形状图的等价重写系统

形状图的等价重写系统可比照代数项的重写系统[12]来讨论. 它用于形状分析和验证过程中对形状图进行等价化简.

形状图等价规则限制为单向后得到的蕴涵规则就是一条形状图重写规则. 一组形状图重写规则构成一个形状图重写系统. 不能用该系统的规则再进行归约的形状图称为最简形状图. 不存在无穷归约序列的系统称为具有终止性的系统. 可以参照项重写系统继续定义系统的临界对(critical pair)、局部合流性和合流性.

仿照 Knuth-Bendix 完备化过程[13]，可以把文献[10]中 2.3 节的等价规则集变换成一个与之等价并且终止与合流的形状图重写系统 $\mathcal{R}$.

下面先以一个简单的项重写系统为例来熟悉项重写系统的一些概念，然后说明项重写和形状图重写之间的主要异同. 例如，代数等式

$$x+0=x, x+(-x)=0, (x+y)+z=x+(y+z)$$

是属于包含 0、加和一元减的自然数等式系统中的等式，把它们从左到右定向，得到 3 条重写规则

$$x+0\to x, x+(-x)\to 0, (x+y)+z\to x+(y+z)$$

项 $x+(y+(-y))$ 的子项 $y+(-y)$ 与第 2 条规则的左部匹配，因此 $x+(y+(-y))$ 可用第 2 条规则重写成 $x+0$，再依据第 1 条规则重写为 x. 不难证明这 3 条规则构成的重写系统是终止的.

要证明合流性，需要考察临界对. 第 3 条规则左部的子项 $x+y$ 和该规则左部 $(x'+y')+z'$(加撇号以便区别)合一，得到临界对

$$\langle (x'+y')+(z'+z), (x'+(y'+z'))+z\rangle$$

该临界对中的两个项都能归约到 $x'+(y'+(z'+z))$. 可以类似地计算该重写系统的其它临界对并证明每一临界对中的两个项都能归约到同一个项，所以该系统具有局部合流性. 再由终止性可得该系统具有合流性.

在上述项重写系统中，规则的左部和右部都是项，被重写的也是项. 而形状图系统与项重写系统相比，最大的区别是：规则的左部和右部都是窗口，被重写的是形状图. 产生该区别的根源在于形状图是二维的. 基于同样的原因，合一和匹配的描述也有区别. 但是两个系统的本质概念和方法是一致的.

下面先给出节点相同和形状图相同的定义.

定义 1. 节点 n_1 和 n_2 若满足下面两个条件，则称 n_1 和 n_2 相同：

(1) n_1 和 n_2 是同类节点，并且若都是声明节点、结构节点或浓缩节点，则有同样的出边及其标记；若都是谓词节点，则谓词名相同.

(2) 若 n_1 和 n_2 是同类浓缩节点或谓词节点，且分别带 e_1 与 a_1 和 e_2 与 a_2，则必须满足 $(e_1==e_2 \wedge a_1)\Rightarrow a_2$ 并且 $(e_1==e_2 \wedge a_2)\Rightarrow a_1$. 其中，无约束浓缩

节点在此都被视为带 m 与 $m>=0$ 的浓缩节点. 下面讨论判定方法时也作如此假设.

定义 2.　形状子图 G_1 和 G_2 若满足下面两个条件，则称 G_1 和 G_2 相同.

(1) G_1 和 G_2 的节点一一对应.

(2) G_1 和 G_2 的对应节点都是相同节点，并且若它们有出边，则标记相同的出边所指向的节点是对应节点.

定义 3.　形状图 G_1 和 G_2 若满足下面的条件，则称 G_1 和 G_2 相同.

G_1 和 G_2 的形状子图一一对应并且相同.

在从形状图的等价规则构造重写系统时，可以忽略如图 4 那样仅替换浓缩节点的 e 和 a 的那些等价规则，因为定义 1 已把替换前后的浓缩节点定义为相同.

在完备化过程中，文献[10]中 2.3 节的(循环)单链表、(循环)双向链表和二叉树的等价规则按照节点数减少的方向定为重写规则，例如图 9 的等价规则(1)和(2)定向为从右向左的重写规则. 规则(1)和(2)其实是同一条规则，只要把其中一条规则中边上的标记r和l对调一下就可看出，这里写成两条是为了便于下面理解和讨论临界对. 对那些两边节点数相等的各种等价规则，可按下面的方式之一定向：

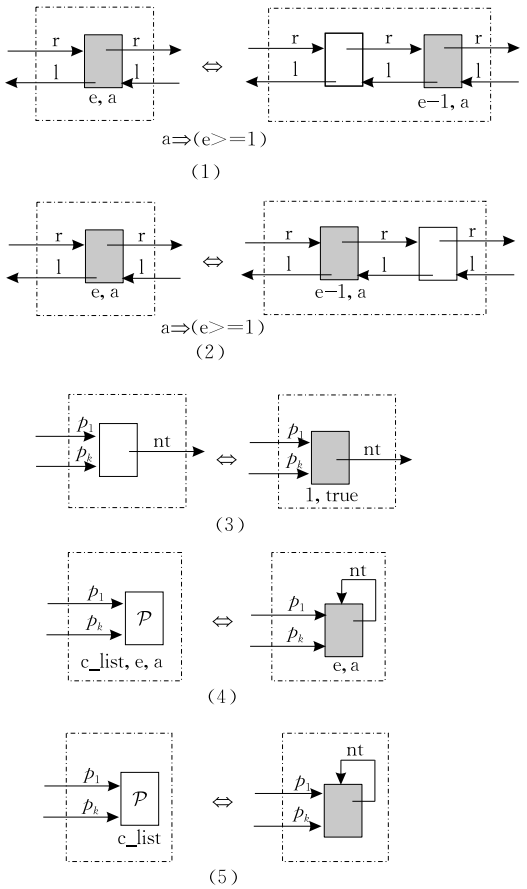

图 9　等价规则的一些例子

(1) 对于 1 个结构节点和 e 等于 1 的浓缩节点之间的等价规则，定向为向浓缩节点的重写，例如图 9(3)定向为从左向右. 按这个方向定向是便于若干个相邻的结构节点归约成浓缩节点.

(2) 对于出自链表定义的等价规则，定向为向谓词节点的重写，例如图 9 的规则(4)和(5)定向为从右向左.

形状图的归约一定会终止，因为每次归约都会使节点数减少或保持不变，而节点数保持不变的重写情况受上面两条的限定，因此归约一定会终止.

$\mathcal{R}$ 的合流性可以参照项重写系统合流性的证明方法来证明. 首先考虑临界对，例如，由图 9 的等价规则(1)和(2)得到两条重写规则 R_1 和 R_2. R_1 和 R_2 的左部(对应图 9 等价规则(1)和(2)的右部)经两种方式合一，得到 2 个临界对，分别是图 10 的(1)和(2)与图 10 的(3)和(4)，它们分别都能化简到相同的窗口. 其余的临界对也都是由于规则的窗口中有(循环)单链表、(循环)双向链表或二叉树的浓缩节点而产生的，例如对应图 3(1)和(2)的相应重写规则也会产生 2 个临界对. 很容易检查它们都可归约到同样的窗口，因而证明 $\mathcal{R}$ 是局部合流的. 再根据 $\mathcal{R}$ 的局部合流性和终止性可以证明 $\mathcal{R}$ 是合流的.

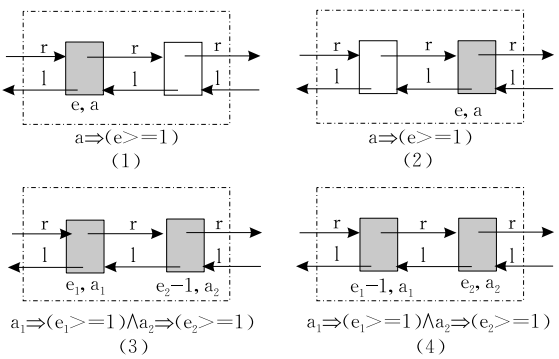

图 10　临界对的两个例子

基于形状图等价重写系统以及图 4 这样的规则，参照代数项上的证明方法，不难证明下面的定理.

定理.　形状图 G_1 和 G_2 等价，即 $G_1 \Leftrightarrow G_2$，当且仅当它们化简后的最简形状图 G_1' 和 G_2' 相同.

3.2　$G_{1,1} \vee \cdots \vee G_{1,m} \Leftrightarrow G_{2,1} \vee \cdots \vee G_{2,n}$ 的判定方法

抽象地说，判定 $G_{1,1} \vee \cdots \vee G_{1,m} \Leftrightarrow G_{2,1} \vee \cdots \vee G_{2,n}$ 可以先用 3.1 节的等价重写系统把两边所有的形状图分别化简，然后比较两边的各个最简形状图是否都能在对方找到相同的形状图. 但是 3.1 节的

等价重写系统并未考虑个数不同的形状图之间的等价性，为此需要利用文献[10]中 2.3 节里的 $W \Leftrightarrow W_1 \vee W_2$ 等价规则(如图 11)展开形状图，以期建立个数不同的形状图之间的等价性. 结合应用场合的特点，本文讨论的等价式中各形状图的声明变量集都相同，等价式的判定方法如下：

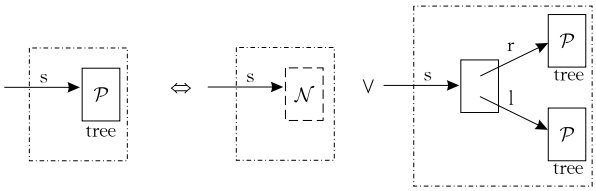

图 11 从二叉树定义得到的等价规则

(1)若 $G_{1,1}$ 和 $G_{2,1}$ 的声明变量集不同则该等价式不成立. 否则继续下面步骤.

(2)将上式中所有的形状图，用 3.1 节介绍的形状图重写系统把它们归约到最简形状图. 仍用原来的符号 $G_{1,i}(1\leq i\leq m)$ 或 $G_{2,j}(1\leq j\leq n)$ 代表各形状图的最简形状图.

(3)若某个 $G_{1,i}(1\leq i\leq m)$ 的某条尽可能长的且不经过浓缩节点的访问路径是某个 $G_{2,j}(1\leq j\leq n)$ 的某条不经过浓缩节点的访问路径的真前缀，并且若 $G_{1,i}$ 中的这条访问路径指向的节点可以用 $W\Leftrightarrow W_1\vee W_2$ 的等价规则展开，则将其展开. 反之亦然. 直到不存在这样的路径或者虽存在但不能展开为止.

为防止滥用 $W\Leftrightarrow W_1\vee W_2$ 这类等价规则，需要从访问路径的比较上加以限制. 例如，若 $G_{1,i}$ 中有不能再延长的二叉树访问路径 $p\rightarrow$l，$G_{2,j}$ 中有访问路径 $p\rightarrow$l$\rightarrow$r，且在 $G_{1,i}$ 中 $p\rightarrow$l 指向谓词节点，则可以根据图 11 的二叉树等价规则将 $G_{1,i}$ 展开，其中等价规则中的 s 是占位符，可以通配访问路径 $p\rightarrow$l 中的 l. 该规则由图 2 的二叉树定义得到.

文献[10]中 2.3 节里的所有的 $W\Leftrightarrow W_1\vee W_2$ 都像图 11 这样，右边 W_1 或 W_2 中指向谓词节点或浓缩节点的访问路径比 W 的多一个结构体的指针域，因此不会出现等价式左右两边的形状图不断交替展开而不终止的情况.

经过这样的展开，原等价式变换成 $G'_{1,1}\vee\cdots\vee G'_{1,m'}\Leftrightarrow G'_{2,1}\vee\cdots\vee G'_{2,n'}$.

(4)对每个 $G'_{1,i}(1\leq i\leq m')$，若存在某个 $G'_{2,j}(1\leq j\leq n')$，使得 $G'_{1,i}$ 和 $G'_{2,j}$ 相同，并且反之亦然，则 $G_{1,1}\vee\cdots\vee G_{1,m}\Leftrightarrow G_{2,1}\vee\cdots\vee G_{2,n}$，否则该式不成立.

显然，若上述方法中涉及的由整型线性表达式之间的关系式所构成的断言都可判定，则 $G_{1,1}\vee\cdots\vee G_{1,m}\Leftrightarrow G_{2,1}\vee\cdots\vee G_{2,n}$ 可判定.

3.3 $G_{1,1}\vee\cdots\vee G_{1,m}\Rightarrow G_{2,1}\vee\cdots\vee G_{2,n}$ 的判定方法

该判定方法用于程序分析和验证过程中对形状图蕴涵关系的判定，例如第 4 节的循环不变形状图和递归函数前后形状图的推断算法需要用到该方法. 先定义节点相容和形状图相容.

定义 4. 节点 n_1 和 n_2 若满足下面的条件，则 n_1 相容于 n_2(与定义 1 的区别仅在条件(2)).

(1)n_1 和 n_2 是同类节点. 并且若都是声明节点、结构节点或浓缩节点，则有同样的出边及其标记；若都是谓词节点，则谓词名相同.

(2)若 n_1 和 n_2 是同类浓缩节点或谓词节点，且分别带 e_1 与 a_1 和 e_2 与 a_2，则[illegible]满足 $(e_1==e_2\wedge a_1)\Rightarrow a_2$.

定义 5. 形状子图 G_1 和 G_2 若满足下面 3 个条件，则称 G_1 相容于 G_2.

(1)G_1 的节点集 N_1 到 G_2 的节点集 N_2 有一对一的函数 f，使得 $N_2-f(N_1)$ 若非空，则其中只有浓缩节点并且每个浓缩节点的 $(e==0)\wedge a$ 可满足，即这些浓缩节点可以有 e 等于 0 的情况.

(2)对任何 $n\in N_1$，n 相容于 $f(n)$.

(3)对任何 $n\in N_1$，若 n 和 $f(n)$ 的某标记相同的出边分别指向节点 n_1 和 n_2，则 n_2 经过 m 个 $(m\geq 0)$ 属于 $N_2-f(N_1)$ 的浓缩节点后所到达的 n'_2 和 n_1 满足 $f(n_1)=n'_2$.

对于图 12 的 G_1 和 G_2，G_1 的那个浓缩节点经 f 函数映射到 G_2 最右边的那个浓缩节点. 因为 G_2 左边两个浓缩节点的 m 和 k 都可以等于 0，所以 G_1 相容于 G_2.

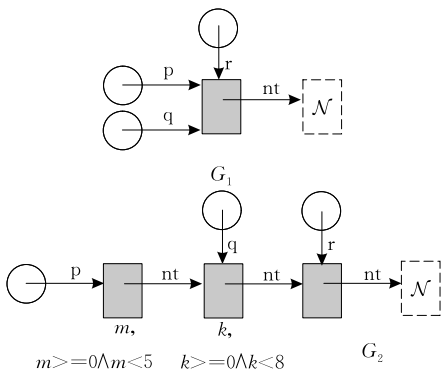

图 12 帮助理解定义 5 的两个形状图

定义 6. 形状图 G_1 和 G_2 若满足下面的条件，则称 G_1 相容于 G_2.

G_1 和 G_2 的形状子图一一对应并且 G_1 的每个形状子图相容于 G_2 中对应的形状子图.

在证明形状图之间的蕴涵时，需要考虑在用等

价重写系统化简形状图后，适当地使用形状图的蕴涵规则. 所使用的蕴涵规则是 2.2 节中第(1)类蕴涵规则，即由 $W \Leftrightarrow W_1 \vee W_2$ 得到的 $W_1 \Rightarrow W$ 和 $W_2 \Rightarrow W$；而对于第(2)和(3)类蕴涵规则，由于它们都是针对浓缩节点的 e 和 a 的规则，其效果已体现在节点相容的定义中，因而不必作为重写规则.

$G_{1,1} \vee \cdots \vee G_{1,m} \Rightarrow G_{2,1} \vee \cdots \vee G_{2,n}$ 的判定方法可通过略微修改 3.2 节的等价式判断方法中的(3)和(4)两点即可得到：

(3) 若某个 $G_{2,j}$ ($1 \leq j \leq n$) 的某条尽可能长的且不经过浓缩节点的访问路径是某个 $G_{1,i}$ ($1 \leq i \leq m$) 的某条不经过浓缩节点的访问路径或其真前缀，但不是指向同类节点，则尝试使用上面提到的蕴涵规则对 $G_{1,i}$ 进行重写.

例如，若图 11 规则右部两个窗口之一和左部窗口分别出现在 $G_{1,i}$ 和 $G_{2,j}$ 中的上述位置，则使用相应蕴涵规则重写 $G_{1,i}$，把窗口中内容归约成谓词节点.

(4) 对每个 $G_{1,i}$ ($1 \leq i \leq m$)，若存在某个 $G_{2,j}$ ($1 \leq j \leq n$)，使得 $G_{1,i}$ 相容于 $G_{2,j}$，则 $G_{1,1} \vee \cdots \vee G_{1,m} \Rightarrow G_{2,1} \vee \cdots \vee G_{2,n}$，否则该蕴涵式不成立.

4　循环不变形状图和递归函数前后形状图的推断

本节应用第 3 节的形状图理论判定方法来推断循环不变形状图. 该推断方法的一个变种可用来推断递归函数的前后形状图.

形状图是指针有效性和指针相等性断言的集合的图形表示，循环不变形状图就是循环不变式中有关指针断言的那部分不变式，函数前后形状图就是函数前后条件中有关指针的那部分断言.

对易变数据结构的处理，循环语句通常完成下面功能之一：

(1) 寻找数据结构中的操作位置或依次对各节点上数据进行操作. 这时，除了某些声明指针的指向在节点间移动外，形状图没有其他变化.

(2) 循环体的每次迭代都在对数据结构进行节点插入或删除. 循环体执行过程中虽然破坏了数据结构的形状，但每次到达循环体结束点时，形状得到恢复，仅节点个数发生变化.

(3) 循环体的每次迭代都在对数据结构某些节点的边的指向进行调整，并且有可能出现这样的情况：每次迭代结束时形状并未得到恢复，但在整个循环执行结束时形状得到恢复. 例如文献[10]中例 2 使用的双向链表倒置算法.

对于这类循环，在循环体的每条执行路径上，声明指针在形状图上的移动引起的形状图变化主要体现在浓缩节点所代表的节点个数的变化上.

4.1　循环不变形状图推断算法的框架

根据基于抽象解释的循环不变式推断框架[11]，循环不变形状图的推断算法框架如图 13. 为表达简单起见，本节用字母 $\mathcal{G}$ 表示可以出现形状图析取的形状图.

对于循环 while (B) S
(1) 计算循环前条件 $\mathcal{G}_0 = \mathcal{G}_{pre}$. $i = 0$.
(2) 根据形状图逻辑的规则计算 $\mathcal{G}'_{i+1}$，使得 $\{\mathcal{G}_i \wedge B\} S \{\mathcal{G}'_{i+1}\}$.
(3) 应用抽象规则计算 $\mathcal{G}_{i+1}$，使得 $\mathcal{G}'_{i+1} \Rightarrow \mathcal{G}_{i+1}$.
(4) 若 $\mathcal{G}_{i+1} \Rightarrow \mathcal{G}_0 \vee \cdots \vee \mathcal{G}_i$，则 $\mathcal{G}_0 \vee \cdots \vee \mathcal{G}_i$ 是循环不变形状图；否则，$i = i+1$，转(2).

图 13　循环不变形状图推断的算法框架

下面对图 13 的算法框架做一些解释：

(1) 算法第(2)步根据形状图逻辑的规则计算 $\mathcal{G}'_{i+1}$，$\mathcal{G}_i \wedge B$ 代表形状图 $\mathcal{G}_i$ 和符号断言 B 的合取[11]. 若碰到引起内存泄漏的操作、悬空指针或 null 指针脱引用(dereference)操作，则无规则可用，报告错误.

(2) 算法第(3)步的抽象是指：对形状图上随循环迭代次数变化而变化的浓缩节点(指其代表的结构节点数随迭代次数变化而变化)，把其代表结构节点数的表达式换成能概括多次(甚至全部)迭代情况的表达式. 应用的抽象规则是采用文献[10]中第 2.3 节的蕴涵规则.

(3) 算法第(4)步 $\mathcal{G}_{i+1} \Rightarrow \mathcal{G}_0 \vee \cdots \vee \mathcal{G}_i$ 的证明是采用 3.3 节的判定方法.

以 2.3 节的那段单链表代码(假定至少有一个表元)的循环不变形状图(图 8(1))的推断为例. 各程序点的形状图见图 14，图中 j 是形状分析系统引入的虚拟变量. 第 1 次迭代后 $G_1 \Rightarrow G_0$ 不成立，但第 2 次迭代后能证明 $G_2 \Rightarrow G_0 \vee G_1$，因此 $G_0 \vee G_1$ 是循环不变形状图. 由于 $G_0 \Rightarrow G_1$，因此 $G_0 \vee G_1$ 可简化为 G_1. 由于没有程序变量可以替代 G_1 中的虚拟变量 j，则略去 j，得到结果就是图 8(1).

4.2　算法终止的讨论

图 13 算法终止取决于下面 3 点：

(1) 算法第(3)步的抽象能够成功.

(2) 算法第(4)步的蕴涵式的证明因得到结果而终止或因形状检查发现错误而终止.

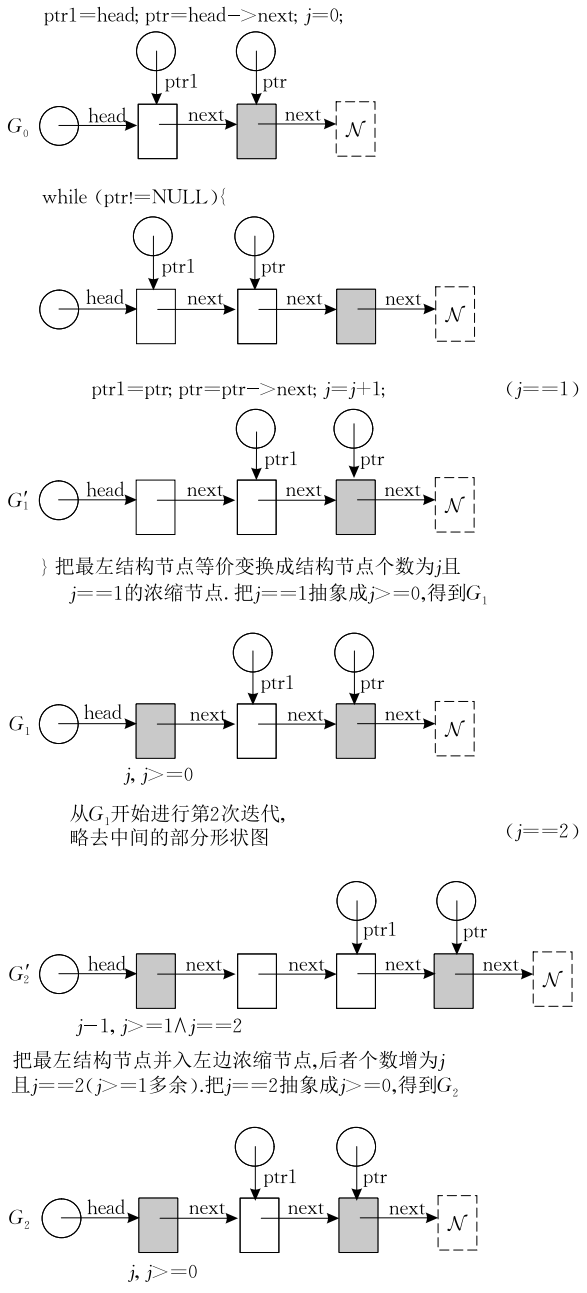

图 14 推断循环不变式的迭代过程

(3)若形状检查没有发现错误,算法第(4)步的蕴涵式最终会成立.

该算法终止的理由概述如下:

(1)本文为循环体 S 的每条执行路径都安排累计该路径执行次数的虚拟变量.选择一种合适的虚拟变量命名方式,能应对出现并列和嵌套循环的情况以及循环体中出现并列和嵌套条件语句的情况.在增加虚拟变量时,还要把表达虚拟变量之间关系的等式或不等式加到相关形状图中.

通过增设虚拟变量,结合下面(2)有关浓缩节点的 e 表达式为线性的讨论,算法第(3)步的抽象就能够成功,即可以把浓缩节点所概括的节点数抽象为用含相应虚拟变量的线性表达式来表示.

(2)从第 3 节知道,形状图蕴涵关系判定方法的终止性依赖于判断浓缩节点之间蕴涵与否的$(e==e' \wedge a) \Rightarrow a'$的证明的终止性.下面是相关的分析.

对于循环体 S 的任意一条执行路径,对于除第 1 次迭代计算以外的其他各次迭代计算的结果形状图,若与上次迭代计算的结果形状图相比,某浓缩节点所代表的节点数发生变化,则再次迭代计算引起的变化与本次迭代计算的变化相同.若循环前某浓缩节点代表 $ax+b$ 个节点,第 1 次迭代时,个数变化是 d,随后各次迭代的个数变化是 c,则该路径执行 $y(y>1)$ 次结束时,该浓缩节点代表的节点数是 $(ax+b)+(cy+d)$,由于 $(ax+b)$ 在循环执行过程中不变,因此在图 13 算法中证明形状图之间的蕴涵时可当成常量.这样,在每次迭代计算过程中,浓缩节点所代表节点的个数本质上可用该路径的虚拟变量的线性表达式来表示,这就保证了该浓缩节点的表达式 e 维持为线性.类似的分析可知 a 中的整型表达式也是线性.这就保证$(e==e' \wedge a) \Rightarrow a'$的证明会终止.图 14 例子的迭代过程体现了这里所说的对浓缩节点的抽象.

另一方面,在有 $p=p\rightarrow$next 这样指针赋值的情况下,除第 1 次迭代计算外,若一次迭代计算的结果形状图中出现无声明节点指向的结构节点,但又不能折叠成(或并入相邻)浓缩节点.则表明该结构节点的出边的指向不符合数据结构定义的要求,先前关于循环语句的 3 种用法中不会出现这样的情况.因此,若经抽象后,相邻 2 个声明指针所指向的节点之间的结构和浓缩节点的个数大于 3(简称节点个数约束),则在循环体结束点的形状检查[10]会报告错误并终止迭代.

例如,对于各节点都有指针域 l 和 r 的双向非空链表,用循环依次将各节点的 l 域置为 NULL 且 r 域的指向不变.代码如下:

```
p=head;
while(p->r!=NULL){
  p=p->r; p->l=NULL;
}
```

这相当于废弃 l 域,用 r 来构成一个单链表.图 15(1)是循环入口的形状图,图 15(2)是经过 3 次迭代的形状图.本算法无法把已经遍历过的结构节点折叠为浓缩节点,因为双向链表没有相应的规则.形状检查会因两个相邻声明节点超出节点个数约束而报告错误.

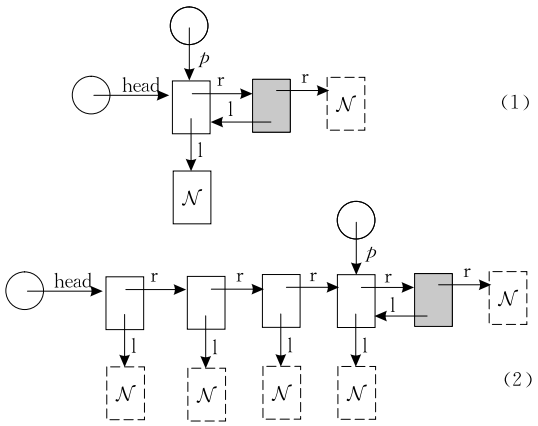

图 15　循环不变形状图推断过程报告错误的例子

(3)由于声明指针的个数有限，再加上(2)提到的节点个数约束，在循环体结束点能形成的不等价形状图有限. 若未见报告程序错误，则一定存在 i，使得 $\mathcal{G}_{i+1}\Rightarrow\mathcal{G}_0\vee\cdots\vee\mathcal{G}_i$ 成立.

4.3　递归函数前后形状图的推断

对于非递归函数，可以从函数前形状图自动推断函数后形状图. 但该方法难以用到有指针型形参且返回值是指针的递归函数，因为推断递归函数前后形状图时要用到该函数的前后形状图.

若仅有指针型的形参，或仅有指针型的返回值，根据形状系统对函数调用点和返回点的形状约束，函数的递归性并没有给形状分析带来困难. 倘若形参和返回值都是指针型，则虽有形状系统的约束，在函数返回点也不能确定实参指针和返回指针之间的关系，因而不能确定函数后形状图. 关注实参指针是因为它的值在被调用函数中虽不变，但其指向的节点在数据结构中的位置可能会变. 为此，函数的前形状图增设虚拟变量来表示实参指针，它与形参指针指向同一个节点，并据此来推断函数后形状图.

可以像推断循环不变形状图那样迭代推断函数后形状图. 因为计算会终止的递归函数总有非递归的出口，可以先通过非递归路径得到函数后形状图的初值，然后再在递归路径上迭代求解.

对于直接递归函数，若函数体中的循环语句里面没有递归调用(通常如此)，则函数前后形状图的推断方法概述如下：

(1)通过程序分析，确定函数体中非递归执行路径和递归执行路径的语句序列. 这里所说的执行路径把无递归调用的循环语句看成一个语句，把两个分支都有(或都无)递归调用的条件语句也看成一个语句，以减少要考虑的路径数. 为叙述方便，假定非递归和递归的执行路径各一条，语句序列分别是 S_{non} 和 S_{rec}.

(2)从函数前形状图 $\mathcal{G}_{entry}$ 和 S_{non} 通过形状图逻辑的规则得到 $\mathcal{G}_0$，使得 $\{\mathcal{G}_{entry}\}S_{non}\{\mathcal{G}_0\}$. 令 $i=0$.

(3)根据形状图逻辑的规则计算 $\mathcal{G}'_{i+1}$，使得 $\{\mathcal{G}_{entry}\}S_{rec}\{\mathcal{G}'_{i+1}\}$. 其中 $\mathcal{G}_{entry}$ 和 $\mathcal{G}_i$ 作为 S_{rec} 中出现的递归函数调用的前后形状图.

(4)应用抽象规则计算 $\mathcal{G}_{i+1}$，使得 $\mathcal{G}'_{i+1}\Rightarrow\mathcal{G}_{i+1}$.

(5)若 $\mathcal{G}_{i+1}\Rightarrow\mathcal{G}_0\vee\cdots\vee\mathcal{G}_i$，则 $\mathcal{G}_0\vee\cdots\vee\mathcal{G}_i$ 是函数后形状图；否则，$i=i+1$，转(3).

除了推断递归函数的函数前后条件需先分析函数的非递归和递归的执行路径外，本算法与图 13 算法没有本质上的区别，把本算法的第(2)到(5)步与图 13 算法的第(1)到(4)步比较就可以看出这一点.

5　形状图理论和整数理论组合的判定方法

在程序验证器原型中，验证条件生成器所产生的验证条件的一般形式是：

$$(G_{1,1}\wedge Q_{1,1})\vee\cdots\vee(G_{1,m}\wedge Q_{1,m})\Rightarrow(G_{2,1}\wedge Q_{2,1})\vee\cdots\vee(G_{2,n}\wedge Q_{2,n})$$

其中 Q 代表符号断言的合取，并且其中没有指针相等性断言和有效性断言.

验证条件的证明是形状图理论和整数线性算术理论的组合的判定问题. 首先，形状图之间蕴涵的证明要根据形状图理论. 其次，Q 中像 $p\rightarrow d<q\rightarrow\mathrm{nxt}\rightarrow d$ 这样的断言，需要根据形状图来判断 $p\rightarrow d$ 和 $q\rightarrow\mathrm{nxt}\rightarrow d$ 分别是属于哪个节点的数据.

注意，在形状图中，浓缩节点和谓词节点可能附带 e 和 a. 这表明形状图理论内部涉及线性整数算术理论. 它与本节讨论的理论组合的判定问题没有联系，因为系统原型限定 e 和 a 只依赖整型声明变量，不依赖于节点的数据. 因为 e 和 a 是决定该浓缩节点或谓词节点所代表的链表(段)的长度、或者所代表的二叉树片段的高度的表达式和断言，通常在实际编程中不会把决定某种数据结构的长度或高度的信息存放在该数据结构节点的域中.

本节讨论组合理论的两种证明方式.

5.1　易变数据结构上被关注性质的描述

通常关注易变数据结构 3 个层次的性质：

(1)各个节点是否连接成预定的形状

该性质的验证在形状分析阶段完成，程序员只要像声明类型那样声明形状即可. 这仅是形状图理论上的问题.

(2)与形状有关的性质

例如链表的长度、树的高度、二叉树的两边是否平衡和节点之间的可达性等.这些性质很容易归纳定义,虽然循环链表的长度的归纳定义会略显复杂.

(3)与各节点上数据排列有关的性质

当数据需按序排列时,一个数据存放到哪个节点依赖于对数据结构上节点的定序.线性链表上节点之间的定序比较简单,二叉树上节点之间有多种定序方式.这样,数据是否有序的比较必定涉及数据所属节点在数据结构上的位置.

在定义上述性质时,比较简单的方式是用以指针为参数(可能还有其他参数)的归纳谓词.另一种方式是先定义节点之间的可达关系,然后基于此再使用量词来定义所需的性质.用这两种方式描述性质都比较有利于对操作易变数据结构的递归程序的验证.

本文的系统原型针对形状图上浓缩节点的特点,还提供使用量词来概括数据性质的方式,以方便操作易变数据结构的循环程序的验证.例如有序单链表可定义为

$\mathrm{SortedList}(\mathrm{Node}^{*}\ p)\triangleq$

$\exists m:\mathbb{N}.(p(\rightarrow\mathrm{nxt})^{m}==\mathrm{NULL}\wedge$

$\forall i:1..m-1.p(\rightarrow\mathrm{nxt})^{i-1}\rightarrow d<=p(\rightarrow\mathrm{nxt})^{i}\rightarrow d)$

其中 Node 类型由 typedef struct node{int d; struct node* nxt}Node 给出,其中 $p(\rightarrow\mathrm{nxt})^{i}$ 代表 $p\rightarrow\mathrm{nxt}\cdots\rightarrow\mathrm{nxt}$,共 i 个"→nxt".在该定义中,量词的约束变元出现在访问路径的上角标中.这种直接基于编程语言中访问路径语法的定义显得比较直观.

5.2 符号断言语言

为便于下面的讨论,将系统原型所使用的断言语言的抽象语法描述在图 16.几点说明如下:

$id\in Identifier$(程序中的标识符集)

$tn\in Sidentifier$(程序中结构体类型的名字集)

predicate definition

$pd::=$ **predicate** $id(pp_1,\cdots,pp_k)=a$

predicate parameter

$pp::=t\ id$

parameter type

$t::=int\mid tn*$

property theorem

$pt::=$ **theorem** $a_1\Rightarrow a_2\mid$ **theorem** $a_1\Leftrightarrow a_2$

assertion($boolexp$ 是布尔表达式,其产生式略去)

$a::=boolexp\mid\forall id:d.a\mid\exists id:d.a\mid(a)$

domain($intexp$ 是整型表达式,其产生式略去)

$d::=\mathbb{Z}\mid\mathcal{N}\mid\mathcal{N}^{+}\mid intexp_1\cdots intexp_2$

l-value

$lv::=\cdots\mid lv(\rightarrow id)^{intexp}$

(左值 lv 是各类型 exp 的一种选择)

图 16 符号断言语言的抽象语法

(1)图 16 前一部分是自定义谓词以及这些谓词之间性质定理的语法,后一部分是断言的语法.谓词应用可出现在布尔表达式中,因而可出现在断言中.

(2)程序员用图 16 的文法编写的断言中出现的变量,若不是程序变量,则称之为逻辑变量.

(3)量词约束变元的论域只能是整数域或它的某个区间,因而禁止使用指针型约束变元.在需要由存在量词来约束指针型变元的地方,可以用指针型逻辑变量来代替.

下面用有序单链表来解释为什么需要提供性质定理.有序单链表的归纳定义如下:

$\mathrm{SortedList}(\mathrm{Node}^{*}\ p)\triangleq$

$p==\mathrm{NULL}\vee p\rightarrow\mathrm{nxt}==\mathrm{NULL}\vee$

$p\rightarrow d<=p\rightarrow\mathrm{nxt}\rightarrow d\wedge\mathrm{SortedList}(p\rightarrow\mathrm{nxt})$

在一个子句中,若出现以 p 为前缀的访问路径,则默认有断言 $p!=\mathrm{NULL}$.SortedList 的定义从表尾向表头方向归纳.但是,在遍历有序单链表时,例如图 17 单链表插入函数中的循环代码(暂且不关心图 17 中的断言),其遍历方向是从表头向表尾,因此还需要用下述表段谓词和性质定理.

$\mathrm{SortedListSeg}(\mathrm{Node}^{*}\ p,\mathrm{Node}^{*}\ q)\triangleq$

$p==q\wedge q!=\mathrm{NULL}\vee$

$p\rightarrow d<=p\rightarrow\mathrm{nxt}\rightarrow d\wedge$

$\mathrm{SortedListSeg}(p\rightarrow\mathrm{nxt},q)$

和

$\mathrm{SortedListSeg}(p,q)\wedge q\rightarrow d<=q\rightarrow\mathrm{nxt}\rightarrow d$

$\Leftrightarrow\mathrm{SortedListSeg}(p,q\rightarrow\mathrm{nxt})$

这个性质定理保证已扫描过的表段再加入下一个被扫描的节点,结果仍然是表段.

SortedList 和 SortedListSeg 两个谓词之间的如下性质定理也是验证时不可缺的:

$\mathrm{SortedListSeg}(p,q)\wedge\mathrm{SortedList}(q)\Rightarrow$

$\mathrm{SortedList}(p)$

上述 2 个性质定理都需要程序员提供,因为它们的证明需要基于表段的长度进行归纳,基于演绎推理的自动定理证明器难以发现这样的性质.

SortedList 谓词出现在函数的前后断言中,而 SortedListSeg 谓词出现在循环不变式中.后者的定义只要能支持函数前后条件的证明就足矣,没有必要选择最一般的定义,即把被验证函数中不会出现的情况也囊括在内,以免给验证增添不必要的麻烦.

在断言文法及其类型系统的基础上,还要限制断言中的整型表达式必须是线性表达式.

```
typedef struct node{struct node* nxt;int d;}Node; /*singly-linked list*/
/*@assertion m==length(head,nxt)∧oldhead==head∧i:1..m−1.(head(→nxt)^{i−1}→d<=head(→nxt)^i→d)
  oldhead 是为记住实参指针的值而引入的逻辑变量*/
Node* insert(Node* head,int data){
  Node* ptr;Node* ptr1;Node* p;int j;
  p=malloc(Node*);p→d=data;p→nxt=NULL;
  if(head==NULL){head=p;
  }else if(p→d<=head→d){p→nxt=head;head=p;
  }else{
    ptr1=head;ptr=head→nxt;j=1;
    /*@loop_invariant ptr!=NULL∧∀i:1..j−1.(head(→nxt)^{i−1}→d<=head(→nxt)^i→d)∧ptr1→d<=ptr→d∧
          ptr1→d<p→d∧j>=1∧oldhead==head∧∀i:1..m−j−1.(ptr(→nxt)^{i−1}→d<=ptr(→nxt)^i→d)∨
       ptr==NULL∧∀i:1..j−1.(head(→nxt)^{i−1}→d<=head(→nxt)^i→d)∧ptr1→d<p→d∧j>=1∧oldhead==head*/
    while((ptr!=NULL)&&(ptr→d<p→d)){ptr1=ptr;ptr=ptr→nxt;j=j+1;}
    p→nxt=ptr1→nxt;ptr1→nxt=p;
  }
  return head;
}/*@assertion length(head,nxt)==m+1∧∀i:1..m.(head(→nxt)^{i−1}→d<=head(→nxt)^i→d)∧oldhead==NULL∨
  head→d<=oldhead→nxt→d∧oldhead==head→nxt∧length(oldhead,nxt)==m∧
                                        ∀i:1..m−1.(oldhead(→nxt)^{i−1}→d<=oldhead(→nxt)^i→d)∨
  length(head,nxt)==m+1∧∀i:1..m.(head(→nxt)^{i−1}→d<=head(→nxt)^i→d)∧oldhead==head  */
```

图 17　单链表的插入函数

5.3　验证条件的证明方法

对于验证条件

$$(G_{1,1}\wedge Q_{1,1})\vee\cdots\vee(G_{1,m}\wedge Q_{1,m})\Rightarrow (G_{2,1}\wedge Q_{2,1})\vee\cdots\vee(G_{2,n}\wedge Q_{2,n}),$$

在形状分析阶段已经证明了

$$G_{1,1}\vee\cdots\vee G_{1,m}\Rightarrow G_{2,1}\vee\cdots\vee G_{2,n},$$

即对每个 $G_{1,i}(1\leq i\leq m)$，存在 $G_{2,j}(1\leq j\leq n)$，使得 $G_{1,i}\Rightarrow G_{2,j}$. 现在仅需要证明：对每个 $G_{1,i}\wedge Q_{1,i}$，存在 $G_{2,j}\wedge Q_{2,j}$（其中 $G_{1,i}\Rightarrow G_{2,j}$ 已经证明），使得 $G_{1,i}\wedge Q_{1,i}\Rightarrow Q_{2,j}$. 可把 $G_{1,i}$ 看成 $Q_{1,i}\Rightarrow Q_{2,j}$ 的证明环境，写成 $G_{1,i}\triangleright Q_{1,i}\Rightarrow Q_{2,j}$. 实际证明时，还有程序员提供的谓词定义和性质定理可用，用字母 T 表示，因此下面仅关注 $G,T\triangleright Q\Rightarrow Q'$ 的证明方法.

$G,T\triangleright Q\Rightarrow Q'$ 的证明有两种方式：第 1 种是常规的方式，把形状图理论和整数理论组合的判定转换为整数理论上的判定. 它通过引入一些未解释函数，把 G 转换成符号断言 P，然后把 $\neg(P\wedge T\wedge Q\Rightarrow Q')$ 交给 Z3. 具体把形状图转换为符号断言的方法见文献[14]. 这种方式的缺点是，在推理 Q 中涉及节点数据的断言时，指针符号断言 P 不能像形状图 G 那样提供对形状的全局把握，从而某些验证条件用 Z3 证不出. 举例说明如下：

例 1.　单链表的插入函数，代码和断言见图 17. 该例的循环不变形状图是根据 4.1 节的算法自动推断的. 为可读起见，断言都出现在注释中. 图 18 是 return 语句之前程序点形状图 $G_{1,1}\vee G_{1,2}\vee G_{1,3}\vee G_{1,4}$（分别对应只有新插入节点、插在表头、插在表中和插在表尾 4 种情况）中的 $G_{1,3}$，其中 head′ 是为表示实参指针而增设的虚拟变量，对应的 $Q_{1,3}$ 是下述符号断言，其中有序性用全称量化断言表示：

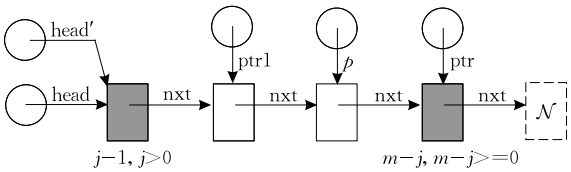

图 18　return 语句前形状图的一种情况

$$\forall i:1..j-1.(\mathrm{head}(\rightarrow\mathrm{nxt})^{i-1}\rightarrow d<=\mathrm{head}(\rightarrow\mathrm{nxt})^{i}\rightarrow d)\wedge \mathrm{ptr1}\rightarrow d<p\rightarrow d\wedge p\rightarrow d<=\mathrm{ptr}\rightarrow d\wedge j>=1\wedge$$
$$\forall i:1..m-j-1.(\mathrm{ptr}(\rightarrow\mathrm{nxt})^{i-1}\rightarrow d<=\mathrm{ptr}(\rightarrow\mathrm{nxt})^{i}\rightarrow d)$$

$G_{1,3}$ 转换成的符号断言如下：

$$\mathrm{head}==\mathrm{head}'\wedge\mathrm{ptr1}==\mathrm{head}(\rightarrow\mathrm{nxt})^{j-1}\wedge p==\mathrm{ptr1}\rightarrow\mathrm{nxt}\wedge\mathrm{ptr}==p\rightarrow\mathrm{nxt}\wedge \mathrm{ptr}(\rightarrow\mathrm{nxt})^{m-j}==\mathrm{NULL}\wedge \mathrm{length}(\mathrm{head},\mathrm{nxt})==j-1+2+m-j$$

由 $G_{1,3}$ 转换得到的断言是一些有关有效指针相等、指针等于 NULL 和链表长度的断言，它们是形状图所包含信息的符号表示. 其他信息，例如 $p==\mathrm{head}(\rightarrow\mathrm{nxt})^j$ 可以从这些断言推导. 显然，这些断言和 $Q_{1,3}$ 的合取能蕴涵函数后条件中的第 3 种情况（即图 17 中的最后一行，记为 $Q_{2,3}$）. 把这些符号断言转换成 Z3 能接受的形式的转换方法见文献[14].

用 Z3 证明不了这样的蕴涵. 原因是在符号断言中,Z3 难以发现 $Q_{1,3}$ 中 4 个断言的前后衔接次序. 若有图 18 的形状图,通过把这 4 个断言分别标注到相关节点上,则很容易发现它们合并后就是 $Q_{2,3}$ 中的全称断言.

采用第 1 种方式的另一个问题是,若与某个归纳谓词定义相对应的某条展开规则对验证来说是需要的,则容易引起 Z3 不停地用该展开规则去展开断言中出现的谓词应用,导致证明不终止.

第 2 种证明方式是把形状图直接用于 $G, T \triangleright Q \Rightarrow Q'$ 的证明. 参照 Nelson-Oppen 框架[10],本文提出的 $G, T \triangleright Q \Rightarrow Q'$ 的证明方法概述为如下 3 步:

(1)将 Q(形式为 $Q_1 \wedge \cdots \wedge Q_m$)中所有与节点相关的 $Q_j(1 \leq j \leq m)$ 都标注在 G 中相关的节点上.

① 相关的节点是指断言中的指针型访问路径(包括整型访问路径的最长指针型前缀)所到达的节点. 例如,例 1 中 $Q_{1,3}$ 的断言 $\text{ptr1} \rightarrow d <= p \rightarrow d$ 标注在图 18 中 ptr1 和 p 指向的节点上,断言 $p \rightarrow d <$ $\text{ptr} \rightarrow d$ 标注在图 18 中 p 和 ptr 指向的节点上.

② 全称断言经常标注在某个浓缩节点上. 若在一个全称断言中,约束变元的取值集合使得该断言与多个节点相关,则把该断言同时标注在这些节点上. 例 1 中 $Q_{1,3}$ 的全称断言 $\forall i: 1..m-j-1.(\text{ptr}(\rightarrow \text{nxt})^{i-1} \rightarrow d <= \text{ptr}(\rightarrow \text{nxt})^{i} \rightarrow d)$ 标注在图 18 中 ptr 指向的节点上,断言 $\forall i: 1..j-1.(\text{head}(\rightarrow \text{nxt})^{i-1} \rightarrow d <= \text{head}(\rightarrow \text{nxt})^{i} \rightarrow d)$ 标注在图 18 中 head 和 ptr1 指向的两个节点上. 对于有两个指针变元的谓词应用,情况与全称断言的类似.

(2)把能推导出的性质继续标注在 G 的相关节点上,一直到 G 的节点都不会再增加新的性质为止.

① 新性质的相关节点若都不在 G 中,则无须标注这类新性质,这有利于保证推导过程的终止.

② 对于标注有全称断言的浓缩节点,注意它与相邻节点之间是否可以利用 $((\forall i; m..n-1.P(i)) \wedge P(n)) \Leftrightarrow \forall i; m..n.P(i)$ 等等价性来使得全称断言的约束变元的取值范围扩大.

(3)对 Q'(形式为 $Q'_1 \wedge \cdots \wedge Q'_k$)中的每个 $Q'_j(1 \leq j \leq k)$,检查 $Q \Rightarrow Q'_j$ 是否成立.

① 若 Q'_j 是节点相关的断言. 不论是谓词应用、全称断言还是整数断言,若 Q'_j 从相关节点的断言以及 Q 中与节点无关的断言可推导,则 $Q \Rightarrow Q'_j$ 成立,否则不成立. 节点数据之间的关系可能通过与节点无关的断言来传递,因此这里强调包括 Q 中与节点无关的断言. 例如,若 $p \rightarrow d == m$ 且 $p \rightarrow \text{nxt} \rightarrow d == n$ 且 $m < n$,则 $p \rightarrow d < p \rightarrow \text{nxt} \rightarrow d$.

② 若 Q'_j 是节点无关的断言. 若 Q' 不能从 Q 推导,则 $Q \Rightarrow Q'_j$ 不成立.

若对每个 $Q'_j(1 \leq j \leq k)$,$Q \Rightarrow Q'_j$ 都可证,则 $G, T \triangleright Q \Rightarrow Q'$ 得证.

可以证明,若整数域上断言的证明都可终止,则该证明过程终止并且可靠.

这个方法专用于面向操作易变数据结构的程序的验证. 与 Nelson-Oppen 框架不同的是,这里没有交错使用两个理论不断推出新性质并进行传播的过程,而是在形状分析阶段先用形状图理论完成形状图蕴涵关系的证明. 这里只是利用形状图来把符号断言分组,一方面便于推导更多对验证有用的性质,另一方面防止出现无止尽地推导.

另一个不同是,Nelson-Oppen 框架限定被组合的理论为无量词的一阶理论,而这里允许量词断言出现在操作易变数据结构的程序的规范中.

6 实例分析

基于形状图逻辑所实现的程序验证器原型(可从 http://kyhcs.ustcsz.edu.cn/SGL 下载)[14]的流程分成下面 3 个阶段.

(1)预处理阶段. 该阶段为源代码生成抽象语法树并像编译器那样完成通常的静态检查;

(2)形状分析阶段. 生成各程序点的形状图,包括推断循环不变形状图和递归函数的前后形状图;

(3)程序验证阶段. 该阶段通过正向演算生成验证条件,并用 Z3 证明它们.

程序员在编程时,需要提供有关节点数据的函数前后条件和循环不变式. 程序员可以定义一些归纳谓词,用以描述递归数据结构的数据特点,以方便写函数前后条件和循环不变式. 程序员需要提供这些谓词之间的、与程序有关的归纳性质,供自动定理证明器使用. 因为基于演绎推理的证明器推导不出这类性质.

该原型可以验证易变数据结构上较为复杂的程序,如有序循环双向链表、二叉排序树、伸展树、树堆(treap)、二叉平衡树和 AA 树的插入和删除函数. 有关这些例子的统计数据见表 1. 其中的验证时间是在 Windows 7 PC, Intel Core i5-2400 3.1 GHz CPU 和 4 GB 内存的机器上实测获得. 这些程序大

多包括插入和删除函数，以及被它们调用的函数.操作二叉平衡树的程序的函数最多，因为除了插入和删除函数外，还有左旋、右旋、左平衡和右平衡等函数.

表 1　有关形状图和验证条件的统计数据

数据结构与程序功能	函数/个	递归函数/个	验证条件/个	循环/个	验证时间/ms
有序单链表：插入、倒置、合并	3	0	11	4	5942
有序双向链表：倒置	1	0	5	2	11790
有序循环双向链表：插入	1	0	3	1	539
二叉排序树：递归和非递归插入、删除	4	2	15	3	1868
AA 树：插入	3	1	8	0	24274
二叉平衡树：插入、删除	8	3	32	0	96556
树堆：插入、删除	7	3	18	0	5912
伸展树：插入、删除	6	0	16	2	122424

表 1 各例的循环不变形状图都是用本文方法自动推断的.其中有序双向链表的倒置函数虽只有两个循环，但验证时间明显高于其他链表程序.这是因为在用循环代码倒置双向链表过程中，链表被分成正向和逆向两部分，需要多次迭代才能得到循环不变形状图，并且循环不变形状图是 8 种情况的析取.

表 1 各例的递归函数的前后形状图也都是用本文方法自动推断的.二叉排序树插入函数的前后形状图分别在图 19 的(1)和(2)中，其中函数后形状图分成插入前实参指向非空表和空表两种情况.从此例可以看出形状图上出现对应形参指针 p 的实参指针 p'的必要性.其他二叉树的插入函数的后形状图也都这样.

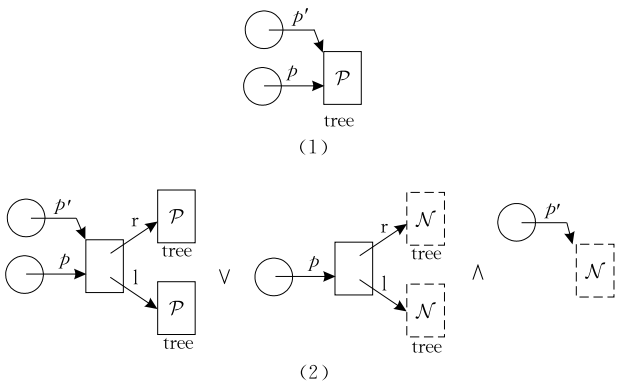

图 19　二叉排序树插入函数的前后形状图

表 1 中二叉排序树的验证时间远低于列在其后的 4 种二叉树，这是因为二叉排序树仅需要验证有序性，而其他几种二叉树还需要验证平衡性或类似平衡性的性质.伸展树的代码没有递归，函数个数和验证条件也不算多，但验证时间远高于其他二叉树.这是因为其 splay 函数的循环不变形状图是 20 多种情况的析取，形状分析阶段和程序验证阶段的耗时都大大增加.

下面再介绍 1 个规模不大但能帮助理解本文的例子.

例 2.　二叉排序树的递归插入函数，代码和断言见图 20，程序断言中的 y 和 z 是整型逻辑变量.本例引入的谓词定义如下：

(1) $\mathrm{sorted}(\mathrm{Node}^*\ p) \triangleq p==\mathrm{NULL} \vee \mathrm{sorted}(p\to \mathrm{l}) \wedge \mathrm{sorted}(p\to \mathrm{r}) \wedge \mathrm{gt}(p\to \mathrm{d}, p\to \mathrm{l}) \wedge \mathrm{lt}(p\to \mathrm{d}, p\to \mathrm{r})$

(2) $\mathrm{gt}(\mathrm{int}\ m,\ \mathrm{Node}^*\ p) \triangleq p==\mathrm{NULL} \vee m>p\to d \wedge \mathrm{gt}(m, p\to \mathrm{l}) \wedge \mathrm{gt}(m, p\to \mathrm{r})$

(3) $\mathrm{lt}(\mathrm{int}\ m, \mathrm{Note}^*\ p) \triangleq p==\mathrm{NULL} \vee m<p\to d \wedge \mathrm{lt}(m, p\to \mathrm{l}) \wedge \mathrm{lt}(m, p\to \mathrm{r})$

其中 Node 的定义见图 20.在函数前后条件中分别有 $y>\mathrm{data} \wedge \mathrm{gt}(y,p)$和 $\mathrm{gt}(y,p)$，它们是说：若 y 大于被插入的数据并大于参数树上的所有数据，则 y 大于结果树上的所有数据.z 的含义与 y 类似.

```
typedef struct node{struct node* l;
                    struct node* r; int d;}Node;
/*@assertion sorted(p)∧y>data∧gt(y,p)∧z<data∧lt(z,p)*/
Node* insert(Node* p,int data){
    if (p==NULL) {
      p=malloc(Node); p→l=NULL;
      p→r=NULL; p→d=data;
    } else if (p→d>data){p→l=insert(p→l,data);
    }else if (p→d<data){p→r=insert(p→r,data);
    }
    return p;
}/*@assertion sorted(p)∧gt(y,p)∧lt(z,p)*/
```

图 20　二叉排序树的插入函数

引入的性质定理有：

(1) $x<y \wedge \mathrm{lt}(y,p) \Rightarrow \mathrm{lt}(x,p)$

(2) $x>y \wedge \mathrm{gt}(y,p) \Rightarrow \mathrm{gt}(x,p)$

在函数出口点产生的验证条件包括下述蕴涵式，它表示图 20 中第 1 个递归调用后的断言蕴涵函数后条件：

$\mathrm{sorted}(p\to \mathrm{r}) \wedge \mathrm{lt}(p\to \mathrm{d}, p\to \mathrm{r}) \wedge y>p\to \mathrm{d} \wedge \mathrm{gt}(y, p\to \mathrm{r}) \wedge z<p\to \mathrm{d} \wedge \mathrm{lt}(z, p\to \mathrm{r}) \wedge$

$y>\text{data}\wedge z<\text{data}\wedge p\rightarrow\text{d}>\text{data}\wedge$
$\text{sorted}(p\rightarrow\text{l})\wedge\text{gt}(y,p\rightarrow\text{l})\wedge\text{lt}(z,p\rightarrow\text{l})\wedge$
$\text{gt}(p\rightarrow\text{d},p\rightarrow\text{l})$
$\Rightarrow\text{sorted}(p)\wedge\text{gt}(y,p)\wedge\text{lt}(z,p)$

其中蕴含式的前件中的前三行断言是从调用点前延续到调用点后,后两行断言是从函数后条件得到.在递归调用结束后,能得到第 5 行 $\text{gt}(p\rightarrow\text{d},p\rightarrow\text{l})$的原因是:在递归调用点前,有 $p\rightarrow\text{d}>\text{data}$ 与 $\text{gt}(p\rightarrow\text{d},p\rightarrow\text{l})$,因此 $p\rightarrow\text{d}$ 可与被调用的递归函数的前、后条件中的逻辑变量 y 匹配.

图 19(2)两种情况分别都属于该蕴涵式的证明环境.按照第 5 节所提出的直接把形状图用于证明 $G,T\triangleright Q\Rightarrow Q'$ 的方法,把该蕴涵式前件的断言分别对图 19(2)两种情况的形状图标注如下:

(1)根节点:$\text{gt}(p\rightarrow\text{d},p\rightarrow\text{l})$,$\text{lt}(p\rightarrow\text{d},p\rightarrow\text{r})$,
$y>p\rightarrow\text{d},z<p\rightarrow\text{d},p\rightarrow\text{d}>\text{data}$

(2)左子节点:$\text{sorted}(p\rightarrow\text{l})$,$\text{gt}(p\rightarrow\text{d},p\rightarrow\text{l})$,
$\text{gt}(y,\text{p}\rightarrow\text{l})$,$\text{lt}(z,p\rightarrow\text{l})$

(3)右子节点:$\text{sorted}(p\rightarrow\text{r})$,$\text{lt}(p\rightarrow\text{d},p\rightarrow\text{r})$,
$\text{gt}(y,p\rightarrow\text{r})$,$\text{lt}(z,p\rightarrow\text{r})$

再把利用谓词定义和性质定理能推导出的断言增加到这些节点上:

(1)根节点:$\text{sorted}(p)$,$\text{gt}(y,p)$,$\text{lt}(z,p)$;

(2)左子节点:无增加;

(3)右子节点:无增加.

该蕴涵式在这两种情况下都得证,因为其后件的断言都已经标注在根节点上了.

Z3 在未能得出待证断言有效或不可满足时,总试图继续推导出更多的性质.一种情况是它不断地把归纳谓词展开,导致证明过程不终止.针对操作易变数据结构的程序的验证,直接把形状图用于验证条件的证明可以有效地阻止这种不终止.从例 2 可以看出,在生成验证条件的演算过程中,符号断言中的谓词断言的展开和形状图上谓词节点的展开本质上是同步的.从例 1 也可以看出,全称断言的展开(指从 $\forall x:m..n.\varphi(x)(m<=n)$ 变换成 $\varphi(m)\wedge\forall x:m+1..n.\varphi(x)(m<=n)$ 等情况)和形状图上浓缩节点的展开也是同步的.由此可知,在证明验证条件时,在把断言按节点分组和推导新性质的过程中,若谓词展开式中与节点有关的断言在形状图上没有可挂靠的节点,则该谓词展开也不会给证明带来什么帮助,因而不必展开.

本文并没有把第 5 节所提出的形状图理论和整数理论组合的证明方法实现到 Z3 中,而是在验证器原型中根据该方法的思想对验证条件进行预处理后再交给 Z3,扩大了可证的范围,并减少了 Z3 不终止的情况.

7 相关工作比较

分离逻辑是验证堆操作程序(操作易变数据结构的指针操作程序是其重要部分)的最流行的手工或半自动推理的程序逻辑[15].除了分离逻辑外,还存在一些其它逻辑,如匹配逻辑[16],也是用于手工或半自动的推理.

分离逻辑引入一种专用的分离符号(*),用它把不能混淆到一起推理并且不能被忽略的断言分离开来.分离逻辑的优点和缺点都源自其引入的分离概念.分离的好处之一是避免了访问路径别名给程序验证带来的困难;其缺点是分离切断了不同堆块之间的信息联系,给需要全局信息才能推断的程序性质的验证带来困难.例如,分离逻辑难以发现对某堆块的一个指针域的赋值引起另一个堆块的泄漏.对于堆操作程序的验证,通常允许用户定义的归纳谓词.在分离逻辑的环境下,必须另行设计归纳谓词的展开或折叠策略,否则因归纳谓词的展开没有控制而导致验证条件的证明可能不终止.因此,分离逻辑的证明辅助工具经常是启发式且不完备的[17].最近也出现了一些小的判定片断[18-21].其中文献[20]提出了一种有效、可靠和完备的自动定理证明器,用于检查带表段谓词的分离逻辑公式之间蕴涵的有效性,但它只能用于操作单链表的程序.文献[21]提出分离逻辑一个较小的可判定片段与其他可判定的一阶理论的组合理论的判定方法.

形状图逻辑的最大特点是,形状图既是机器状态中有关堆部分的抽象表示,又是指针相等性断言和有效性断言等的图形表示,它总揽了分离性和整体性[10],至少带来下面的好处:

(1)形状图逻辑无须使用分离合取算符,避免了引入分离合取算符给定理证明带来的复杂性.在形状图上,关联到不同节点的断言,就是分离逻辑中用分离合取算符隔开的断言.

(2)形状图提供了全局信息.

①用形状图逻辑很容易发现对某堆块的一个指针域的赋值引起另一个堆块的泄漏.

②形状图可用来指导验证条件证明过程中与易变数据结构有关的归纳性质的展开,以避免证明过程的不终止.

分离逻辑优于形状图逻辑的重要一点是它适用于汇编语言级的程序推理[22].

使用分离逻辑，但验证条件的证明策略类似本文方法的是 Chin 等人的论文[23]. 他们设计了一个蕴涵检查过程，该过程使用展开和折叠推理，能够处理可能是归纳定义的良基谓词. 其中的创新点是他们找到了一种在有用户定义归纳谓词情况下，保证过程可靠和终止的充分条件. 该过程把验证条件简化到标准的逻辑理论. 该过程能够处理的易变数据结构超越了树.

用符号堆也是把握程序状态中的指针信息以避免使用分离逻辑，从而可自动推理的办法. Madhusudan 等人[24]使用节点集合和声明变量集合，还有代表有向边集、节点到值的映射和声明变量到值的映射的若干函数来描述程序状态. 该文的内存印迹(footprint)由符号堆和 DRYAD 公式组成，它们分别对应到本文的形状图 G 和数据结构所需满足的除了形状以外的性质 Q. DRYAD 是分离逻辑的一种方言，本质上是堆上无量词一阶逻辑，并用多个类型的归纳定义来增强，这些类型是为其下有树结构的存储单元定义的. 它用一个可靠且终止的过程，来证明用命令式语言编写的树结构上很多递归算法的功能完全正确性. Qiu 等人[25]使用集合论，把 DRYAD 翻译成带递归的经典逻辑，并提出一种自然证明技术来证明验证条件. 这种技术提供在手工证明中常用的证明策略来帮助验证，并把这些策略自动部署到代码中. Pek 等人[26]为利用 VCC 框架[27]，把自然证明策略翻译成嵌在 VCC 标注中的幽灵代码，促成 VCC 获得采用可判定理论的自然证明. 经过这些努力，他们的工具能够验证多种常见的、包括用递归或迭代方式实现的、操作易变数据结构的程序.

本文与 Qiu 和 Edgar 等方法的主要区别有 3 点. 首先，有关易变数据结构的形状的验证，本文是基于形状图蕴涵理论，直接在形状图上而不是在符号断言上验证. 其次，有关归纳谓词的展开策略，他们采用由程序员提供证明策略的方式，而本文所设计的方法是利用已经建立的形状图来指导归纳谓词的展开和折迭. 最后，他们的方法必须由程序员提供完整的循环不变式，而我们的方法自动推断有关形状部分的循环不变式.

研究全自动推理的逻辑的目标是研发快速、可靠和完备的判定过程. 为达到该目标，这些逻辑的表达力通常受到严重的限制. 对于堆操作程序的验证逻辑来说，使用可达性谓词是一种避开分离逻辑的常用简便方式，但表达能力上受到较大限制. 具体例子有在一阶逻辑上扩展可达性的若干逻辑、LISBQ[28]、CSL[29]及组合树逻辑和整数逻辑的 $\text{STRAND}_{\text{dec}}$[30]. LISBQ 逻辑提供一种带受限的可达性谓词和量化的推理. 这种逻辑有完全自动、可靠、完备和终止的判定过程. 虽然这是一个高效的判定过程，但是在描述归纳数据结构性质时表达力上非常有限. $\text{STRAND}_{\text{dec}}$ 也是一种这样的逻辑，它能够应对某些数据结构的性质，并且通过组合可判定的树理论和算术理论，容许可判定的片断. 然而它在表达力上极其有限. 这些逻辑都不足以描述诸如平衡二叉树的平衡性和堆中的一组键值在通过一段程序后没有改变等复杂性质.

在有关指针性质的循环不变式推断上，Magill 等人[31]提出了一种用分离逻辑推断单链表操作程序的循环不变式的方法. 他们使用符号执行机制，并且给出一组重写规则来交互地计算不动点. 所实现的原型仅完成了计算表长、求各节点数据的和、以及链表的删除、倒置、合并和拆分这几种简单的单链表操作函数的循环不变式的推断. Distefano 等人[32]提出了一种类似的基于分离逻辑的形状分析方法. 和文献[31]一样，它使用了专为单链表的展开和折叠而设计的单链表谓词，也需要关于易变数据结构形状的先验知识，并且也使用符号执行机制. 但是它的单链表谓词和文献[31]的不同，与文献[28]相比，它还能计算循环单链表操作函数的循环不变式，例如循环单链表的删除函数和过滤器函数. Guo 等人[33]提出了一种基于分离逻辑的过程间形状分析，它执行归纳的递归综合，以自动推断有类似树骨架的任意递归形状不变式. 其主要特点是从循环体的固定迭代次数的符号执行中抽取循环不变式，但是只能用于树骨架的形状. 对于单链表、循环单链表和二叉树，Qin 等人[34]不仅能推断有关形状的循环不变式，而且能推断有关相邻节点数值数据的循环不变式，推进了他们先前的工作[23]，也使得他们的结果优于其他人的结果. 他们也采用符号执行机制，对循环体进行迭代计算，其特点是使用专门设计的最小上界算子和加宽算子来保证收敛到不动点.

本文和文献[34]相比，两者循环不变式的推断算法都是基于文献[11]. 我们的特点是采用本文的形状图蕴涵理论，在形状图上直接推断循环不变形状图，文献[34]的特点是还能推断有关相邻节点数值数据的循环不变式. 在保证收敛到不动点上，我们

采用形状图上的蕴涵规则[10],并且用该文的形状系统来保证算法的终止.就推断有关形状的循环不变式而言,文献[34]能推断操作单链表、循环单链表和二叉查找树的代码的循环不变式,而我们还能推断操作双向链表、循环双向链表和伸展树的代码的循环不变形状图.

从第6节及表1可知,本文推断循环不变形状图的能力远胜于上面几项工作.这是由于本文的推断得到形状系统的支持.也正是由于形状系统的支持,本文还能推断递归函数的前后形状图.据我们所知,尚未有这方面论文的发表.

图逻辑或形状逻辑一般是指推导图(或有向图、形状图)性质的空间逻辑[35-36].其中图是用符号公式来描述而符号公式的语法由简单文法来定义,要推导的图上的性质也用符号公式来表示,推导规则也还是基于符号公式的.超边替换文法(hyperedge replacement grammar)[37]是一个实例.它用超图为堆状态建模,把双向的图规则用于超图的变换.但是该文法只是作为动态数据结构建模的一种直观的形式方法,超图和规则等还是用符号公式来表示的.本文所用的形状图逻辑与它们有根本的区别:形状图直接作为逻辑公式,出现在程序逻辑的推理规则中,并且还可以和符号公式一起演算.

从规范语言的综述[38]和工业界的使用[8]看,迄今为止尚无像本文这样采用形状图或采用其他非符号方式来表达程序的部分性质,并与表达程序其余性质的符号断言一起作为程序的规范并一起演算的.

8 总 结

随着面临越来越复杂系统的定理证明,逻辑系统也面临逻辑公式复杂、推理规则复杂和自动定理证明困难等问题.为降低难度,从本文的经验看,把其中适合于用图形表示的领域专用逻辑图形化,形成符号逻辑和图形逻辑的组合,是一条可以尝试的途径.

下一步将考虑怎样在目前基础上设计实用性大大提高的形状图逻辑和形状系统.首先,充实基本形状集,并分成单态基本形状和多态基本形状.单态基本形状的特点是,易变数据结构中各节点的类型相同,各节点所含指针数相同并且它们指向的类型都是节点本身的类型.本文谈及的单链表等5种形状都属于单态基本形状.多态基本形状的特点是,易变数据结构中各节点的类型相同,各节点所含指针数可以不同但它们指向的类型都是节点本身的类型.节点的多态性依靠节点类型中的共用体域来体现.例如编译器中常用的抽象语法树就可以设计成多态基本形状.

其次,规定复杂形状的构造方式,并相应地修改形状系统中的形状推断规则和形状检查规则.复杂形状至少有嵌套形状、含内部附加指针的形状和含外来附加指针的形状这样三类,它们在构造方式上的特点分别是:各节点都有指向同种内嵌形状的指针(例如双向链表的节点都有指针指向各自的单链表)、各节点都有指向本形状节点的附加指针(例如带父节点指针的二叉树)、来自形状外部的附加指针(例如队列可以看成带一个这种附加指针的单链表,附加指针指向链表的最后一个节点).

用基本形状集加上复杂形状的几种构造方式,可以控制住易变数据结构的复杂性,并满足大部分应用对易变数据结构的需求,同时也使得扩展形状图逻辑来自动验证操作复杂易变数据结构的程序成为可能.

致 谢 感谢刘刚、张志天、宋艳辉、孟建超、韩亚慧和郝燧等研究生,他们为实现系统原型做出了贡献!

参 考 文 献

[1] Moy Y, Ledinot E, Delseny H, et al. Testing or formal verification: DO-178C alternatives and industrial experience. IEEE Software, 2013, 30(3): 50-57

[2] Nanevski A, Morrisett G, Shinnar A, et al. Ynot: Dependent types for imperative programs//Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. Victoria, Canada, 2008: 229-240

[3] Barnett M, Leino K R M, Schulte W. The Spec# programming system: An overview//Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. Marseille, France, 2004: 49-69

[4] Flanagan C, Leino K R M, Lillibridge M, et al. Extended static checking for Java//Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation. Berlin, Germany, 2002: 234-245

[5] Berdine J, Calcagno C, O'Hearn P W. Smallfoot: Modular automatic assertion checking with separation logic//Proceedings of the 4th International Conference on Formal Methods for Components and Objects. Amsterdam, The Netherlands, 2006: 115-137

[6] Distefano D, Parkinson M J. jStar: Towards practical verification for Java//Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications. Nashville, USA, 2008: 213-226

[7] Paulson L C. Isabelle: A generic theorem prover. Volume 828 of Lecture Notes in Computer Science, Berlin: Springer-Verlag, 1994

[8] De Moura L, Bjørner N. Z3: An efficient SMT solver//Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Budapest, Hungary, 2008: 337-340

[9] Nelson G, Oppen D C. Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems, 1979, 1: 245-257

[10] Li Z P, Zhang Y, Chen Y Y. A shape graph logic and a shape system. Journal of Computer Science and Technology, 2013, 28(11): 1063-1084

[11] 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 Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1977: 238-252

[12] Mitchell J C. Foundations for Programming Languages. Cambridge, MA: The MIT Press, 1996

[13] Knuth D, Bendix P. Simple word problems in universal algebras//Leech J ed. Computational Problems in Abstract Algebra. Oxford: Pergamon Press, 1970: 263-297

[14] Zhang Zhi-Tian, Li Zhao-Peng, Chen Yi-Yun, Liu Gang. An automatic program verifier for PointerC: Design and implementation. Journal of Computer Research and Development, 2013, 50(5): 1044-1054(in Chinese)
(张志天，李兆鹏，陈意云，刘刚. 一个程序验证器的设计和实现. 计算机研究与发展，2013，50(5)：1044-1054)

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

[16] Rosu G, Ellison C, Schulte W. Matching logic: An alternative to Hoare/Floyd logic//Proceedings of the 13th International Conference on Algebraic Methodology and Software Technology. Lac-Beauport, Canada, 2010: 142-162

[17] Berdine J, Calcagno C, O'Hearn P W. Symbolic execution with separation logic//Proceedings of the 3rd Asian Conference on Programming Languages and Systems. Tsukuba, Japan, 2005: 52-68

[18] Jia L M, Walker D. A foundation for automated reasoning about pointer programs//Proceedings of the 15th European Conference on Programming Languages and Systems. Vienna, Austria, 2006: 131-145

[19] Magill S, Tsai M-H, Lee P, Tsay Y-K. THOR: A tool for reasoning about shape and arithmetic//Proceedings of the 20th International Conference on Computer Aided Verification. Princeton, USA, 2008: 428-432

[20] Navarro Pérez J A, Rybalchenko A. Separation logic+superposition calculus=heap theorem prover//Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. San Jose, USA, 2011: 556-566

[21] Piskac R, Wies T, Zufferey D. Automating separation logic using SMT//Proceedings of the 25th International Conference on Computer Aided Verification. Saint Petersburg, Russia, 2013: 773-789

[22] Jensen J B, Benton N, Kennedy A. High-level separation logic for low-level code//Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Rome, Italy, 2013: 301-314

[23] Chin W-N, David C, Nguyen H H, Qin S. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Science of Computer Programming. Amsterdam: Elsevier North-Holland, Inc., 2010, 77(9): 1006-1036

[24] Madhusudan P, Qiu X, Stefanescu A. Recursive proofs for inductive tree data-structures//Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Philadelphia, USA, 2012: 123-135

[25] Qiu X, Garg P, Stefanescu A, Madhusudan P. Natural proofs for structure, data, and separation//Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. Seattle, USA, 2013: 231-242

[26] 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. Edinburgh, UK, 2014: 440-451

[27] Cohen E, Dahlweid M, Hillebrand M, et al. VCC: A practical system for verifying concurrent C//Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics. Munich, Germany, 2009: 23-42

[28] Lahiri S, Qadeer S. Back to the future: Revisiting precise program verification using SMT solvers//Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. San Francisco, USA, 2008: 171-182

[29] Bouajjani A, Dragoi C, Enea C, Sighireanu M. A logic-based framework for reasoning about composite data structures//Proceedings of the 20th International Conference on Concurrency Theory. Bologna, Italy, 2009: 178-195

[30] Madhusudan P, Parlato G, Qiu X. Decidable logics combining heap structures and data//Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Austin, USA, 2011: 611-612

[31] Magill S, Nanevski A, Clarke E, Lee P. Inferring invariants in separation logic for imperative list-processing programs//Proceedings of the 3rd Workshop on Semantics, Program

Analysis, and Computing Environments for Memory Management. Charleston, USA, 2006

[32] Distefano D, O'Hearn P W, Yang H. A local shape analysis based on separation logic//Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Vienna, Austria, 2006: 287-302

[33] Guo B, Vachharajani N, Aigist D. Shape analysis with inductive recursion synthesis//Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation. San Diego, USA, 2007: 256-265

[34] Qin S C, He G H, Luo C G, et al. Loop invariant synthesis in a combined abstract domain. Journal of Symbolic Computation, 2013, 50: 386-408

[35] Cardelli L, Gardner P, Ghelli G. A spatial logic for querying graphs//Proceedings of the 29th International Colloquium on Automata, Languages and Programming. Malaga, Spain, 2002: 597-610

[36] Rensink A. Canonical graph shapes//Proceedings of the 13th European Symposium on Programming. Barcelona, Spain, 2004: 401-415

[37] Jansen C, Noll T. Generating abstract graph-based procedure summaries for pointer programs//Proceedings of the 7th International Conference on Graph Transformation. York, UK, 2014: 49-64

[38] Hatcliff J, Leavens G T, Leino K R M, et al. Behavioral interface specification languages. ACM Computing Surveys, 2012, 44(3): 16

ZHANG Yu, born in 1972, Ph. D., associate professor. Her research interests include theory and implementation technology of programming languages, program verification, as well as deterministic parallel programming models and runtime systems.

CHEN Yi-Yun, born in 1946, professor, Ph. D. supervisor. His research interests include theory and implementation technology of programming languages, program verification, and software safety.

LI Zhao-Peng, born in 1978, Ph. D., associate professor. His research interests include programming languages, program verification, certifying compilers and automated theorem proving.

Background

Logical inference is one approach of formal verification to improve the dependability of software. It uses mathematical methods to reason about software. Most logical inference based researches such as Ynot design logic systems to reason about programs and generate verification conditions (VCs), then use certain theorem prover, e. g. Z3, to prove these VCs. Although these tools have been developed in labs, there are difficulties in automated theorem proving to support verifying real-world software, including alias analysis, loop invariant inference, expressivity of assertion languages, design of domain-specific logics, etc.

To alleviate the burdens of automated theorem proving, we consider lowering requirements on the capability of theorem provers. We design mechanisms for programming languages to raise the threshold of legal programs and reject some programs containing errors logically. We also use program analysis to collect information for verification.

Our current research concentrates on verifying pointer programs manipulating mutable data structures in two stages: shape analysis and program verification. First, we propose shape graphs as assertions of pointer equality and validity, and design a shape graph logic as an extension of Hoare logic. The proposed logic includes inference rules for statements manipulating pointers, which can be used in analyzing and verifying pointer programs. Second, we propose a shape system for PointerC language. It contains shapes and their definitions, rules for shape inference and shape checking. Third, we propose a method doing verification using traditional Hoare logic directly by eliminating aliasing with the aid of shape graphs. Finally, we implement a prototype for automatically verifying pointer programs manipulating data structures such as sorted circular doubly-linked list, binary search trees, splay trees, treaps, AVL trees and AA trees.

This paper is a complement to our paper *A Shape Graph Logic and a Shape System* published in *Journal of Computer Science and Technology*, 28(6), 2013. We investigate the theory of shape graphs and its decision method using analogous methods to those in studying the theory of algebraic specification. We then propose methods for inferring loop invariant shape graphs, and the pre-/post-shape graphs of recursive functions. We also propose a decision method for the combination of the theory of shape graphs and integer theory with respect to characteristics of VCs.

This work is supported by the National High Technology Research and Development Program (863 Program) of China under Grant No. 2012AA010901, and the Natural National Science Foundation of China under Grant Nos. 61170018, 61229201.