操作带附加链表的数据结构的程序的验证*

[1](中国科学技术大学 计算机科学与技术学院，安徽 合肥 230026)
[2](中国科学技术大学 苏州研究院软件安全实验室，江苏 苏州 215123)

Verification of Programs Manipulating Data Structures with an Additional Linked List

(School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026)
(Software Security Lab., Suzhou Institute for Advanced Study, University of Science and Technology of China, Suzhou 215123)
+ Corresponding author: E-mail:

Abstract: A mutable data structure in some programs has an extra pointer field through which part of nodes in the structure can be linked into an additional linked list. The equality between extra pointers and other pointers to the data structures is always uncertain, which brings significant difficulty to reason about such programs. In this paper, we extend our original shape graph logic and shape system to propose a verification method for such programs. The method requires programmers to notate extra pointers for additional linked lists while defining recursive structures, and stipulates programming constraints for extra pointers to easily check the correctness of shapes. During the verification process, the shape analysis separates the additional linked list from the shape graph of a main data structure and establishes a separated virtual additional linked list, thus the equality between extra pointers and other pointers can be ignored. Using the extended verification prototype, we have verified the code manipulating a complex request queue in the program of asynchronous I/O operations in GNU C Library (file aio_misc.c).

Key words: Shape Graph Logic; Pointer Logic; Shape Analysis; Program Analysis; Program Verification

摘 要：某些易变数据结构含有通过节点中的附加指针域将部分节点链接成附加链表。附加指针与形成主数据结构的指针之间的相等关系通常静态不可确定，这给推理验证操作这类数据结构的程序带来困难。本文扩展我们已有的形状图逻辑和形状系统，提出一种验证这类程序的方法。该方法要求程序员在定义这类结构体类型时对其中形成主结构或附加链表的指针等分别进行标注，并围绕这些指针定义便于检查形状正确性的程序约束；在程序验证时，形状分析采取将附加链表从主数据结构的形状图上分离出来并建立独立的虚拟附加链表，从而无需考虑主数据结构指针和附加指针之间的相等关系即可进行验证。我们使用扩展后的验证系统原型，验证了 GNU C Library 中实现异步 I/O 操作的程序（aio_misc.c）中处理复杂请求队列的代码。

关键词： 形状图逻辑、指针逻辑、形状分析、程序分析、程序验证
中图法分类号: TP301 **文献标识码**: A

* Supported by the National Natural Science foundation of China under Grant No. 61170018, 61229201（国家自然科学基金）and China Postdoctoral Science Foundation (NO.2012M521250，中国博士后基金)

1 引言

形式验证是提高软件可信程度的重要方法，它包括抽象解释、模型检测和演绎验证等途径，它们在工业界已经逐步得到应用，尤其是前两者。例如，形式方法已被增补到民用航空机载软件研制标准 DO-178B/C 的补充文件 DO-333 中，在空客 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对部分安全攸关嵌入式软件的测试[1]。

演绎验证（也称为程序证明）是极有前景但仍需加强研究和逐步推广的方法。演绎验证是指用逻辑推理的方法证明程序的性质，常用的方式是：程序员用某种规范（*specification*）语言写出所期望的程序性质，系统采用对应某种程序逻辑的一种演算（例如对应Hoare逻辑的逆向演算）来产生验证条件，然后用某个（或几个）定理证明器来证明验证条件。若所有验证条件得证，则程序具有规范所描述的性质。

针对操作易变数据结构（本文简称数据结构）的指针程序，其演绎验证必须解决指针别名等导致的 Hoare 逻辑的赋值公理不再适用的问题。为此，我们为指针类型设计过两种专用逻辑：指针逻辑[2-4]和形状图逻辑[5]。在指针逻辑的断言语言中，使用访问路径集合来表示指针型访问路径（例如 q、p->next->next，强调其语法特征时称访问路径，强调其语义时称指针）之间的相等断言，同一个集合中的指针都相等。在形状图逻辑的断言语言中，使用专门设计的形状图来图形化表示指针型访问路径的有效性和相等性，指向形状图中同一个节点的指针都相等。基于这两种断言语言，针对各种引起指针值发生变化的语句，我们设计了两种 Hoare 逻辑风格的推理规则，分别表达指针集合和形状图怎样变化。利用这两种逻辑都能推理指针相等信息，并据此先消除指针别名，再使用赋值公理；但形状图逻辑比指针逻辑直观且更易于实现。

基于形状图逻辑，我们还设计了一种形状系统[5]，它可以理解为面向形状（即数据结构节点之间的链接特点，如单链表）的类型系统，用来提高合法程序的门槛，排除部分没有构造出或操作在程序员所声明形状上的程序，减轻自动定理证明器的负担。利用形状系统对合法形状的约束，我们设计了循环不变形状图和递归函数前后形状图的自动推断算法[6]。我们在断言语言中还允许程序员使用自定义谓词，以提高断言语言的表达能力；并采用由程序员提供自定义谓词之间的性质定理来给自动定理证明器以提示[7]。

经过上述多方面的努力，我们设计和实现的 PointerC 语言的程序验证系统原型[8]已经能够自动验证操作 AVL 树、splay 树、树堆（*treap*）和 AA 树等较为复杂的数据结构的程序。该系统采取两阶段方式完成指针程序的验证：先用形状分析构建各程序点的形状图，然后借助形状图验证程序的其他性质。

该系统的一个重要缺点是，它只能用于验证操作单向链表、循环单向链表、双向链表、循环双向链表和二叉链表示的二叉树这 5 种数据结构形状（称为基本数据结构）的程序；尚不适用于基本数据结构的节点含有额外的指向同类节点的附加指针和/或指向其他类结构节点的次结构指针的情况。例如，图 1 是 GNU C Library 中实现异步 I/O 操作所使用的数据结构的示意图。从 s 开始，由节点的 next_fd 和 last_fd 指针链接的双向链表（基本结构或主结构）是请求队列；从 t 开始，由节点的 next_run 指针（附加链表指针）链接的单向链表（附加链表）是就绪队列；双向链表各节点的 next_prio 指针指向内嵌的描述待处理 I/O 操作的单向链表（次结构）。形状嵌套引出的主次结构因其各节点的主次域指针之间的相等关系可准确定义，而比较容易解决对操作它们的程序的验证；但是附加链表带来的程序验证就困难得多，主要根源在于附加链表指针（如图 1 中就绪队列的指针）指向主结构（如图 1 中的请求队列）

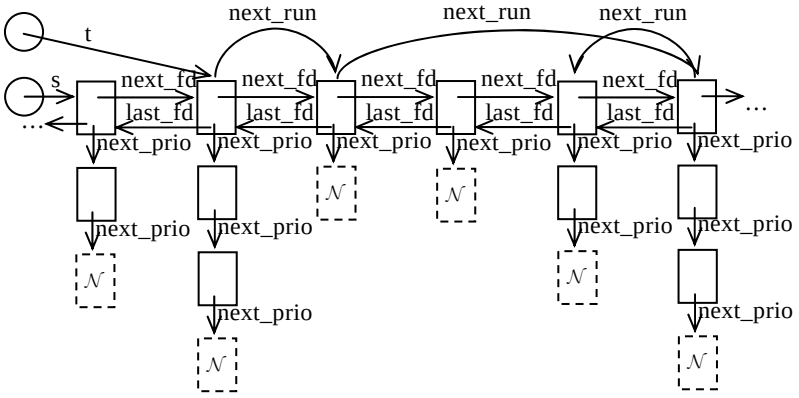

Fig. 1 Schematic diagram of data structure used in implementing async. I/0

图 1 实现异步 I/O 操作所使用的数据结构的示意图

的哪个节点不能静态确定。这导致系统在形状分析时难以判断附加指针和主结构上的哪些指针相等，以及附加指针是否正确地构成附加链表。虽然通过为相等关系不确定的访问路径增加某种机制，能把扩展后的逻辑应用于有向图等指针关系不确定的数据结构[9]，但是扩展后的断言演算规则和程序逻辑的推理规则都大大复杂，以至难以用到程序的自动验证工具中。

本文介绍我们在形状图逻辑的基础上，为操作带附加链表的数据结构的程序所设计的验证方法。该方法的出发点是基于程序员通常只关注节点的附加指针是否把主结构的部分节点正确地链成附加链表，而不关心附加链表上各节点都在主结构上的什么位置。本文的贡献包括如下 3 个方面：

1、为操作带附加链表的数据结构的程序设计一种程序约束。它围绕着维持带附加链表的数据结构的一种不变性：主结构的节点在附加链表上，当且仅当节点的标志域为 true，并且附加链表仅含这些节点。

2、形状分析方法和程序验证方法的扩展设计。其要点是，将附加链表从它所依附的主结构的形状图中分离出来，建立分离的虚拟附加链表。虚拟附加链表的节点是主结构上标志域为 true 的那些节点的拷贝。有了虚拟链表，主结构节点上附加链表指针在主结构形状图上不再体现，避免了因不能确定它们指向哪个主结构节点而难以构建形状图；同时，通过检查虚拟链表即可推理附加链表的构成是否正常。

3、对扩展后的形状分析方法进行正确性证明。该证明基于的想法是：在形状分析过程中，在任何程序点，若除了附加链表指针和主结构指针之间的相等关系没有表达在形状图上外，其余的指针相等关系都准确表达在形状图上，那么该分析方法就是正确的。

本文第 2 节简单介绍形状图逻辑，第 3 节介绍程序约束的设计，第 4 节介绍形状分析方法的扩展设计及其正确性证明，第 5 节是程序验证方法的扩展设计，第 6 节是证明实例，第 7 节和相关研究工作进行比较。第 8 节是总结。

2 形状图逻辑简介

Hoare 逻辑用于程序推理时的一个重要限制是，程序中不同的名字（包括访问路径）必须代表不同的程序对象，即不允许出现别名。例如，对于

赋值语句 p->data = 5 和 后断言 p->data + q->data == 10

用 Hoare 逻辑的赋值公理，可得到该语句前断言 q->data == 5；但该语句的最弱前断言是 p == q ∨ q->data == 5。这里用赋值公理未能得到最弱前断言是因为该公理认为 p->data 和 q->data 代表不同的程序对象，而实际上若 p 等于 q 时，p->data 和 q->data 互为别名，它们代表同一个程序对象。

在将 Hoare 逻辑用于实际程序验证时，必须解决下标变量别名和指针别名等会导致赋值公理不再适用的问题。通常的做法是为相应数据类型设计专用逻辑，如数组逻辑和指针逻辑，用以扩展 Hoare 逻辑。

形状图逻辑[5]是我们针对指针类型设计的专用逻辑。形状图逻辑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表达指针操作语句引起的形状图上被关注部分的变化。

形状图是描述程序中静态声明指针和动态分配的数据结构（这里忽略与本文关系不大的其他类型的数据）中域指针的指向关系的一种有向图，是机器状态的图形表示[5]，它准确表达了指针的有效性和指针之间的相等关系。形状图的节点有六种，在有栈和堆的机器上，节点所代表的程序变量及语义指称如下：

（1）声明节点代表程序中的声明指针，其出边的标记就是该声明指针的名字。声明节点指称栈上的存储单元。

（2）结构节点代表动态创建的结构体变量，其出边的条数及标记与该结构体变量的域指针的个数和名字一致。结构节点指称一个堆块，其存储单元的个数与该节点的出边条数一样。

（3）null 节点和悬空节点不代表任何程序元素，不指称机器上任何东西。

（4）浓缩节点和谓词节点代表若干个动态生成的相互有联系的结构体变量。

有向边不代表任何程序元素，也不指称机器上任何存储单元。边的指向表明由其标记所代表的声明指针（栈单元）或域指针（堆块单元）的值。

（1）若边指向结构节点，则相应指针的值是该结构节点所指称的堆块的地址。

（2）若边指向 null 节点（悬空节点），则相应指针的值等于 NULL（是悬空指针）。

例如，图 2 是表示单向链表的两个形状图，其中圆形节点表示指针型声明变量（称为声明节点），实线矩形节点表示动态分配的结构体变量（称为结构节点，在此即表元），灰色矩形节点表示若干个动态分配的结构体变量的浓缩表示（称为浓缩节点）。浓缩节点下的表达式代表被浓缩的节点的个数，断言是对表达式中变量的约束。中间含字母$\mathcal{N}$和$\mathcal{D}$的虚线矩形节点分别称为 null 节点和悬空节点，表示指向它们的指针分别是 NULL 指针和悬空指针。图 2(a)是下面程序片段（假定 head 指向的单向链表至少有一个表元，并且表长为 n）

```
ptr1 = head; ptr = head->next; m = 0;
while (ptr != NULL) {
    ptr1 = ptr; ptr = ptr->next; m = m +1;
}
```

的循环不变形状图，图 2(b)是循环体中第 1 个语句之前程序点的形状图。在 ptr != NULL 成立时，根据形状图的等价变换规则，将图 2(a)右边的浓缩节点展开一个结构节点，得到图 2(b)。在图 2(b)上，根据形状图逻辑的规则，语句 ptr1 = ptr 使得 ptr1 调整到与 ptr 指向同一个节点，语句 ptr = ptr->next 使得 ptr 指向右边的浓缩节点。再经过语句 m = m +1，循环体结束，根据形状图等价变换规则对形状图进行整理，把原先 ptr1 指向的结构节点折叠进入左边的浓缩节点，就又得到图 2(a)。

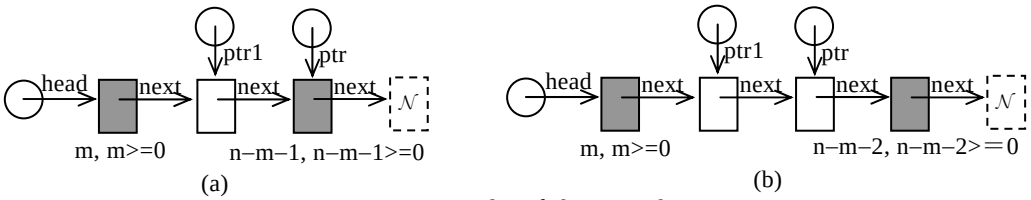

Fig. 2 Two examples of shape graphs

图 2　形状图的两个例子

形状图逻辑可用于对指针程序进行精确的指针分析和程序验证。其最大特点是，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。例如，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。

但是，原有的形状图逻辑[18]是基于数据结构上各节点的域指针之间的相等关系可准确定义，还基于程序每步指针操作对指针指向的修改是静态可判断的。这种限制导致形状图逻辑不能用于指针相等关系部分确定或完全不确定的数据结构。引言提到的带附加链表的数据结构中，附加链表上的指针的源节点和目标节点在主结构上的位置就是不确定的。用形状图难以描述附加链表指针与其他指针之间的关系。

本文后续三节将介绍我们针对验证操作带附加链表的数据结构的程序，对 PointerC 语言的程序验证系统原型所进行的扩展，包括对 PointerC 语言和形状图逻辑的扩展，以及基于它们的验证方法及其正确性证明。

3　PointerC 语言的扩展和程序约束的设计

为便于程序验证系统识别程序中带附加链表、甚至包含次结构的数据结构和它们的形状，我们对 PointerC 语言进行了扩展。扩展后的 PointerC 语言要求，在用于构造数据结构的结构体类型的定义中，对域指针要进行形状标注，包括指向什么形状，构成的是主结构、次结构还是附加链表。若结构体类型中有附加链表指针，则它还必须有一个 bool 类型的标志域。带附加链表的数据结构始终维持一种不变性：主结构的节点在附加链表上，当且仅当节点的标志域为 true，并且附加链表仅含这些节点。例如，针对图 1 所示的数据结构，主结构和次结构在原始的 GNU C Library 的 aio_misc.h 中用同一个数据结构类型定义。根据本文对数据结构的层次观点，我们将图 1 例子使用的数据结构修改为图 3 所示的定义，主次结构类型分别定义，且主结构的 next_prio 换名为 request_ptr 以保存次结构的头指针，并且在主结构类型中通过注释@TAG 标记标志域为 running；主结构指针、附加链表指针和次结构指针分别通过注释 PRIMARY、ADDITIONAL 和 SECONDARY 来标记。

```
typedef struct requestqueue{/* 有关 I/O 的域都被忽略 */
   int abs_prio; /* 按此域递减排序 */
   struct requestqueue* next_prio; /*@ LIST */
}Requestqueue;   // 次结构类型

typedef struct requestlist{
   int aio_fildes; /*主链表按此域递增排序，无相同值*/
   int abs_prio; /*总等于次链表上第一个表元的 abs_prio，
                  附加链表按此域递减排序*/
   bool running; /*@ TAG */
   struct requestlist* last_fd;     /*@ DLIST, PRIMARY */
   struct requestlist* next_fd;    /*@ DLIST, PRIMARY */
   struct requestlist* next_run;   /*@ ADDITIONAL */
   Requestqueue*request_ptr;   /*@ LIST, SECONDARY*/
}Requestlist;   // 主结构类型
```

Fig.3 Data structure definition of the example in Fig. 1

图 3 图 1 实例对应的 PointerC 数据类型定义

为便于保证带附加链表的数据结构的不变性和形状正确，并且便于其他性质的验证，我们设计如下的编程原则：

编程原则 1 区分主结构指针和附加链表指针，它们分别用于操作主结构和附加链表。

从类型系统角度看，这两类指针是指向同一种结构体类型的指针。为此，对于声明指针，附加链表的声明指针通过在变量声明时加标注来与主结构的声明指针相区分。以主结构声明指针开始且只用主结构指针域构成的访问路径仍然是**主结构指针**；类似地，以附加链表声明指针开始且只用附加链表指针域构成的访问路径仍然是**附加链表指针**。

编程原则 2 新创建的主结构节点只允许由主结构指针来指向。主结构的节点只有在其不同时为附加链表的节点时，才可通过主结构指针释放。

编程原则 3 主结构的节点用其标志域来指示其是否同时为附加链表的节点。

编程原则 4 一个函数仅通过主结构指针或仅通过附加指针访问节点数据。

编程原则 4 的引入主要是为了避免因缺少主结构指针和附加指针之间的相等关系而无法判断以这两类指针为访问路径前缀的数据域访问路径之间的别名关系，从而便于验证除形状以外的其他性质（见第 5 节）。

基于上述编程原则，我们设计如下 8 点程序约束，以便于静态检查程序是否遵循这些编程原则：

（1）主结构指针和附加链表指针应分别声明。以主结构声明指针开始的访问路径（下面用 p 表示）中不允许出现附加链表指针域；以附加链表声明指针开始的访问路径（下面用 a 表示）中不允许出现主结构指针域。这一点用于保证编程原则 1。

（2）语句 p = malloc(…)之后必须是语句 p->tag = false（若标志域名是 tag）。任何情况下都不允许用 a = malloc(…)语句。

（3）在 p->tag 等于 false 时，可以使用语句 free(p)。任何情况下都不能使用 free(a)。

上述两点保证了编程原则 2。

（4）在 p->tag 等于 false 时，可通过 p->tag = true; a = p 两个连续语句使得 p 指向的节点对附加链表开放，即程序可以用附加链表指针访问该节点。语句 a = p 只有在前一个语句是 p->tag = true 时才能使用。

（5）在 a->tag 等于 true 时，可用语句 a->tag = false 来禁止 a 所指向节点继续被附加链表指针访问。

（6）除了（2）、（4）和（5）外，任何其他情况下不能修改节点标志域的值。

上述三点保证了编程原则 3。

（7）任何情况下不允许用赋值语句 p = a。

以上 7 点一起可保证编程原则 1。

（8）在一个函数体中，仅允许以主结构指针或者仅允许以附加链表指针为前缀对节点上的数据域（非指针域，包括标志域）进行访问。这一点保证编程原则 4。

程序约束（1）、（2）、（7）和（8）是简单的语法问题，很容易检查。对于其他有前提条件（例如“在 a->tag 等于 true 时”）的程序约束，通过简单的程序分析就能完成程序是否满足这些约束的检查。

4 带附加链表时的形状分析方法及其正确性证明

由于程序员并不关心附加链表各节点在主结构上的位置，即不关心附加链表指针和主结构指针之间的相

等关系，故对带附加链表时的形状分析的解决思路是：将附加链表从主结构的形状图中分离出来，建立分离的虚拟附加链表（简称虚拟链表）。虚拟链表的节点是主结构上标志域为 true 的那些节点的副本。有了虚拟链表，原先指向主结构节点的附加链表指针在主结构的形状图上不再出现，避免了因不知它们的指向而难以构建形状图，同时也便于检查附加链表的构成是否正常。主结构节点是否有次链表指针不会影响采用虚拟链表的方案。图 4 是该方法的示意图。图 4(a)是带附加链表的数据结构的示意图，图 4(b)是程序分析时相应的示意图，其中上下两个图分别是虚拟链表和主结构的示意图。各程序点的形状图可能比这复杂，因为主结构和附加链表都会有经过多个语句完成的插入或删除等操作。

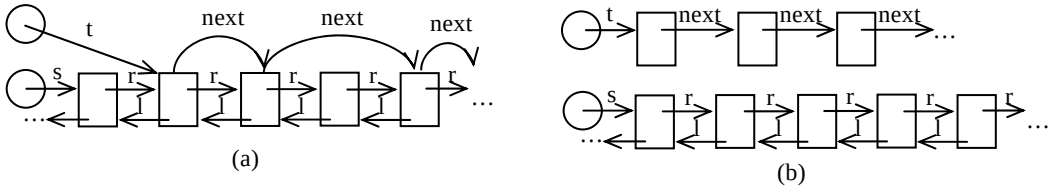

Fig. 4 Schematic diagram of shape analysis in the case with additional linked list

图 4 带附加链表时形状分析的示意图

系统在形状分析时，通过对第 3 节程序约束（4）和（5）中提到的语句进行特别处理，就能把程序对附加链表的操作体现在虚拟链表的形状图上，从而正确构造没有附加链表指针的主结构形状图和虚拟链表形状图。形状分析中的特别处理有如下几点：

（1）对程序中符合程序约束（4）的代码 p->tag = true; a = p，构造虚拟链表的一个节点，并让 a 指向这个节点。即把程序段 p->tag = true; a = p 当作 p->tag = true; a = malloc(Node)来处理。

在有次结构的情况下，还需让新构造的虚拟链表节点的次结构指针指向新建的次结构谓词节点[18]。

（2）对程序中符合程序约束（5）的代码 a->tag = false，在虚拟链表形状图上把 a 所指向的节点释放。即把 a->tag = false 当作 a->tag = false; free(a)来处理。

在有次结构的情况下，需先释放指向的次结构谓词节点，然后再释放 a 所指向的主结构节点副本。

（3）程序中对附加链表的其他操作（例如指针赋值语句的两边都是附加链表指针），自然都变成在虚拟链表上进行相应的操作。

（4）若在虚拟链表的形状图上出现内存泄漏，它并不是真正的内存泄漏。它表明节点的标志域仍为 true 但已不能被任何 a 访问了。这时需要报告程序错误：节点已不在附加链表上，但节点标志却仍然是 true，即会导致违反数据结构不变式。

形状分析的其他部分仍然和[5]中描述的一样，没有变化。

下面讨论该方法的正确性的证明。非形式地说，在采用该方法的形状分析过程中，在任何程序点，若除了附加链表指针和主结构指针之间的相等关系没有表达在形状图上外，指针之间的其余相等关系都准确表达在形状图上，那么该分析方法是正确的。下面的证明未考虑有次结构的情况，但可推广到有次结构的情况。

对一个程序 P，形状分析实际上分析的是略有区别的程序 P′，P′与 P 的区别在于上述特别处理（1）和（2）。P 中的 p->tag = true; a = p 对应于 P′中的 p->tag = true; a = malloc(Node)。类似地，P 中的 a->tag = false 对应于 P′中的 a->tag = false; free(a)。P′与 P 的其余部分相同。一个合理的假定是：P 和 P′这两组对应程序段中的顺序语句看成一个整体，即这些语句之间没有程序点。

下面先由形状图的语义定义 P′和 P 的程序状态之间的相容性，然后证明在运行时 P′和 P 对应程序点的程序状态之间会维持这种相容性。根据相容性定理，再基于形状图就是程序状态的抽象表示，就可以得出分析方法的正确性。

由第 2 节知，形状图是机器状态的图形表示[18]。对于像 PointerC 这样有动态存储分配的编程语言，形状图的节点（null 节点和悬空节点除外）指称机器栈单元、堆块或堆块集，边代表相应指针的值。一个没有浓缩节点和谓词节点的形状图是一个机器状态中指针型数据的图形表示（其他类型的数据与本文关系不大，在谈论机器状态时将它们忽略），而一般的形状图则是某个机器状态集的图形表示。

假定动态分配的各堆块的地址是各不相同的抽象值，再认为栈和各堆块上的存储单元分别按声明指针和域指针的名字访问，则机器的抽象状态（简称机器状态）就可由两个函数

$s_d: DecVar \to AbsValue \cup \{\mathcal{N},\mathcal{D}\}$和 $s_f: AbsValue \times FieldVar \to AbsValue \cup \{\mathcal{N},\mathcal{D}\}$

构成，其中 s_d 的定义域是声明指针名字集，它给出声明指针的抽象值。*AbsValue* 是堆块抽象地址集，*FieldVar* 是域指针名字集，s_f 给出程序能访问到的各堆块的域指针的抽象值。$\mathcal{N}$和$\mathcal{D}$是两个特殊的抽象值，$\mathcal{N}$表示相应指针的值等于 NULL，$\mathcal{D}$表示相应指针是悬空指针。

本文涉及附加链表指针，故函数 s_d 和 s_f 都需要拆分成两部分。程序 P 的状态 s 由如下 4 个函数组成：

$s_{dp}: DecPrimaryVar \to AbsValue \cup \{\mathcal{N},\mathcal{D}\}$，$s_{da}: DecAdditionalVar \to AbsValue \cup \{\mathcal{N},\mathcal{D}\}$，

$s_{fp}: AbsValue \times PrimaryFieldVar \to AbsValue \cup \{\mathcal{N},\mathcal{D}\}$，$s_{fa}: AbsValue \to AbsValue \cup \{\mathcal{N},\mathcal{D}\}$

前两个函数的定义域分别是主结构声明指针名字集和附加链表声明指针名字集。s_{fp} 的 *PrimaryFieldVar* 是构造主结构的域指针名字集；由于构造附加链表的域指针名字唯一，s_{fa} 函数略去了域指针名字集。

构成程序 P′的程序状态 s'的 4 个函数如下：

$s'_{dp}: DecPrimaryVar \to AbsValue \cup \{\mathcal{N},\mathcal{D}\}$，$s'_{da}: DecAdditionalVar \to AbsValue' \cup \{\mathcal{N},\mathcal{D}\}$，

$s'_{fp}: AbsValue \times PrimaryFieldVar \to AbsValue \cup \{\mathcal{N},\mathcal{D}\}$，$s'_{fa}: AbsValue' \to AbsValue' \cup \{\mathcal{N},\mathcal{D}\}$

为便于描述，在对应程序段 p->tag = true; a = p 和 p->tag = true; a = malloc(Node)中，若 p 的抽象值是 v，则 a = malloc(Node)使得 a 的抽象值为 v'。这是 *AbsValue′* 与 *AbsValue* 相区别的一个原因。

定义 若下面三个条件都成立，则称程序 P′的状态 s'相容于程序 P 的状态 s：

（1）函数 s'_{dp} 等于 s_{dp}，函数 s'_{fp} 等于 s_{fp}。

（2）函数 s'_{da} 和 s_{da} 的定义域相同，且对任意的 $x \in \mathrm{dom}(s_{da})$，

若 $s_{da}(x) = v$ 且 v 节点的标志域为 true，则 $s'_{da}(x) = v'$，v'节点是 v 节点的副本；

若 $s_{da}(x) = v$ 且 v 节点的标志域为 false，则 $s'_{da}(x) = \mathcal{D}$；

若 $s_{da}(x) = \mathcal{N}$或 $s_{da}(x) = \mathcal{D}$，则 $s'_{da}(x) = s_{da}(x)$。

（3）若函数 s_{fa} 的定义域是$\{v_1, \ldots, v_n\}$，则函数 s'_{fa} 的定义域是$\{v'_1, \ldots, v'_n\}$的某个子集，且对该子集的任意 y'，存在 $y \in \mathrm{dom}(s_{fa})$且 y' 节点是 y 节点的副本，

若 $s'_{fa}(y') = v'$，当且仅当 $s_{fa}(y) = v$ 且 v 节点的标志域为 true；

若 $s'_{fa}(y') = \mathcal{D}$，则存在某个 v，使得 $s_{fa}(y) = v$ 且 v 节点的标志域为 false；

若 $s'_{fa}(y') = \mathcal{N}$，则 $s_{fa}(y) = \mathcal{N}$。

这个定义体现状态 s'和 s 之间的联系包括：它们的主结构指针之间的相等关系相同，在状态 s'上虚拟链表指针之间的相等关系与状态 s 上附加链表指针（不包括节点的标志域为 false 的那些附加链表指针）之间的相等关系相同。

定理 对任何满足第 3 节程序约束的程序 P 和按本节形状分析方法调整后的相应程序 P′，在运行过程中的任何程序点，程序 P′的状态 s'相容于程序 P 的状态 s。

图 5 是该定理的图形表示。横向箭头表示程序执行，P'_i 和 P_i 表示剩余程序，最右一列程序为空表示执行结束。

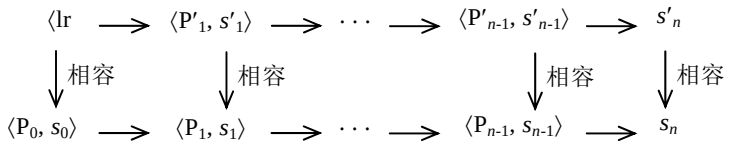

Fig. 5　Schematic diagram of the theorem

图 5 定理的示意图

证明 只要证明，在状态 s'相容于 s 的情况下，P′和 P 一步执行后的状态仍然相容即可。这需要对各种语句分情况考虑。

1）若 P 执行 p->tag = true; a = p，P′执行 p->tag = true; a =

malloc(Node)。若 p 的值是 v，则这两段代码执行后 a 分别等于 v 和 v'。s'_{fa} 的定义域扩大，但仍满足相容性定义中所说的与 s_{fa} 的定义域之间的关系。显然执行后的状态仍然相容。

2）若 P 执行 a->tag = false，P′执行 a->tag = false; free(a)。则在 P 中，语句的执行没有改变任何指针的值，而在 P′中，执行后 a 以及原先与 a 相等的指针都变成悬空指针，并且 s'_{fa} 的定义域缩小。显然执行后的状态仍然相容。

3）对于主结构上的 malloc(Node)函数调用语句，它同时扩大函数 s_{fp} 和 s'_{fp} 的定义域，并且它们仍然保持相等。显然执行后的状态仍然相容。

4）对于主结构上的 free(p)函数调用语句，它同时缩小函数 s_{fp} 和 s'_{fp} 的定义域，并且它们仍然保持相等。由于执行前 p->tag 一定等于 false，因此 s_{fa} 的定义域虽缩小，但仍满足相容性定义中所说的与 s'_{fa} 的定义域之间的关系。显然执行后的状态仍然相容。

5）对于其他的指针赋值语句，由于程序约束的限制，它只能是主结构上的操作，或者是附加链表（在 P′中是虚拟链表）上的操作，显然执行后的状态仍然相容。

6）通过归纳证明可以得出，整个函数体和函数调用语句也具有这样的性质。

据此，该定理得证。 □

推论 在函数入口，若带附加链表的数据结构满足形状不变式（即主结构的节点在附加链表上，当且仅当节点的标志域为 true，并且附加链表仅含这些节点），那么在函数出口，它仍然满足形状不变式。

这是显然的，因为上述定理证明中分析的每一种情况都不改变这种不变性。 □

形状分析实际上是对程序 P′进行的。由形状图逻辑的可靠性知道，形状分析过程中各程序点的形状图就是程序状态的一种图形表示。再由上述定理可知，对程序 P′进行形状分析所得的形状图对程序 P 是适用的，只是不知道附加链表指针和主结构指针之间的相等关系。

5 程序验证方法的扩展

我们所研发的程序验证系统原型（可从 http://kyhcs.ustcsz.edu.cn/SGL 下载）[10]的流程分成下面三步。

1、**预处理阶段** 该阶段为源代码生成抽象语法树并完成通常的静态检查。

2、**形状分析阶段** 该阶段遍历语法树，基于形状图逻辑生成各程序点的形状图。

3、**程序验证阶段** 该阶段在逻辑上可分成验证条件生成和自动定理证明两个子阶段。验证条件生成子阶段遍历语法树，根据程序员提供的、以符号断言表示的有关非指针型数据的函数前后条件和循环不变式，基于形状图逻辑，按最强后条件演算方式生成验证条件。验证条件的一般形式是 $G, T \triangleright Q \Rightarrow Q'$，其中 G 是产生验证条件那个程序点的形状图，T 是程序员提供的、以符号方式表示的谓词定义和性质引理，它们是 $Q \Rightarrow Q'$ 的证明环境，Q 和 Q'是符号断言。自动定理证明子阶段将每个 $G, T \triangleright Q \Rightarrow Q'$中的 G 转化为符号断言，再连同其他部分一起交给 SMT 求解器 Z3[14]。

第 4 节的方法保证了能对附加链表完成形状分析，但是无法把握附加链表指针和主结构链表指针之间的相等关系。它给程序验证带来如下两个主要问题：

（1）若在一个函数中既有以主结构指针又有以附加链表指针为访问路径前缀的数据域访问，则因缺少这两类指针间的相等关系而无法判断这两类数据域访问路径之间的别名关系。而消除别名是使用赋值公理的先决条件。

（2）即使在一个函数中仅有以附加链表指针为访问路径前缀的数据域访问，也会因不能确定访问的是主结构的哪个节点，因而不知道被访问节点的数据特点。

这两个问题也会出现在以附加链表指针为访问路径前缀的次结构访问场合。

程序约束（8）保证不会出现问题（1）所谈及的别名。这个约束是合理的，因为程序员为了便于分析判断程序正确与否，一般不会把通过主结构指针和通过附加链表指针对主结构节点和/或次结构的操作混合在一起。第 6 节的实例也证实了这一点。

附加链表通常用来快速找到所需访问的主结构节点，对所找到节点的数据和/或次结构的数据的特殊性

并不知晓，因此对各节点的操作代码是统一的，与节点在主结构上的位置无关。也就是不用担心问题（2）。

解决上述两个问题后，对程序验证方法的扩展可概述如下：

（1）就生成验证条件子阶段而言，不改变原程序验证方法中所使用的程序逻辑，也不改变在使用推理规则之前进行的必要的别名检查及替换、谓词应用的展开和量化断言的展开。需要修改的仅是别名查找方法。先举例说明原有的别名查找和替换方法。若符号断言中有 p->next->data == 10，下一条语句是对 p->next 的赋值，则需要先将 p->next->data 用其别名替换，然后再根据该赋值语句调整形状图。在形状图上从 p 和与 p 同类型的声明指针开始搜索，若发现指针 p->next 等于 q->next，则把 p->next->data 换成 q->next->data。赋值后符号断言中有 q->next->data == 10。当含有附加链表时，由于主结构声明指针和附加链表声明指针虽是同类型指针，但按第 4 节介绍的形状分析方法，从它们开始在形状图上不可能得到指向同一个节点的指针，因此需要把主结构声明指针和附加链表声明指针当成不同类型的指针来使用别名查找方法。这个修改还可提高别名查找的效率。

（2）在自动定理证明子阶段，在证明验证条件 $G, T \triangleright Q \Rightarrow Q'$之前，有时可以先将验证条件化简，并且无须把 G 的各形状子图都转化为符号断言。例如，一个处理主结构的函数 A，它调用处理附加链表的函数 B，那么 A 的前后条件都包含描述附加链表性质的断言，但 A 函数体中除了调用 B 的语句外，其余语句并不处理附加链表。在为 A 的其余语句产生的验证条件中，Q 和 Q'中描述附加链表性质的断言往往完全一样，这时可将它们从 Q 和 Q'中略去，并且 G 中有关附加链表形状子图的信息也不必转化为符号断言。这样可以提高自动定理证明的效率。第 6 节的例子有这样的特点。

6 验证实例

我们将 GNU C Library 中采用请求队列实现异步 I/O 操作的代码（文件 aio_misc.c）进行简化，略去其中与计算请求优先级和创建进程等有关的代码，只留下处理数据结构的代码。用 PointerC 描述的数据结构类型定义见图 3，主结构节点类型 Requestlist 和次结构节点类型 Requestqueue 都含 I/O 操作的描述数据（在图 3 中被略去）。

函数 void aio_enqueue_request(int fildes, int prio)是该例的主要函数之一，其代码见图 6。它对数据结构操作的功能是：若主链表（由全局的指针 requests 指向）中没有文件描述字 fildes，则在主链表中新增 1 个表元，并且为它构成只有 1 个表元的次链表，该表元的优先级为 prio，还要把主链表上该新增表元加入附加链；若主链表中已经存在文件描述字 fildes 的表元，则在其次链表上增加优先级为 prio 的表元。

该函数的前条件如下：

$$\begin{aligned}&\text{assertion length(runlist, next\_run) == m} \wedge \text{length(requests, next\_fd) == n} \wedge \\ &\quad \forall i{:}1..m\text{-}1.(\text{runlist}(\text{->next\_run})^{i-1}\text{->abs\_prio} >= \text{runlist}(\text{->next\_run})^{i}\text{->abs\_prio}) \wedge \\ &\quad\quad \forall i{:}1..n\text{-}1.(\text{requests}(\text{->next\_fd})^{i-1}\text{-> aio\_fildes} < \text{requests}(\text{->next\_fd})^{i}\text{-> aio\_fildes}) \wedge \\ &\quad\quad\quad \forall i{:}1..n.\text{Sorted}(\text{requests}(\text{->next\_fd})^{i-1}\text{->request\_ptr})\end{aligned} \tag{1}$$

其中 runlist 是全局的附加链表指针，m 和 n 是逻辑变量，length 是系统内建的求表长的函数，Sorted 是程序员提供的单向链表有序性谓词[20]。由于各次链表的长度不一样，用 Sorted 谓词可避免断言中出现存在量词。requests(->next_fd)^{i}-> aio_fildes 表示 requests->next_fd-> …->next_fd-> aio_fildes，其中->next_fd 有 i 次。该函数的后条件是两种情况的析取，分别是与前条件完全一样和仅区别在两个链表的长度各增 1。全局指针 runlist 仅出现在断言中，说明除了其中的函数调用语句外，函数体其余语句不涉及附加链表。

该函数有一个循环，其循环不变式如下，它由未搜索到主链表表尾和搜索到表尾两种情况组成。

$$\begin{aligned}&\text{runp != NULL} \wedge \text{j >= 0} \wedge \text{j < n-1} \wedge \\ &\quad \forall i{:}1..m\text{-}1.(\text{runlist}(\text{->next\_run})^{i-1}\text{->abs\_prio} >= \text{runlist}(\text{->next\_run})^{i}\text{->abs\_prio}) \wedge \\ &\quad \forall i{:}1..j.(\text{requests}(\text{->next\_fd})^{i-1}\text{-> aio\_fildes} < \text{requests}(\text{->next\_fd})^{i}\text{-> aio\_fildes}) \wedge \\ &\quad \text{last->aio\_fildes} < \text{runp->aio\_fildes} \wedge \\ &\quad \forall i{:}1..j.\text{Sorted}(\text{requests}(\text{->next\_fd})^{i-1}\text{->request\_ptr}) \wedge \text{Sorted(last->request\_ptr)} \wedge\end{aligned}$$

```
void aio_enqueue_request(int fildes, int prio) {
    Requestlist* runp; Requestlist* last; Requestlist* newp1; Requestqueue* newp2; int j;
    Requestlist* addp; /*@: ADDITIONAL */ //标注指明 addp 是附加链指针
    newp1 = NULL; addp = NULL; newp2 = NULL; last = NULL; runp = requests;
    if (runp != NULL && runp->aio_fildes < fildes) {
        last = runp; runp = runp->next_fd; j = 0;
        while (runp != NULL && runp->aio_fildes < fildes){ /* 在主链表中查找 fildes */
            last = runp; runp = runp->next_fd; j = j +1;
        }
    }
    if(runp == NULL || runp->aio_fildes > fildes){
        /* 在主链表中增加一个表元 */
        newp1 = malloc(Requestlist); newp1->running = false; newp1-> aio_fildes = fildes;
        if (last == NULL){
            newp1->last_fd = NULL; newp1->next_fd = requests;
            if (requests != NULL){requests->last_fd = newp1;}
            requests = newp1;
         }else{
            newp1->next_fd = last->next_fd; newp1->last_fd = last; last->next_fd = newp1;
            if (newp1->next_fd != NULL){ newp1->next_fd->last_fd = newp1;}
        }
        /* 在次链表中增加一个表元 */
        newp2 = malloc(Requestqueue); newp2->abs_prio = prio; newp2->next_prio = NULL;
        newp1->next_prio = newp2; newp1->abs_prio = prio;
        /* 把主链表上的这个表元加到附加链表上 */
        newp1->running = true; addp = newp1; addp->next_run = NULL;
        add_request_to_runlist(addp);
    } else { /* 仅在次链表中增加一个表元 */
        runp->request_ptr = add_request_to_queuelist(runp->request_ptr, prio);
        if (runp->abs_prio != runp->request_ptr->abs_prio) {
            runp->abs_prio = runp->request_ptr->abs_prio;
            if (runp-> running == true) {modify_runlist(fildes);} /* 因主链节点的 abs_prio 被修改，*/
        }                                                          /* 调整其在附加链表中的位置 */
    }
}
```

Fig. 6 Simplified code of function aio_enqueue_request

图 6 函数 aio_enqueue_request 的简化代码

$$
\begin{aligned}
&\forall i{:}1..n\text{-}j\text{-}2.(runp(\text{->next\_fd})^{i-1}\text{-> aio\_fildes} < runp(\text{->next\_fd})^{i}\text{-> aio\_fildes}) \wedge \\
&\forall i{:}1..n\text{-}j\text{-}1.\mathrm{Sorted}(runp(\text{->next\_fd})^{i-1}\text{->request\_ptr}) \vee \\
&runp == \mathrm{NULL} \wedge j >= 0 \wedge j == n\text{-}1 \wedge \\
&\forall i{:}1..m\text{-}1.(runlist(\text{->next\_run})^{i-1}\text{->abs\_prio} >= runlist(\text{->next\_run})^{i}\text{->abs\_prio}) \wedge \\
&\forall i{:}1..j.(requests(\text{->next\_fd})^{i-1}\text{-> aio\_fildes} < runlist(\text{->next\_fd})^{i}\text{-> aio\_fildes}) \wedge \\
&\forall i{:}1..n.\mathrm{Sorted}(requests(\text{->next\_fd})^{i-1}\text{->request\_ptr})
\end{aligned}
$$

函数 void aio_remove_request (int fildes, int prio, bool all)是另一个主要函数。其对数据结构的操作是：删除主链表上文件描述字为 fildes 的表元所挂的次链表上优先级为 prio 的 I/O 操作，或若形参 all 为 true 时则删除所有小于等于 prio 的 I/O 操作。若该次链表无剩余 I/O 操作了，则删除主链表上该表元。在删除该表元时，若它在附加链表上，则应该先把它从附加链表上摘下。该函数的前后条件和循环不变式类似上述插入的情况。

该例共有 7 个函数，其余 5 个函数比较简单，分别是次链表的插入函数和删除函数、附加链表的插入函数和删除函数、调整附加链表一个表元次序的函数。从该例程序的编写和验证知道，程序员为保证代码的清晰，一般不会违反第 3 节的程序约束。限于篇幅，不能把所有函数及断言都列在这里。

验证这些函数的统计数据见表 1。其中的验证时间是在 Windows 7 PC，Intel Core i5-2400 3.1GHz CPU 和 4G 内存的机器上实测获得。

Table 1 Statistical data about each verified function

表 1　各函数的验证统计数据

函数名	代码和断言（行）	验证条件(个)	循环(个)	验证时间(ms)
add_request_to_queuelist，次链表增加表元	37	3	1	216
remove_request_from_queuelist，次链表删除表元	89	5	2	495
add_request_to_runlist，附加链表增加表元	41	3	1	185
remove_request_from_runlist，附加链表删除表元	58	3	1	309
modify_runlist，调整附加链表一个表元的次序	16	3	0	139
aio_enqueue_request，主链表增加表元	150	6	1	1126
aio_remove_request，主链表删除表元	118	8	2	1130
合计（含未统计在各函数中的代码和运行时间）	565	31	8	34424

7　相关工作比较

程序分析广泛用于编译器的代码优化中。由于程序分析通常是一种近似分析，因此代码优化的策略是：程序分析所获得的信息虽不精确但只要对优化是稳妥的就可行，信息不精确虽会失去一些优化机会但稳妥能保证所完成的优化是可靠的。这对用户是可接受的，因为代码优化旨在改进代码而非一定要获得最优代码。但是，对于程序验证前期所需的程序分析来说，若所得信息不精确，则难以基于这些信息开展后期程序其他性质的验证，因此需要各种措施来保证程序分析的精确性。

一般的形状分析也是一种近似分析，它试图发现可以当作指针程序不变式的数据结构的形状。已经有大量基于形状图来开发形状分析的研究，它们都用图的节点来表示堆块。特别是，对于归纳定义的数据结构，它们大都用概括节点（summary node）来使数量可能无界的堆块归类到有限的图节点中[11-14]。对内存状态的这种近似导致丢失了归纳数据结构的形状信息。一种改进精度的方式是将形状图的概括节点与文法联系起来，用有限的文法产生式来精确描绘出运行时的堆结构[15]。

我们所设计并实现的基于形状图逻辑的形状分析是一种精确的指针分析[5, 6]，它得益于程序中形状声明的提示和对指针操作的限制，并且它通过使用浓缩节点和谓词节点以及它们附带的节点个数信息来得到可能无界的数据结构的精确有穷图形表示。

我们的方法与他人方法的重要区别是：形状图和形状图逻辑同时用于形状分析和程序验证两个阶段，其最大特点是把形状图看成图形表示的指针相等断言，程序推理规则中有关指针语句的部分也是基于形状图设计并直接在形状图上进行演算和推理的。Chang 和 Rival 等[16, 17]也基于形状图来进行形状分析，但是他们用边来抽象内存堆块。在其形状图上，完全检查边（complete checker edge）和部分检查边（partial checker edge）分别相当于我们的谓词节点和浓缩节点。更大的区别是他们只把形状图看作符号断言的一种便于理解的表示，程序分析的演算和推理仍在符号断言上进行。在这两篇文章的分析统计表上，未见处理循环链表的例子。Madhusudan 等[18, 19]把形状图符号化，他们的内存印迹（footprint）由符号堆和递归树逻辑 DRYAD 的公式组成，前者相当于我们的形状图，后者相当于形状图所代表的数据结构所满足的性质。这两篇文章的符号堆没有适用于循环程序的浓缩节点概念，因而只支持递归程序，也未见处理循环链表的例子。

分离逻辑是验证堆操作程序并且无需形状图帮助的最流行的手工或半自动推理的程序逻辑[20]，其缺点是难以实现自动验证。最近也出现一些小的可判定片段[21, 22]。其中[21] 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向链表的程序。[22]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相的理论组合判定方法。分离逻辑的另一优势是它适用于汇编语言级的程序推理[23]。

我们的方法的缺点是只适用于指针相等关系完全确定的数据结构，文章[9]虽扩展到指针相等关系不确定的场合，但它只适用于手工验证。本文把形状图逻辑推广到可用于指针相等关系不完全确定的一类特殊数据结构。为适用于程序验证，本文不采用传统的近似形状分析方法，而是只收集确定的指针相等信息，放弃程序员实际不需要的不确定的指针相等信息。该方法虽然给编程带来一些约束，但是它提供了自动验证的可能。尚未在文献上见到介绍能用于操作带附加链表的数据结构的程序的自动验证工具。

8 总结

在基于逻辑推理的程序验证方法的研究中，我们集中在指针类型的专用逻辑设计及指针程序验证工具的研发上。和我们先前的研究工作相比，本文代表我们已经开始考虑操作复杂数据结构的程序的自动验证，有附加链表只是复杂数据结构的一种情况。

下一步考虑怎样在目前基础上设计一个实用性大大提高的形状图逻辑。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。本文前言谈及的 5 种基本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，增加复杂形状的构造方式。例如，可以考虑确定和部分确定的内部附加指针（例如带父节点指针的二叉树和跳表）和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向链表，附加指针指向链表的最后一个节点）。

用基本形状集加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。

参考文献

[1] Yannick Moy, Emmanuel Ledinot, Hervé Delseny, Virginie Wiels, and Benjamin Monate. Testing or Formal Verification: DO-178C Alternatives and Industrial Experience. *IEEE Software*, 30(3):50-57, 2013.

[2] Yiyun Chen, Baojian Hua, Lin Ge and Zhifang Wang. A pointer logic for safety verification of pointer programs. *Chinese Journal of Computers*, 31(3):372-380, 2008.

[3] Yiyun Chen, Zhaopeng Li, Zhifang Wang, and Baojian Hua. Pointer logic for verification of pointer programs. *Journal of Software*, 21(3):415-426, 2010

[4] Yiyun Chen, Lin Ge, Baojian Hua, Zhaopeng Li, Cheng Liu, and Zhifang Wang. A pointer logic and certifying compiler. *Frontiers of Computer Science in China* , 1(3), pp. 297-312, 2007.8

[5] Zhaopeng Li, Yu Zhang, and Yiyun Chen. A shape graph logic and a shape system. *Journal of Computer Science and Technology*. 28(6):1063-1084, 2013.11.

[6] Yu Zhang, Yiyun Chen, and Zhaopeng Li. Theorem proving for a theory of shape graphs. *Submitted to Chinese Journal of Computers*. (in Chinese with English abstract)

[7] Yu Zhang, Zhaopeng Li, and Yiyun Chen, Design of assertion languages in program verification systems. *Submitted to Journal of Software.* (in Chinese with English abstract)

[8] Zhitian Zhang, Zhaopeng Li, Yiyun Chen, and Gang Liu. An Automatic Program Verifier for PointerC: Design and Implementation. *Journal of Computer Research and Development*, 50(5):1044-1054, 2013. (in Chinese with

English abstract)

[9] Hongjin Liang, Yu Zhang, Yiyun Chen, Zhaopeng Li, and Baojian Hua. Pointer logic dealing with uncertain equality of pointers. *Journal of Software*, 21(2):334-343, 2010. (in Chinese with English abstract)

[10] Zhaopeng Li, Yu Zhang, Yiyun Chen, Yanhui Song, and Jianchao Meng. The verifier prototype system of PointerC based on the Shape Graph Logic and Shape System. URL: http://kyhcs.ustcsz.edu.cn/SGL, May. 2014.

[11] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *ACM Transactions on Programming Languages and Systems*, 20(1):1-50, 1998.

[12] Tal Lev-Ami, Thomas W. Reps, Shmuel Sagiv, Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *Proceedings of the International Symposium on Software Testing and Analysis* (ISSTA 2000), pages 26-38, 2000.

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

[14] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proc. of PLDI'90*, pages 296-310, June 1990.

[15] Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. Automatic Verification of Pointer Programs Using Grammar-Based Shape Analysis. In *Proc. of ESOP '05*, LNCS vol. 3444, pages 124-140. Springer, 2005.

[16] B. E. Chang, X. Rival, and G. C. Necula. Shape analysis with structural invariant checkers. In *Proc. of SAS'07*, pages 384-401, 2007.

[17] B. E. Chang and X Rival. Relational inductive shape analysis. In *Proc. POPL'08*, pages 247-260, 2008.

[18] P. Madhusudan, G. Parlato, and X. Qiu. Decidable logics combining heap structures and data. In *Proc. POPL'11*, ACM Press, 2011.

[19] P. Madhusudan, X. Qiu, and A. Stefanescu. Recursive Proofs for Inductive Tree Data-Structures. In *POPL'12*, pages 123-135. ACM, 2012.

[20] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS'02*, pages 55-74, July 2002.

[21] Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI'11*, pages 556-566. *ACM, 2011.*

[22] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating Separation Logic Using SMT. In *Proc. CAV'13* vol. 8044 of LNCS, 2013.

[23] Jonas Braband Jensen, Nick Benton, and Andrew Kennedy. High-Level Separation Logic for Low-Level Code. In *POPL'13, pages 301-314. ACM,* 2013.

附中文参考文献：

[2] 陈意云，华保健，葛琳，王志芳. 一种用于指针程序安全性证明的指针逻辑.计算机学报，2008, 31(3):372-380.

[3] 陈意云, 李兆鹏, 王志芳, 华保健. 一种用于指针程序验证的指针逻辑. 软件学报, 2010, 21(3):415-426.

[6] 张昱, 李兆鹏, 陈意云. 形状图理论的定理证明. 已投计算机学报. 见 thhp://staff.ustc.edu.cn/~yiyun.

[7] 张昱, 李兆鹏, 陈意云. 程序验证系统中断言语言的设计.已投软件学报.见 thhp://staff.ustc.edu.cn/~yiyun.

[8] 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，50(5): 1044-1054, 2013.

[9] 梁红瑾, 张昱, 陈意云, 李兆鹏, 华保健. 处理指针相等关系不确定的指针逻辑. 软件学报，2010, 21(2):334-343.

研究背景

随着国家、社会和日常生活对软件系统的依赖程度日益增长，复杂软件系统的正确、安全和可靠等对安全攸关的基础设施和应用是至关重要的。安全攸关软件的高可信成了保障国家安全、保持经济可持续发展和维护社会稳定的必要条件。

形式验证是提高软件可信程度的重要方法，它包括抽象解释、模型检测和演绎验证等途径，它们在工业界已经逐步得到应用，尤其是前两者。演绎验证使用形式方法对软件系统进行数学推理，其中通常要用到像 Isabelle/HOL 或 Coq 这样的定理证明软件。虽然基于逻辑推理的一些工具已在实验室研发出来，但是尚无可供工业界使用的产品问世。

指针程序的分析和验证一直是个难题，也一直是国际广为关注的一个研究热点。在对操作易变数据结构的程序的验证方法研究中，我们为指针类型设计过两种专用逻辑。第一种称为指针逻辑，在其断言语言中，指针型访问路径的相等断言用访问路径集合来表示，同一个集合中的指针都相等。第二种专用逻辑称为形状图逻辑，在其断言语言中，一种专门设计的形状图直接被看成指针型访问路径的相等断言的图形表示，指向同一个节点的指针都相等。基于这两种断言语言，我们设计了两种 Hoare 逻辑风格的推理规则。经过多方面和多年的努力，我们设计和实现的 PointerC 语言的程序验证系统原型，已经能够自动验证操作 splay 树, treap, AVL 树和 AA 树等较为复杂数据结构的程序。该系统的一个重要缺点是，它只能用于操作单向链表、循环单向链表、双向链表、循环双向链表和二叉链表示的二叉树这 5 种易变数据结构的程序。

本文基于形状图逻辑，对于部分节点由节点中的附加指针构成附加链表的数据结构，提出一种操作这种数据结构的程序的验证方法，并证明了其中形状分析的正确性。