形状图逻辑和形状系统

(中国科学技术大学 计算机科学与技术学院，安徽 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：指针程序的分析和验证一直是个难题，本文用形状图逻辑和形状系统来解决其中的困难，并采取先用形状分析构建各程序点的形状图，再借助形状图进行其它性质验证的两阶段方式完成指针程序的验证。所实现的系统原型可自动验证使用 splay tree, treap, AVL tree 和 AA tree 等数据结构的程序。

本文提出一种直接把形状图作为程序中指针断言的形状图逻辑。它是 Hoare 逻辑的一种扩展，可用于对操作易变数据结构的指针程序的分析和验证。用形状图表示指针断言的好处是，便于程序验证阶段获取所需的指针信息。其次，提出形状系统概念。它要求程序员在声明含自引用的结构体类型时，指明其指针域参与构建的数据结构的形状，在形状分析阶段的形状检查据此排除所构建的形状偏离程序员期望的程序。其好处是可免去程序员为程序验证提供有关指针的函数前后条件和循环不变式。此外，还提出利用形状图来消除访问路径别名，让程序验证阶段对形状以外性质的验证仍用 Hoare 逻辑的规则进行推理的方法。该方法使得程序验证阶段生成的验证条件仍可以用通常的定理证明器来证明。

关键词： 形状图逻辑、形状系统、程序验证、形状分析、指针逻辑

1、引言

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（包括 safety 和 security）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。粗略地说，软件的形式验证有两种途径。The first approach is model checking, which consists of a systematically exhaustive exploration of the mathematical model。模型检测方法已经在工业界逐步得到应用。The second approach is logical inference. It consists of using a formal version of mathematical reasoning about software systems, usually using theorem proving software such as Isabelle/HOL[1] or Coq[2] theorem provers. 在这种途径中，大部分的研究围绕采用某种演算来产生验证条件，然后用某个定理证明器来证明验证条件，如 Ynot[3]、Spec#[4]和 ESC/Java[5]。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生成步骤，如 smallfoot[6]和 jStar[7]。还有的研究采用经严格证明的变换，从抽象规范逐步求精得到具体程序，如 Perfect Developer[8]。虽然这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。究其原因，根源在于自动定理证明方面的困难。因为不管是验证条件的证明、循环不变式的推断、访问路径（访问堆变量的表达式）的别名判断、断言语言的表达能力和领域专用逻辑的设计等，最终都受到自动定理证明器的能力的影响。

在研究自动定理证明技术的同时，也应该考虑怎样降低对自动定理证明器的能力的期望。例如，设计新的编程语言机制来提高合法程序的门槛，以排除部分有逻辑错误的程序。还有，采用程序分析方法来收集程序信息，用所得信息来支持程序验证。这些都有可能减轻自动定理证明器的负担。本文介绍在指针程序验证方法的研究中，我们采用先形状分析，后程序验证所取得的进展。

第一，提出一种把形状图直接作为指针相等断言集以及指针有效性断言集的方法，并基

于此设计出形状图逻辑。形状图逻辑是 Hoare 逻辑的一种扩展，主要增加了对指针操作语句的推理规则。该逻辑可用于操作易变数据结构的指针程序的分析和验证。

基于对指针算术运算和取地址运算（&）等的限制，本文的形状图可以精确表达相应程序点指针（包括静态声明的指针和动态分配的结构体变量中的域指针）之间的相等关系和各指针有效与否（有指向对象的指针称为有效指针），并可用来查询访问路径的别名等。在形状分析和程序验证两阶段都把形状图看成图形表示的指针相等断言集，并都用形状图逻辑来进行推理。形状分析阶段主要用其中指针操作语句的规则来构建各程序点的形状图；程序验证阶段则利用形状图提供的信息来验证形状以外的其他程序性质，如二叉树的有序性。可以说，这是把指针性质的证明从程序验证中剥离出来，先行在形状分析阶段完成。两个阶段都采用形状图逻辑有利于整个程序逻辑的可靠性的证明。

用形状图作为指针相等断言的好处是，便于分析时从指针操作语句之前的指针断言集获得其后的指针断言集，也便于程序验证时获取所需的指针信息。例如，指向节点的指针是否多于 1 个（用于防止内存泄漏），2 条访问路径是否互为别名（用于消除访问路径别名）。

第二，提出形状系统概念，粗略地说它类似于类型系统。本文为一个类 C 小语言 PointerC 设计一种形状系统，它包括所允许构建的形状及其严格定义、形状推断规则和形状检查规则。形状系统要求程序员声明含自引用的结构体类型（指其中有这样的域：它们的类型是该结构体类型的指针类型）参与构建的形状。形状分析阶段利用形状推断规则来推断各结构体变量链接成的形状，并依据形状检查规则来判断在被关注的程序点，所推断出的形状与程序员的声明是否一致，由此拒绝没有构造出（或操作在）程序员所声明形状的程序，减少分析和验证要应付的情况。

形状系统带来的明显好处是，免去程序员为程序验证提供作为函数前后条件和循环不变式一部分的函数前后形状图和循环不变形状图。由此，用形状图作为指针相等断言不会给程序员添加麻烦。

第三，提出利用形状图来消除访问路径别名，使形状以外性质的验证仍可用 Hoare 逻辑的规则进行推理的方法。Hoare 逻辑的一个重要局限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。倘若能够保证在使用 Hoare 逻辑的赋值公理时，所涉及的语句和断言中没有别名，则该规则的使用是可靠的。在程序验证阶段，形状图被用来作为判断和消除访问路径别名的依据，先消除别名，然后再用赋值公理。

这种做的好处是，避免像 Hoare 逻辑的某些扩展（如分离逻辑）那样需要专门的定理证明器来证明验证条件。

第四，实现了一个指针程序验证系统的原型。该原型可以验证易变数据结构上较为复杂的程序，如 sorted circular doubly-linked list, binary search tree, splay tree, treap, AVL tree 和 AA tree 的插入和删除函数。

本文组织如下。第 2 节介绍形状图并把形状图看成断言，第 3 节介绍形状图逻辑，第 4 节介绍形状系统。第 5 节介绍为 PointerC 语言开发的程序验证系统的原型。第 6 节和相关研究工作进行比较。第 7 节是总结。

2、形状图作为指针断言

形状图是描述程序中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，它不仅准确地表达了指针之间的相等关系，还可用来判断访问路径的别名等。

本节主要定义形状图及其语义，并把形状图看成有关指针的断言。

2.1 形状图的语法

和图论中的有向图不一样，形状图的顶点有六种形式，见图 1。它们主要用来表示程序

操作的栈单元或堆块，在此称为节点。下面是六种节点的名称和语法。

（1）声明节点：圆节点。

（2）结构节点：矩形节点。

（3）null 节点：虚线边框的矩形节点，中间含字母N。

（4）悬空节点：虚线边框的矩形节点，中间含字母Δ。

（5）浓缩节点：灰色矩形节点，下侧有表达式 e 以及约束 e 取值范围的断言 a。e 是仅使用常量和变量的整型线性表达式，a 是这类表达式的关系运算的逻辑合取式。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。

（6）谓词节点：矩形节点，中间含字母Π，节点下侧有作为谓词名字的标识符 name，还有与浓缩节点中一致的表达式 e 和断言 a。

声明节点　结构节点　null 节点　悬空节点　浓缩节点　谓词节点

图 1 形状图的各种节点

形状图的有向边上都有标识符作为其标记。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。

（2）结构节点和浓缩节点：有入边和出边，各出边的标记不相同。

（3）null 节点、悬空节点和谓词节点：有入边，没有出边。

定义 1 （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（diagram）是**形状图**。

（2）若形状图 $G_1, G_2, \ldots, G_n$ 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∧连接的 $G_1 \wedge G_2 \wedge \ldots \wedge G_n$ 也是形状图。

（3）若形状图 $G_1, G_2, \ldots, G_n$ 的声明节点出边标记集都相同，则逻辑析取符号∨连接的 $G_1 \vee G_2 \vee \ldots \vee G_n$ 也是形状图。

本文中符号 G 仅用来表示不含符号∨的形状图，并且不含符号∧的形状图 G 被称为**形状子图**。在介绍形状图语义时可知，在 $G_1 \wedge G_2$ 中，G_1 和 G_2 各代表程序状态的不同部分，而在 $G_1 \vee G_2$ 中，G_1 和 G_2 代表不同的程序状态。

受编程语言类型系统的约束，本文涉及的形状图只是定义 1 确立的形状图集的子集，因为类型系统保证源于不同结构体类型的结构节点在形状图上不会相邻。

图 2 是两个表示单向链表的形状图。在讨论形状图的语义后会知道，下面程序片段（假定 head 指向的单向链表至少有一个表元）

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

的循环不变特性由图 2(a)概括，其中 head 和 ptr 所指向的浓缩节点分别代表 m 和 n（$m, n \geq 0$）个表元。由此可知，浓缩节点的 e 代表被浓缩的结构节点的个数；指向 null 节点的有向边是 NULL 指针。在循环体中第 1 个语句之前的程序点，单向链表的特点由图 2(b)概括。

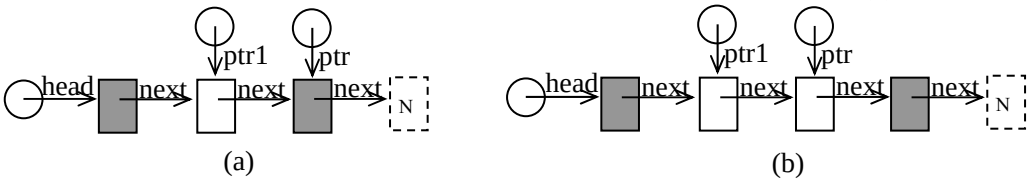


图 2　形状图的两个例子

2.2 数据结构的形状图

图 3 给出基于形状图定义的单链表（包括单向链表和循环单向链表）、双链表（包括双向链表和循环双向链表）和二叉树，其中最左边的 dlist(s, e, a)等符号表示是为了便于在文中的引用。在这些形状图中，边上的标记都是占位符，可根据程序的需要来选择名字。

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。为节约篇幅，list(s)、c_list(s)和 c_dlist(s)定义的形状图没有在图 3 列出。

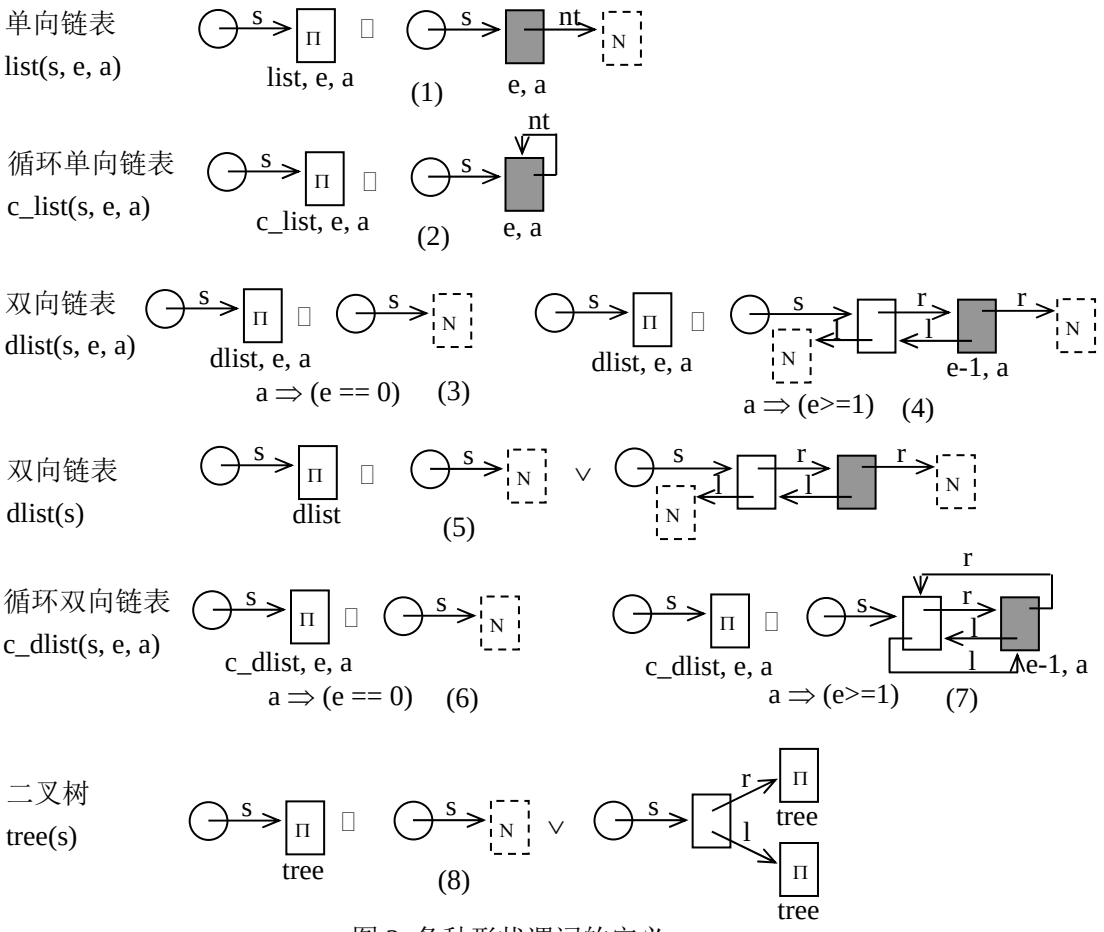


图 3 各种形状谓词的定义

定义 2 图 3 定义式（包括因篇幅限制未列出的定义式）左右两边出现的形状子图，都称为**数据结构最简**（minimal）**形状图**。

2.3 形状图的变换规则

在给出变换规则之前，需先定义窗口，它描述形状子图上被关注的那部分。

定义 3 一个形状图的某子图上用点划线方框明示并满足下面条件的部分称为**窗口，**形状图的其余部分称为窗口的**上下文**。窗口需要满足的条件如下：

（1）形状图的各节点处于窗口或上下文中，不得与窗口的方框边界相交。

（2）窗口内各节点之间的边都处于窗口中；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，它们的一份拷贝在上下文中。

窗口 *W* 和上下文 *G*[ ]的匹配就是检查穿越 *W* 边界的边和 *G*[ ]中的拷贝边能否重合，重合后得到的形状图用 *G*[*W*]表示。

在下述规则中出现的窗口有这样一种缩写：若要求从窗口外指向窗口内某节点至少要有一条边（如图 4 的 p_1），则可能还有的其他边（可以是 0 条）用一条标记为 p_k 的边统一表示。

下面首先介绍等价变换规则，它们用来表示保持语义等价的形状图变换规则。语义等价性在定义形状图的语义后证明。

1、单链表的等价规则

（1）基本规则

- 基于单链表谓词定义的规则。例如，由图 3 的 list(s, e, a)定义得到的规则见图 4 的规则(1)。
- 添加或取消 e = 0 的浓缩节点的规则。例如，图 4 的规则(2)和(3)。针对规则(2)，把规则(2)右部窗口中的浓缩节点和结构节点交换一下位置，也是一条规则。另外，把该规则中的结构节点改成谓词、null 或悬空节点（它们都没有 nt 出边），结果也是规则。
- e > 0 的浓缩节点的展开和折叠规则。例如，图 4 的规则(4)。把规则(4)右部窗口中的浓缩节点和结构节点交换一下位置，也是一条规则。
- 无约束浓缩节点的展开与折叠规则。例如图 4 的规则(5)和(6)。规则(5)针对由一个无约束浓缩节点构成的循环单向链表。规则(6)右部第 1 个窗口中的结构节点和浓缩节点交换一下位置，结果也是一条规则。仿照规则(6)可以得到浓缩节点右边是其他节点的规则。
- 谓词节点的展开与折叠规则。图 4 的规则(7)和(8)是这种规则。

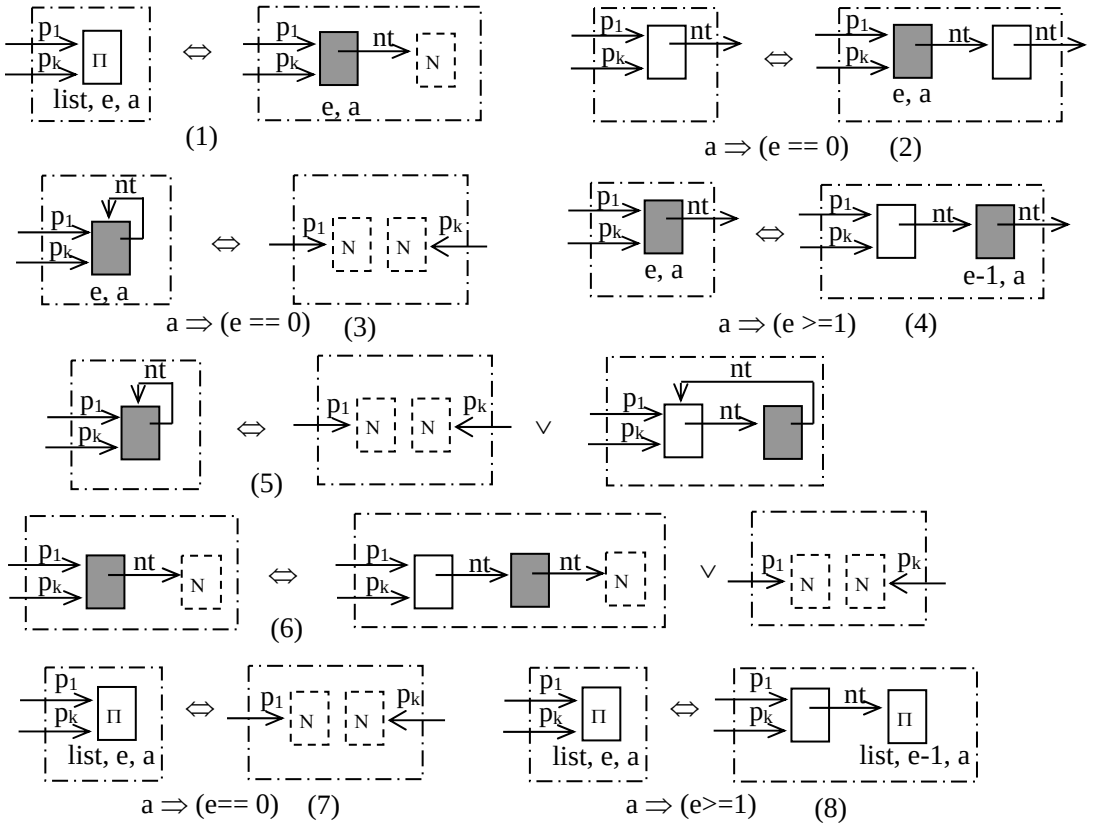


图 4 单链表的部分等价规则

（2）导出规则

- 分别带 e_1 与 a_1 和 e_2 与 a_2 的两个相邻浓缩节点等价于一个带 e_1+e_2 与 $a_1 \wedge a_2$ 的浓缩节点的规则，两个相邻的无约束浓缩节点等价于一个无约束浓缩节点的规则。
- 单个结构节点等价于带 1 与 true 的浓缩节点的规则。
- 可以作为 list(s, e, a)谓词另一种归纳定义的规则。将图 4 规则(7)和(8)的右部用析取符号链接并略去多余的 p_k 及相应边，可作为 list(s, e, a)定义的右部。list(s)也有类似的规则。

2、双链表的等价规则

（1）基本规则

- 基于双链表谓词定义的规则。
- 添加或取消 e = 0 的浓缩节点的规则。例如图 5 的规则(1)。
- e > 0 的浓缩节点的展开和折叠规则。例如图 5 的规则(2)。

由于 l 和 r 的对称性，图 5 的规则(1)和(2)包括了浓缩节点处于结构节点右边的情况。

- 在带 e 和 a 的浓缩节点作为双向链表边缘节点时的展开与折叠规则。它们和图 5 的规则(1)和(2)略有区别，图 5 的规则(3)是其中的一条。
- 无约束浓缩节点的展开与折叠规则。例如图 5 的规则(4)。浓缩节点右边是其他节点以及浓缩节点本身是双向链表边缘节点的规则类似。

（2）导出规则

类似于单链表的情况，但没有可作为归纳定义的等价规则。

双链表定义和变换规则维持一个重要原则：除标记为 r 和 l 的边外，无其他边指向浓缩节点。若有这种边，则浓缩节点展开时，不知该边应指在展开后的最左还是最右结构节点上。

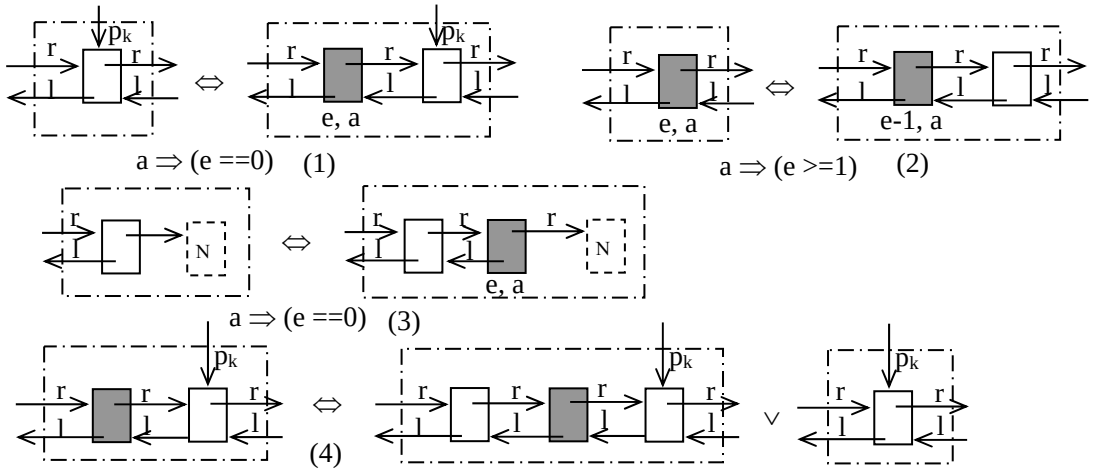


图 5 双链表的部分等价规则

3、二叉树的等价规则

除了基于 tree(s)谓词定义的规则外，二叉树中有关浓缩节点的规则类似于单向链表的规则，区别在于每个结构节点和浓缩节点多了一条指向谓词节点的边。图 6 的规则(1)和(2)是其中的两条。

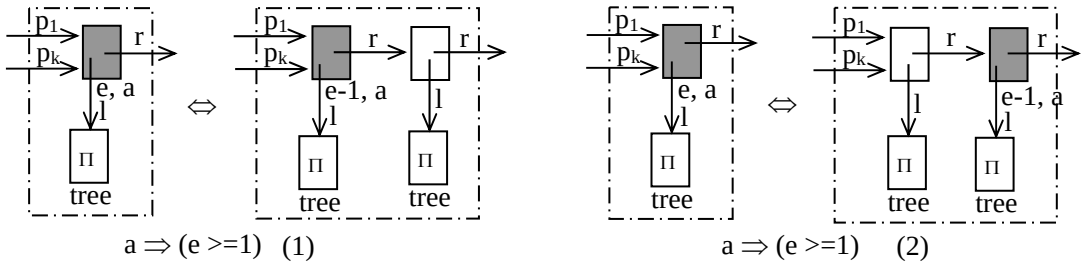


图 6 二叉树的部分等价规则

从上述浓缩节点的展开与折叠规则可知，浓缩节点是若干相邻同类结构节点（出边条数和标记都一样）的浓缩表示。

4、对浓缩节点的 e 和 a 进行替换的等价规则

图 7 的规则(1)和(2)分别是单链表和双链表浓缩节点在两边的 e 和 a 的变量集一样时的规则。浓缩节点是双向链表边缘节点时的规则以及二叉树浓缩节点的规则可类似地给出。

$(a_1 \Rightarrow (e_1 == e_2 \wedge a_2)) \wedge (a_2 \Rightarrow (e_2 == e_1 \wedge a_1))$ (1)　　$(a_1 \Rightarrow (e_1 == e_2 \wedge a_2)) \wedge (a_2 \Rightarrow (e_2 == e_1 \wedge a_1))$ (2)

图 7 对浓缩节点的 e 和 a 进行变换的部分等价规则

使用等价规则，可以对形状图进行变换。例如，若有等价规则 $W_1 \Leftrightarrow W_2$ 和上下文 $G[]$，则将该规则用于 $G[W_1]$可得到 $G[W_2]$，反之亦然。写成 $G[W_1] \Leftrightarrow G[W_2]$。类似地，由规则 $W \Leftrightarrow W_1 \vee W_2$ 知道 $G[W] \Leftrightarrow G[W_1] \vee G[W_2]$。

下面再考虑蕴涵规则，它们用来表示保持语义蕴涵的形状图变换规则。蕴涵性在定义形状图的语义后证明。

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。

2、若等价规则 $W_1 \Leftrightarrow W_2$ 的副条件是$(a_1 \Rightarrow (e_1 == e_2 \wedge a_2)) \wedge (a_2 \Rightarrow (e_2 == e_1 \wedge a_1))$，则有蕴涵规则 $W_1 \Rightarrow W_2$ 和 $W_2 \Rightarrow W_1$，其副条件分别是$(a_1 \Rightarrow (e_1 == e_2 \wedge a_2))$和$(a_2 \Rightarrow (e_2 == e_1 \wedge a_1))$。

3、将等价规则中浓缩节点改成无约束浓缩节点的蕴涵规则

- 若等价规则 $W_1 \Leftrightarrow W_2$ 中，W_1 和 W_2 的浓缩节点分别带 e_1 与 a_1 和 e_2 与 a_2 且 $e_1 < e_2$，则有蕴涵规则 $W'_1 \Rightarrow W'_2$，W'_1 与 W'_2 和 W_1 与 W_2 的区别在于浓缩节点改为无约束浓缩节点。
- 若在等价规则 $W_1 \Leftrightarrow W_2$ 中，W_2 有带 e 与 a 的浓缩节点，W_1 无浓缩节点，则有蕴涵规则 $W_1 \Rightarrow W'_2$，其中 W'_2 和 W_2 的区别在于 W_2 中的浓缩节点改为无约束浓缩节点。

2.4 形状图的语义

对于像 C 这样有动态存储分配的编程语言，形状图节点（null 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，而一般的形状图则是某个机器状态集的图形表示。

先定义节点和边的含义。在有栈和堆的机器上，节点所代表的程序变量及指称如下。

（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）结构节点代表由 malloc 调用动态生成的结构体变量，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，其存储单元的个数和地址与该节点的出边条数和标记一样。

为集中于关注的重点，忽略非指针的变量及它们在机器上的存储单元。另外把无约束浓缩节点看成带 n 和 $n \geq 0$ 的浓缩节点，并入浓缩节点一起讨论。

（3）带 e 和 a 的浓缩节点代表 n 个结构体变量，指称 n 个分离的堆块，若 e 的值是 n。

（4）null 节点和悬空节点都不代表任何程序元素，当然也不指称机器上任何东西。

（5）谓词节点代表若干个动态生成的结构体变量，指称堆上若干个分离的堆块。基于 2.3 节的谓词展开规则，根据下面给出的有向边含义，可以明确这些堆块之间的联系。

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）。

由此可知，形状图中 n 条边集中指向一个还是分散指向 n 个 null 节点（悬空节点）不影响形状图的含义。为简单起见，本文是按分散指向来介绍规则的。

（3）边指向带 e 和 a 的浓缩节点。若 $e > 0$，则相应指针的值是浓缩节点展开后第 1 个结构节点（以该边的指向为序）所指称的堆块的地址；若 $e = 0$，则相应指针的含义在删除这个节点后的形状图上确定。

（4）若边指向谓词节点，则边的含义在展开谓词节点后的形状图上确定。

再定义形状图的语义。对于非 NULL 且非悬空的指针，认为它的值就是它所指向堆块的抽象地址，对应地，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。机器的抽象状态（以下简称机器状态）就可由两个函数：s_d: *DecVar* → *AbsValue*∪{N,Δ}和 s_f: *AbsValue* × *FieldVar* → *AbsValue*∪{N,Δ}构成，其中 s_d 的定义域是声明指针集，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，N和Δ是两个特殊的抽象值。下面用 *s* 或⟨s_d, s_f⟩表示机器状态。

在此简化下，基于先前节点和边的含义，一个没有浓缩节点和谓词节点的形状图 *G* 则是某个机器状态的图形表示，而一般的形状图则是某个机器状态集的图形表示。

定义 4 形状图 *G* 所代表的机器状态集Σ□*G*□由下面几条规则定义。

（1）若 *G* 无浓缩节点和谓词节点，则Σ□*G*□中只有一个状态。*G* 直接体现该状态，其中所有声明节点及其出边的指向构成函数 s_d，各结构节点及其出边的指向构成函数 s_f。

（2）*G* 有带 e 和 a 的浓缩节点。若 a 蕴涵 e 可等于 k 个值，这 k 种情况下浓缩节点完全展开后的形状图分别是 G_1,…, G_k，则Σ□*G*□ = Σ□G_1□ ∪ … ∪ Σ□G_k□；若 a 蕴涵 e 可等于 0, 1,…，浓缩节点完全展开成 n 个结构节点（n ≥ 0）的形状图是 G_n，则Σ□*G*□ = Σ□G_0□ ∪ Σ□G_1□ ∪ …。

（3）若 *G* 有谓词节点，且谓词节点展开后的形状图是 *G′*或 $G_1 \vee G_2$，则Σ□*G*□ = Σ□*G′*□或Σ□*G*□ = Σ□G_1□ ∪ Σ□G_2□。

很容易进一步定义形状图 $G_1 \vee G_2 \vee \ldots \vee G_n$ 的语义。

形状图上的路径描述采用和程序中访问路径同样的语法形式，以利于讨论两者之间的对应。显然，只需要考虑从声明节点开始到达某个节点的路径，分成下面两种情况。

（1）路径的完全表示。若路径最多到达浓缩节点，则由依次列出路径各边上的标记来表示该路径。若边上的标记依次为 p, l, r，则写成 p->l->r。

（2）路径的浓缩表示。若路径包括带 e 和 a 的浓缩节点的出边 l，则路径需使用上角标来表示重复次数。例如在图 2(b)中，有 head(->next)^m 和 head(->next)^m->next 等路径。

显然，形状图上一条路径的最后一条边上的标记所代表的程序指针，与程序中表示这个指针的访问路径一致，以下统称它们为访问路径。

形状图是程序中指针有效性断言和指针相等断言等的图形表示。

定义 5 形状图 *G* 所表示的断言集A□*G*□由下面这些有关指针的断言组成：

（1）指向结构节点的指针都是有效指针，指向 null 节点或悬空节点的指针都是无效指针。

（2）指向同一个结构节点或谓词节点的指针相等，指向浓缩节点展开后的同一个结构节点的指针相等，例如图 2(b)表示的断言中有 head(->next)^m->next == ptr；

（3）指向 null 节点（悬空节点）的指针都等于 NULL（是悬空指针）；

（4）指向谓词节点的指针都满足相应的谓词。

基于这个断言集，可以知道哪些指针不相等，可以推导哪些访问路径互为别名。

定理 1 对任何形状图 *G*，Σ□*G*□的状态都满足A□*G*□的所有断言，写成∀*s*:Σ□*G*□. *s*□A□*G*□。并且对任何 *s′*∉Σ□*G*□，若 dom(*s′*$_d$) = dom(s_d)（*s*∈Σ□*G*□）且二元函数 *s′*$_f$: *AbsValue* × *FieldVar* → *AbsValue*∪{N,Δ}的 *AbsValue* 和 *FieldVar* 的大小和 s_f的一样，则 *s′*□A□*G*□。

证明概述 *G* 上访问路径集用 *AccessPath* 表示，Σ□*G*□用 *State* 表示。则求访问路径 u 所代表指针的值的函数 *GetAbsValue* : *AccessPath* × *State* → *AbsValue*∪{N,Δ}的定义如下：

GetAbsValue□u□⟨s_d, s_f⟩ = s_d(u)， 若 u 是声明变量

GetAbsValue□u□⟨s_d, s_f⟩ = s_f(*GetAbsValue*□v□⟨s_d, s_f⟩, next)， 若 u 是 v->next 的形式

显然，指向一个节点的 u∈*AccessPath* 的值就是该节点所代表堆块的抽象地址或者属于{N,Δ}，指向同一个节点的 u, v∈*AccessPath* 的抽象值相同。因此，对不含浓缩节点和谓词节点的 G，定理 1 成立。仿照定义 4 的步骤进行归纳证明，可将定理 1 扩大到一般的 G。

定理 2 若使用等价变换规则可得 $G \Leftrightarrow G'$或者 $G \Leftrightarrow G_1 \vee G_2$，那么$\Sigma(G)=\Sigma(G')$或者$\Sigma(G)=\Sigma(G_1)\cup\Sigma(G_2)$，并且对任何 s:$\Sigma(G)$，s□A□G□当且仅当 s□A□G'□或者 s□A□G□当且仅当 s□A□G_1□或 s□A□G_2□。

若使用蕴涵变换规则，则把状态集的相等改成子集包含，把当且仅当改成蕴含。

证明概要 对每条形状图变换规则，依据定义 4 中对应的状态集构造规则来证明。例如，若由图 4 的规则(5)得到 $G \Leftrightarrow G_1 \vee G_2$，则根据定义 4 的规则（3）可以知道，$\Sigma(G)$可以分成两个子集$\Sigma(G_1) \cup \Sigma(G_2)$，其中的状态分别满足A□$G_1$□和A□$G_2$□。

由定理 2，当把形状图看成断言集合时，形状图的变换规则就是断言演算的等价规则或蕴涵规则。并且这些规则对所考虑的语义模型有效（valid）。

由此，形状图可以作为程序断言，形状图之间的符号∧和∨分别是逻辑合取和析取连接词。通常使用析取范式 $G_1 \vee G_2 \vee \ldots \vee G_k$ 来表示数据结构形状的不同情况。形状图之间的符号⇔和⇒分别可以看成逻辑等价和蕴涵连接词。

2.5 形状图和符号断言之间的联系

形状图是关于易变数据结构中的指针的断言，它们可以和符号断言一起进行演算。例如，条件语句的推理规则会使语句中布尔表达式 u==NULL 和 u!=v 等（u 和 v 都是指针型访问路径）和形状图形成合取，对它们需要有特殊的演算规则。这类断言若和形状图无矛盾则被忽略；有矛盾则否定形状图，连形状图一起的整个断言为假。图 8(1)是一条这样的规则。

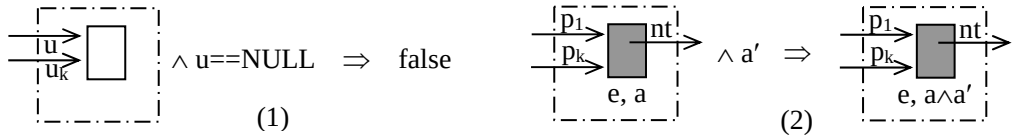


图 8 形状图和符号断言合取的部分规则

同样，程序中出现约束 e 取值的断言 a′时，需要把它加到相应浓缩节点的断言上。图 8(2)是一条这样的规则。若 a∧a′为假则整个断言为假。

在图 8(1)的规则中，放在边下方的 u 和 u_k 是访问路径而不是边的标记（边的标记在边的上方），它们对使用该规则的要求与标记情况略有不同。把该规则中的窗口称为 W，若对 $G[W]$使用该规则时，$G[W]$要有到达 W 中节点的访问路径 u。u_k 同样表示访问路径。第 3 节的规则中出现在边下面的 u 和 v，含义和这里的一样。

这些规则对语义模型的有效性也不难证明。

2.6 形状图和访问路径集合之间的联系

在设计形状图逻辑之前，我们曾经设计过一种指针逻辑[21]。指针逻辑是一种符号化了的形状图逻辑。在这里先比较形状图和指针逻辑的访问路径集合。

我们用循环双向链表插入函数中的一段代码以及程序点的形状图和访问路径集合来比较这两种逻辑。在图 9 中，对于 p 所指向的循环双向链表，在找到插入位置（q 所指向节点的左邻）后，该段代码实施把 s 所指向节点插入链表的操作。n 是链表长度（n ≥ 0），i 是 q 所指向节点之前的节点数。链表节点的类型是 typedef struct node{Node* l; Node* r;…}Node。图 9 仅给出插在表中某位置（0 < i < n）时，某些程序点的形状图和指针集合。

形状图上的节点及有向边和访问路径集合的对应如下：

（1）形状图上的每个结构节点对应到一个访问路径集合，并且指向该节点的有向边和该集合中的访问路径一一对应。在图 9 中，带上角标的访问路径，如 $p(->r)^2->l$，表示 p->r->r->l。

（2）浓缩节点对应到若干个（由浓缩节点下的 e 和 a 确定）访问路径集合，它们可以用全称量词断言来概括。不过，对单向链表而言，当有多个指针指向被浓缩节点序列中的左边缘节点时，它对应的访问路径集合不能包括到全称量词断言中。

（3）所有 NULL 节点（悬空节点）对应到一个带下角标 N 的访问路径集合，所有指向 NULL 节点（悬空节点）的有向边和该集合中的访问路径一一对应。

（4）没有与谓词节点对应的访问路径集合，但形状谓词对应到一个用访问路径集合来定义的谓词。例如图 3 中的二叉树定义对应到下面的定义：

$$tree(s) \triangleq \{s\}_N \vee (\{s\} \wedge tree(s\text{->}l) \wedge tree(s\text{->}r))$$

（5）没有与声明节点对应的访问路径集合，因为没有指向声明节点的指针。

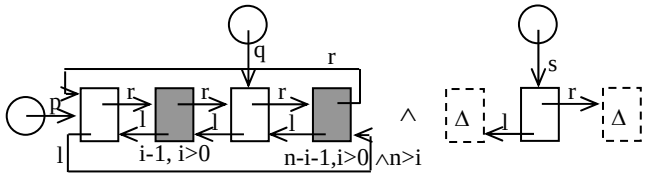
显然，指向同一个节点（悬空节点除外）的指针都相等对应到同一个集合（带下角标 D 的集合除外）中的访问路径所代表的指针都相等。访问路径集合也就是相等指针的集合。当有多条访问路径可表示一个指针时，指针逻辑取其中一条来代表该指针。这样，访问路径集合可以看成指针相等断言集合，因而也可以像形状图那样，对访问路径集合进行合取或析取。

2.3 节的形状图变换规则在指针逻辑的断言语言中也都有对应的断言演算规则。例如，在被浓缩的边缘节点也能包括在带全称量词的断言中时，浓缩节点的展开和折叠规则对应到谓词逻辑中全称量词断言所囊括区间的展开和折叠规则。在指针逻辑中，对应图 8(1)的规则如下：

$$\Sigma_1 \wedge \ldots \wedge \Sigma_{n-1} \wedge \Sigma^u \wedge N \wedge \Delta \wedge u == NULL \Rightarrow false$$

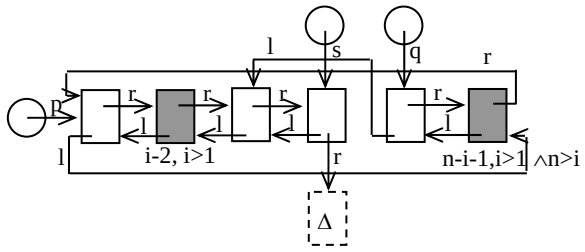
其中各个Σ是对应到各结构节点的访问路径集合，N和Δ分别是 NULL 指针集合和悬空指针集合。Σ^u 表示 u 所在的指针集合，它相当于图 8(1)规则的窗口。

形状图　　指针集合



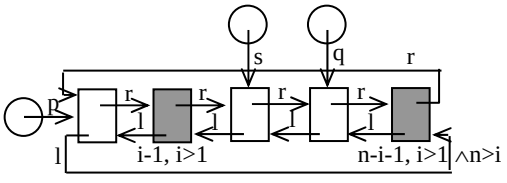
$\{p, p(->r)^n, p->r->l\} \wedge \forall k:1..i-1.\{p(->r)^k, p(->r)^{k+1}->l\} \wedge \{q, p(->r)^i, p(->r)^{i+1}->l\} \wedge \forall k:i+1..n-1.\{p(->r)^k, p(->r)^{k+1}->l\} \wedge \{s\} \wedge \{s->r, s->l\}_D \wedge i > 0 \wedge n > i$

s->l = q->l; q->l->r = s;　/*下面仅给出 1< i（略去 i = 1）时，形状图及其对应的访问路径集合*/



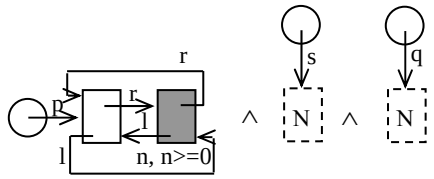
$\{p, q(->r)^{n-i}, p->r->l\} \wedge (\forall k:1..i-2.\{p(->r)^k, p(->r)^{k+1}->l\} \wedge \{s->l, p(->r)^{i-1}, q->l\} \wedge \{p(->r)^i, s\} \wedge \{s->r\}_D \wedge \{q, q->r->l\} \wedge \forall k:i+1..n-1.\{q(->r)^{k-i}, q(->r)^{k-i+1}->l\} \wedge i > 1 \wedge n > i$

s->r = q; q->l = s;



$\{p, q(->r)^{n-i}, p->r->l\} \wedge \forall k:1..i-1.\{p(->r)^k, p(->r)^{k+1}->l\} \wedge \{p(->r)^i, q->l, s\} \wedge \{s->r, q, q->r->l\} \wedge \forall k:i+1..n-1.\{q(->r)^{k-i}, q(->r)^{k-i+1}->l\} \wedge i >1 \wedge n > i$

s = NULL; q =NULL;　/*下面是 0 ≤ i ≤ n 的各种情况都能满足的形状图和对应的访问路径集合*/



$\{p, p(->r)^{n+1}, p->r->l\} \wedge \forall k:1..n.\{p(->r)^k, p(->r)^{k+1}->l\} \wedge \{q, s\}_N \wedge n >= 0$

图 9 循环双向链表插入函数的部分代码和断言

3 程序逻辑

一般而言，各程序点除了形状图作为断言外，还会有表达节点数据性质的符号断言 Q。在程序验证时，程序点完整的断言保持析取范式 $G_1 \wedge Q_1 \vee G_2 \wedge Q_2 \vee \ldots \vee G_n \wedge Q_n$ 的形式。在设计程序逻辑的推理规则时，只需要用析取范式的一种情况来讨论，故本节认为程序规范的形式是$\{G \wedge Q\}C\{G' \wedge Q'\}$。

若 C 是指针操作语句（指针赋值、分配空间和释放空间等以指针为操作对象的语句），G 到 G'的变化不受 Q 的影响，而 Q 到 Q'可能是因访问路径的别名替换而引起语法上而非语义上的变化。若 C 不是指针操作语句，则 G'和 G 最多在浓缩或谓词节点的 e 和 a 上有区别。

3.1 节主要先考虑指针操作语句的推理规则，且只涉及形状图变化，Hoare 三元组暂且是$\{G\}C\{G'\}$形式，它正好用于形状分析阶段。G 和 Q 都包括在内的完整推理规则在 3.2 节。

3.1 只关注形状图变化的推理规则

在使用本小节的规则时，要求

（1）指针相等符号断言因化简而消失。

（2）对语句 C 和其之前程序点的 G，C 中出现的访问路径在 G 中一定存在，且最多到达 G 浓缩节点而不穿越浓缩节点。否则需要展开 G 的浓缩节点或报错。

1、指针赋值语句 u = v

（1）u 指向 null 或悬空节点，v 指向结构、浓缩（限于单链表的情况）或谓词节点。

从 G 到 G'的变化就是让 u 也指向 v 原来指向的节点。图 10 代表其中一条规则，花括号中的不是形状图，而只是其中被所关注的窗口。若这 4 个窗口依次用 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 $\{G[W_{21}, W_{22}]\}$　　（$G[W_{11}, W_{12}]$上的窗口 W_{11} 和 W_{12} 无重迭）

因此，严格说图 10 只是规则的窗口而不是规则，简单起见仍称为规则。

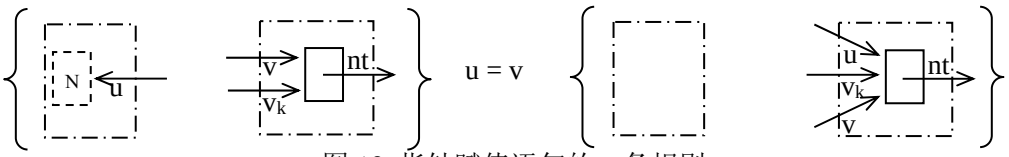
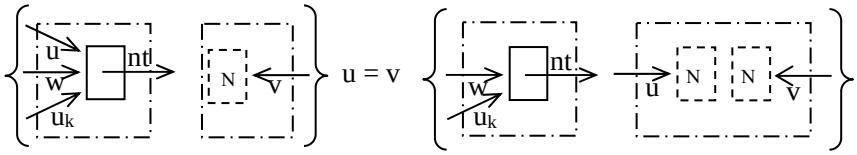


图 10 指针赋值语句的一条规则

（2）u 指向结构、浓缩（限于单链表的情况）或谓词节点，v 指向 null 或悬空节点（包括 v 是常量 NULL 的情况）。

图 11 是其中的一条规则。若赋值前只有 u 指向结构节点，则报告内存泄漏。结构节点有 2 条出边的规则类似。用简单的例子来解释该规则的副条件是必须的。若仅有声明指针 s 指向只有一个结构节点的循环单向链表，则其形状图类似图 3(2)的右部，其中浓缩节点需改成结构节点。若将该规则用于语句 s = NULL 可知，若无该副条件，有可能漏报内存泄漏。



（访问路径 w 和 u_k 中有不经过 u 所代表的边的访问路径）

图 11 指针赋值语句的另一条规则

图 12 可用来解释图 11 规则的副条件。假如赋值语句为 p = NULL，若无副条件，则虽有两个指针 p 和 p->next->next 指向同一个节点，仍会引起这两个节点的内存泄漏。副条件禁止了该赋值，因为 p->next->next 以 p 为前缀。

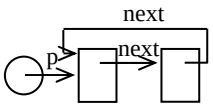


图 12 一个简单的单向循环链表

（3）u 和 v 指向不同的结构、浓缩或谓词节点

把该语句当作语句序列 dummy = v; u = NULL; u = dummy; dummy = NULL，逐步修改 *G*。其中 dummy 是各函数都有的虚拟局部指针变量，它与其他局部变量都不同名。采用该语句序列是为了避免为这种指针赋值设计一条复杂的专用规则。例如对于图 12，赋值 p = p->next 不会引起内存泄漏。若设计专用规则来保证不会误报或漏报内存泄漏，则规则会很复杂。

2、**分配空间语句 u = malloc(t)**

（1）u 指向 null 或悬空节点

在 *G* 中增加一个结构节点，让 u 改成指向该节点。并且根据类型 t 中域指针的个数，给该节点添加指向悬空节点的出边，其标记分别为这些域指针的名字，就得到 *G*′。图 13 的规则代表其中一种情况，假定结构体类型 t 只有一个域指针 f。

（2）u 指向结构、浓缩或谓词节点

把该语句当作语句序列 u = NULL; u = malloc(t)，逐步修改 *G*。

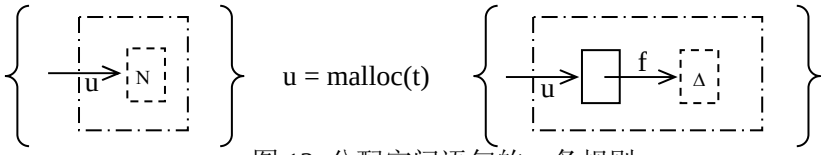


图 13 分配空间语句的一条规则

3、**释放空间语句 free(u)**

根据 u 的类型可知其指向的结构体的所有域指针的访问路径 u->r_1,…,u->r_n。把该语句当成语句序列 u->r_1=NULL; …; u->r_n=NULL; free(u)来逐步修改 *G*。末尾的 free(u)引起 *G* 的修改是：在 *G* 上删除 u 指向的节点、该节点所有出边及它们指向的节点，并把 u 以及与 u 指向同一个节点的指针都改成指向悬空节点。

4、**与函数构造有关的语句**

仅介绍只有一个指针形参 arg，结果类型也是指针的情况。假定函数体中有虚拟指针变量 res 和 dummy，前者用以代表函数结果，后者用于简化赋值规则。

（1）局部指针声明语句

为所有局部指针 q_1, …, q_k 以及 res 和 dummy 分别构造声明节点指向悬空节点的形状子图，合取到函数前形状图上。

（2）函数返回语句 return exp

把它变换成 res = exp; q_1 = NULL; … q_k = NULL; arg = NULL; return 来逐步修改 *G*。末尾的 return 语句将 *G* 中 q_1, …, q_k, arg 和 dummy 各自所在形状子图删除，形成函数后形状图。注意，若其中浓缩节点的 e 和 a 中出现形参或局部变量，则需要将它改成无约束浓缩节点。

（3）函数调用语句 ret = f(act)

仅考虑 ret 等于 NULL 且 ret 与 act 是不同指针的规则，见图 14，假定函数体是 S。注意，在函数前形状图上增加代表实参的声明节点 arg′，它和形参 arg 指向同一个节点。其作用是，若它们指向的 *G* 被函数 f 修改，即便 f 不返回 arg，修改也会反馈到调用语句之后的 *G* 上。

若程序中对 f 的调用都是 act = f(act)的形式，则不必增加声明节点 arg′。

在图 14 中，实线大方框称为**黑箱**，其含义和窗口的含义相反，黑箱表示只关心暴露在黑箱边缘的部分，在黑箱里面的部分不发生变化。该规则表示如何根据 f 函数后形状图 G_{callee} 来对函数调用语句之前的 G_{caller} 进行修改，以得到它之后的 *G*：

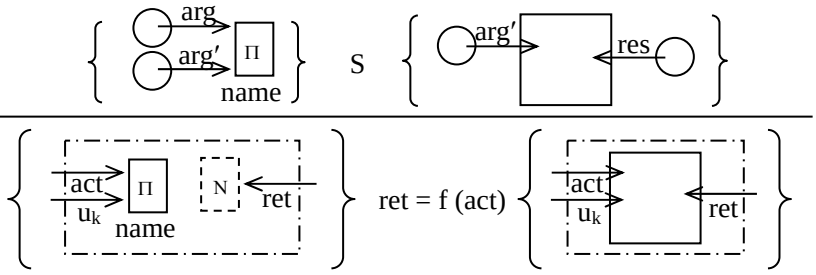


图 14 函数调用规则的一种情况

用 G_{callee} 的黑箱覆盖 G_{caller} 窗口中的谓词节点和 null 节点，用 act 和 u_k 代替黑箱接口的 arg′，用 ret 代替黑箱接口上的 res，就得到调用语句之后的 G。

5、整型赋值语句

若整型赋值语句修改浓缩节点或谓词节点的 e 和 a 中的变量，则 e 和 a 要根据赋值公理做相应调整。

6、复合、条件和循环语句的规则

复合、条件和循环语句的规则以及推论规则的形式和 Hoare 逻辑相应规则的一致。

7、其他规则

（1）分情况规则

若语句的前断言是 $G_1 \vee G_2$，则用下面的分情况规则可得到后断言。

$$\frac{\{G_1\}C\{G_1' \vee ... \vee G_m'\} \quad \{G_2\}C\{G_1'' \vee ... \vee G_n''\}}{\{G_1 \vee G_2\}C\{G_1' \vee ... \vee G_m' \vee G_1'' \vee ... \vee G_n''\}} m, n \geq 1$$

（2）框架规则（frame rule）

先前设计的规则都体现局部推理的思想，即指针操作语句仅使得形状图的某个窗口发生变化，而该窗口的上下文不变。下面的规则就是体现该思想的框架规则。

$$\frac{\{W_1\}C\{W_2\}}{\{G[W_1]\}C\{G[W_2]\}}$$

3.2 同时关注 *G* 和 *Q* 的推理规则

本小节将形状图逻辑补充完整，包括非指针操作语句的推理规则和将指针操作语句的推理规则补充完整，它们用于程序验证阶段，从语句 C 的前断言 $G \wedge Q$ 产生后断言 $G' \wedge Q'$ 中的 Q'。这时要保证：

（1）Q 和 C 中指针类型的访问路径都合法，即在 G 上都存在。且 Q 和 C 中无别名。

（2）Q 中没有和 G 重复或矛盾的诸如 p == NULL 和 p == q 的指针断言。

（3）Q 中出现谓词应用断言和量词断言时要关注的事项见第 5.1 节。

若有别名，则先用 G 来消除别名（把互为别名的访问路径改成都用其中某条），然后再用推理规则。定义 *eliminate_aliases* 函数为$(C', Q') = eliminate_aliases(G, C, Q)$，它根据 G 消除 C 和 Q 中的别名，得到 C'和 Q'。

把所有的规则限定为无别名才能使用，并增加下面的消除别名推理规则，就可以对含访问路径别名的程序进行推理。

$$\frac{\{G \wedge Q''\}C'\{G' \wedge Q'\}}{\{G \wedge Q\}C\{G' \wedge Q'\}} \quad (C', Q'') = eliminate\_liases(G, C, Q)$$

1、非指针操作语句

对于非指针操作的赋值语句，若 x 不涉及 G 中节点的 e 和 a，则直接使用正向的赋值公理，它不影响 G。

$\{G \wedge Q\}$ x = E $\{G \wedge Q[x'/x] \wedge x == E[x'/x]\}$（x′是一个与已有变量都不同名的新变量）

其他非指针操作语句，如不修改任何指针的函数调用，可直接使用 Hoare 逻辑的规则。

2、指针操作语句

对指针操作语句，完整的规则基于 3.1 节的规则来调整 Q。

（1）指针赋值语句 u = v

推理规则是

$$\frac{\{G\}\ \mathrm{u}\ =\ \mathrm{v}\ \{G'\}}{\{G \wedge Q\}\ \mathrm{u}\ =\ \mathrm{v}\ \{G' \wedge Q[\mathrm{u}'/\mathrm{u}]\}}$$ （u'是与u相等的访问路径）

其前提由 3.1 节的规则在分析阶段得到。

（2）对指针赋值的其他语句

对于分配空间语句 u = malloc(t)和函数调用语句 ret = f (act)，有关 Q 的处理同上面赋值语句的规则一样。

（3）释放空间语句 free(u)

释放 u 指向的堆块后，Q 中含 u 的基本断言不应再存在，因此规则如下：

$$\frac{\{G\}\ \mathrm{free}(\mathrm{u})\ \{G'\}}{\{G \wedge Q\}\ \mathrm{free}(\mathrm{u})\{G' \wedge Q\}}$$ （Q中不含u）

3.3 形状图逻辑的可靠性

3.1 节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。

定理 3 若用 3.1 节的推理规则得到$\{G\}\ C\ \{G'\}$，则对任何 $s \in \Sigma\square G\square$，若语句 C 的操作语义是$\langle C, s\rangle \to s'$，则 $s' \in \Sigma\square G'\square$。

以指针赋值语句 u = v 为例来说明，其操作语义是

$\langle \mathrm{u} = \mathrm{v}, \langle s_d, s_f\rangle\rangle \to \langle s_d[\mathrm{u} \mapsto \mathit{GetAbsValue}\square \mathrm{v}\square\langle s_d, s_f\rangle], s_f\rangle$ 若 u 是声明指针

$\langle \mathrm{u} = \mathrm{v}, \langle s_d, s_f\rangle\rangle \to \langle s_d, s_f[\langle \mathrm{x}, \mathrm{next}\rangle \mapsto \mathit{GetAbsValue}\square \mathrm{v}\square\langle s_d, s_f\rangle]\rangle$

若 u 是 w->next 且 $\mathit{GetAbsValue}\square \mathrm{w}\square\langle s_d, s_f\rangle = \mathrm{x}$

该操作语义表示，根据 u 的两种情况，仅修改状态 s 的一个栈单元或某堆块的一个单元，s 的其他部分不变。图 10 和 11 的规则用窗口表示形状图的其他部分不受影响，仅修改一条边的指向，与操作语义一致。

其他推理规则中形状图的变化也同样表达了相应语句的操作语义。

定理 4 3.1 节那些指针操作语句的推理规则对操作语义可靠。

先考虑公理形式的任意推理规则$\{W_1\}\ C\ \{W_2\}$。需要证明：对能够使形状图 $G[W_1]$和 $G[W_2]$正好包括程序中所有声明指针的任意上下文 $G[]$，对任意 $s_1 \square \mathrm{A}\square G[W_1]\square$的状态 s_1，若$\langle C, s_1\rangle \to s_2$，则 $s_2 \square \mathrm{A}\square G[W_2]\square$。

从推理规则$\{W_1\}\ C\ \{W_2\}$可知，$\{G[W_1]\}\ C\ \{G[W_2]\}$。从定理 1 可知，若 $s_1 \square \mathrm{A}\square G[W_1]\square$，则 $s_1 \in \Sigma\square G[W_1]\square$。从定理 3 可知，$s_2 \in \Sigma\square G[W_2]\square$。再从定理 1 可知，$s_2 \square \mathrm{A}\square G[W_2]\square$。

对于有前提的推理规则，也不难证明它们对操作语义可靠。

在 Hoare 逻辑对操作语义可靠的基础上，完成形状图逻辑对操作语义的可靠性证明还需要证明下面几点，在此略去。

（1）对于 PointerC，Hoare 逻辑的赋值公理在没有别名时的使用是可靠的。

（2）*eliminate_aliases* 函数得到的 C'和 C 有同样的语义，Q'和 Q 等价。

（3）消除别名的推理规则是可靠的。

3.4 形状图逻辑和指针逻辑的推理规则之间的联系

形状图逻辑和指针逻辑的推理规则之间也有很好地对应。例如，对于图 13 的推理规则，指针逻辑中对应的推理规则是

$$\{\Sigma_1 \wedge \ldots \wedge \Sigma_n \wedge \mathrm{N}^{\mathrm{p}} \wedge \Delta\}\ \mathrm{p} = \mathrm{malloc(t)}\ \{\ \Sigma_1 \wedge \ldots \wedge \Sigma_n \wedge \{\mathrm{p}\} \wedge (\mathrm{N}^{\mathrm{p}} - \mathrm{p}) \wedge (\Delta + \{\mathrm{p\text{->}f}\})\ \}$$

对于图 11 的推理规则，指针逻辑中对应的推理规则是

$$\{\Sigma_1 \wedge \ldots \wedge \Sigma_{n-1} \wedge \Sigma^{\mathrm{u}} \wedge \mathrm{N}^{\mathrm{v}} \wedge \Delta\}\ \mathrm{u} = \mathrm{v}\ \{\Sigma_1/\mathrm{u} \wedge \ldots \wedge \Sigma_{n-1}/\mathrm{u} \wedge (\Sigma^{\mathrm{u}}/\mathrm{u} - \mathrm{u}) \wedge (\mathrm{N}^{\mathrm{v}}/\mathrm{u} + \mathrm{u}) \wedge \Delta/\mathrm{u}\}$$

（若赋值前，$\mathrm{leak}(\Sigma^{\mathrm{u}}, \mathrm{u})$为假）

其中前缀替换Σ/p 是指（严格定义在[21]），对访问路径集合Σ中以 p 或 p 的别名为前缀的每条访问路径 q，都用它的一个别名 q′来代替，q′不以 p 或 p 的别名为前缀；Σ中其余访问路

径维持不变。

前缀替换是一个非常复杂的操作，同样，判断对 u 赋值不会引起内存泄漏的 leak(Σ^u, u) 谓词也非常复杂。因此，指针逻辑的规则看似简单，实际上访问路径的别名判断和访问路径的前缀替换等基本操作非常复杂，而采用形状图时不出现访问路径，因而没有这些问题。例如，图 9 中第 1 行代码的两条赋值语句引起 p 所指向的循环双向链表被断开，导致部分指针由原来用 p 开头的访问路径作为名字改成用 q 开头的访问路径作为名字。而从这两个赋值前的形状图经过浓缩节点的展开和调整指针的指向就可得到赋值后的形状图，显然简单得多。

对于图 9 第 1 行的形状图和访问路径集合，就判断 p 指向的是否为循环双向链表而言，从形状图比从访问路径集合更容易得出结论。在具体实现中，在很多情况下，甚至只有把访问路径集合转换为一种图形表示才能方便地判断它的性质和对它进行修改。

采用完整的访问路径表示指针的方式使得指针逻辑难以有框架规则。

鉴于这些原因，我们在设计了指针逻辑之后，进一步设计形状图逻辑。

4、形状系统

设计形状系统的目的是提高合法程序的门槛，以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，降低形状分析和程序验证的难度。

4.1 形状系统的设计

形状系统包括：所允许的各种形状及其定义、形状推断规则和形状检查规则。目前所允许的形状就是图 3（包括未列出的定义式）所定义的各种形状。

形状推断就是推断声明指针 p 所在形状子图是什么形状。其步骤及规则概述如下：

（1）从待推断的形状图上分离出 p 所在形状子图 G_1。

（2）用语句 q = NULL（q 是声明变量）的规则来减少 G_1 上的声明指针，得到只剩一个声明指针（可以不是 p）的 G_2。

（3）用形状图的等价变换规则将 G_2 尽可能进行折叠化简，得到形状图 G_3。

（4）若 G_3 是与 p 的声明关联的形状的最简形状图，则 G_1 就是相应形状的形状图。

若步骤（2）仅限于删除和 p 指向同一个节点的声明指针并且保留 p 时，称为**严格形状推断**，否则称为**粗略形状推断**。即严格推断不接受还有额外指针指向其他节点的形状子图。

形状检查利用形状推断来判断声明指针所在形状子图是否与其声明所表达的指向形状一样。形状检查规则规定在什么程序点对哪些指针进行形状检查，采用严格形状推断、粗略形状推断还是更宽松的检查。这是较难决策的事情，因为必须容忍插入或删除操作导致的形状被暂时破坏，即必须允许在某些程序点，声明指针暂时未指向所声明的形状。目前确定的检查点有下面几种。

（1）函数的调用点和返回点

函数调用的指针型实参所在形状图（经严格推断）的形状，与其声明形状必须相同。这就避免了可能还有其它指针指向形状图内部某个节点给被调用函数的形状分析带来的困难。

在函数返回点关注两种可能有的指针：实参和返回值。当它们的类型相同并且指向同一个形状子图时，它们有可能指在不同节点上，见第 5 节例 2。因此在函数返回点进行粗略的形状检查。若有必要，程序员可在调用语句之后增加严格形状检查。还有一种特殊情况不作为形状错误：实参所指向节点被释放，实参成为悬空指针。

（2）循环语句的入口点

凡是在循环体中被修改的声明指针，若在循环体的结尾没有被修改成 null 或悬空指针，则在循环语句的入口，它们所指向形状图（经粗略推断的）形状与它们的声明形状必须相同。在此用粗略推断是因为循环语句在操作数据结构的过程中，可能有多个声明指针指在不同节

点上，例如图 2(a)的循环不变状态图。

（3）循环体的结束点

很多情况下可以让形状检查出现在循环体的结束点，但并非所有程序都保证每次执行到循环体结束点时都保持或恢复形状。典型例子是第 5 节例 2 的双向链表的倒置程序。在用循环语句逐步倒置每个节点的两个指针时，在循环体的结束点，已经倒置的部分和尚未倒置的部分看上去都像双向链表，但整体上这两部分并不构成双向链表。只有等所有节点都倒置，也就是循环结束时，结果才是双向链表。还有这样的例子，在循环结束时离所声明形状还有一步之遥，需等随后的个别指针赋值才能恢复形状。

在循环体的结束点进行形状检查，便于及时发现形状图偏离所声明形状的情况。这有利于保证循环不变形状图推断过程的终止。例如，若某个循环将一个双向链表各节点的域指针 l 都赋值为 NULL，而域指针 r 都不变。这相当于废弃 r，用 l 构成单向链表。这时不存在对已经遍历过的节点进行折叠的规则，若不拒绝该程序，推断过程将不能终止。

由于上面两段所讲的原因，在循环体的结束点采取较为宽松的检查：只要能够对遍历过的节点进行折叠，则检查通过。

形状系统和类型系统看上去相似，其实他们有本质的区别。首先，普通的类型系统给出对静态程序文本的上下文有关的约束，不涉及语言的操作语义；而形状系统给出对程序动态地构造的数据结构的形状限制，它依赖于语言的操作语义。其次，类型系统的定型规则一般是结构化的，即根据语言构造的各子构造的类型来确定该语言构造的类型，而形状系统的形状推断和形状检查是根据各节点的连接方式来推断所构成的形状以及它是否被认可。

形状系统也不同于 shape types[34]，虽然两者目的相同。该文 in terms of context-free graph grammars 来定义形状，而本文直接用图而不是图的符号表示来定义形状。该文用抽象的 graph transformers 来描述数据结构的插入和删除等操作，并给出 an algorithm for the static shape checking of graph transformers，而本文的形状检查直接施加于操作数据结构的 PointerC 代码。

4.2 形状系统给程序验证带来的好处

1、免去程序员提供函数前后形状图，并保证有良好的函数接口。

函数调用点的形状限制使得函数的接口简单，使得形状分析能够推断函数后形状图。对指针型递归函数，函数后形状图的推断见[9]。

2、免除程序员提供循环不变形状图。

在循环不变形状图的推断过程中，通过形状检查排除背离程序员意图的程序，简化了循环不变形状图的推断，使得自动推断成为可能。循环不变形状图的推断见[9]。

3、帮助程序员排错。

除了上面提到的形状检查程序点外，程序员还可以指定进行形状检查的其他程序点，以帮助他们发现程序错误。这是静态分析时的检查，不会影响程序运行效率。

5、系统原型

基于形状图逻辑，我们实现了 PointerC 语言的一个程序验证器原型[10]，它能够验证使用图 3（包括未列出的定义式）所定义的各种形状的程序。指针断言的图形表示不会给程序员增添任何麻烦，因为他们不必提供函数前后形状图和循环不变形状图。

5.1 系统概述

验证器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，并在需要形状检查的程序点进行形状推断和形状检查。对循环语句需要遍历多次，以推断循环不变形状图。递归函数后形状图的推断类似于循环不变形状图的推断。

系统目前在形状分析阶段忽略了循环语句和条件语句中控制条件含的整型数据断言，若这些断言决定这些语句之中的指针操作语句是否执行，则形状分析可能出错。例如：

（1）若 p 指向长度为 n 的单向链表，代码段是 i = 0; while(i < m) { p = p->next; i = i +1; } 且 m > n，则系统不能发现对 NULL 指针的 dereference 操作。

（2）若 p 指向长度为 n（$n \geq 0$）的单向链表，代码段是 if(n > 0) {q = p; p = p->next; free(q); }，则系统原型会误报对 NULL 指针 dereference 的错误，因为形状分析仍然把链表当成长度为 0 和非 0 两种情况来分析。

（3）程序员有时会写出 if(B) C_1 else C_2; if(B) C_3 else C_4 这样的 2 个顺序条件语句，并且 C_1 和 C_2 不修改 B 中的变量。它们等效于 if(B) {C_1;C_3;} else{C_2;C_4;}。形状分析因未分析出这种特殊性，仍按常规形成 4 种情况而导致分析可能出错。

在形状分析阶段增加对这些整型数据断言的分析，并在需要时使用自动定理证明器来判断断言的真假，可以克服这些问题。

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 $G \triangleright Q \Rightarrow Q'$，其中 G 是产生验证条件那个程序点的形状图，是 $Q \Rightarrow Q'$的证明环境。

程序断言中可以有谓词应用断言、量词断言和普通断言，其中都可能出现访问路径。这些断言的例子见 5.2 节。有关的合法性要求如下：

- 断言中的指针型访问路径在 G 中一定存在。
- 谓词应用所涉及（影响到，touch）的任何节点上，不得有外来指针指在该节点上，除非它和该应用的某个指针参数相等。外来指针是指不会出现在谓词应用展开图中的指针。
- 对量词断言所涉及（影响到，touch）的节点，所受约束与上一点一致。

验证条件生成阶段除了检查程序员提供断言的合法性外，还要通过适时地展开谓词应用断言和量词断言，以维持程序点断言的合法性。

自动定理证明阶段把 G 上指针相等信息转换为符号断言，称为 P。P 中可能还包含一些其他信息。例如，若 G 中包括浓缩节点，则约束其节点个数表达式 e 的断言 a 也包括在 P 中。若 Q'中有内建函数 length 的应用，则 G 中相应链表的长度断言也在 P 中。内建函数 length(head, next)是指针 head 顺着节点的 next 域一直到 null 节点所经过的结构节点数。本系统将$\neg(P \wedge Q \Rightarrow Q')$交给可满足性模理论求解器 Z3[11]。若$\neg(P \wedge Q \Rightarrow Q')$不可满足，则 $P \wedge Q \Rightarrow Q'$得证。

形状图是分析阶段和验证阶段共用的数据结构，并都被看成指针相等断言集的一种图形表示，这既有利于分析阶段的指针断言推理，也有利于验证阶段的指针信息查询。

该原型可以验证易变数据结构上较为复杂的程序，表 1 给出部分程序在 Windows 7 PC，Intel Core i5-2400 3.1GHz CPU 和 4G 内存上运行时间和空间等的统计数据。其中二叉树的插入和删除函数的统计包括了它们所调用的函数，例如 balance 函数，splay 函数。Splay 树消耗空间和时间突出，这是因为 splay 函数的循环不变形状图是多余 20 个形状图的析取。表 1 中第 2 和第 3 例子只验证了形状，没有验证有序性。

表 1　有关形状图和验证条件的统计数据

数据结构	函数	构建所有形状图耗时 (ms)	所有形状图共占空间(KByte)	循环的平均迭代次数和耗时(次，ms)	谓词个数(个)	性质定理条数(条)	验证条件个数(个)
sorted list	merge	263	81.6	4,　85.5	3	3	5

sorted double linked list	reverse	213	50.0	4.5, 23.5	0	0	0
sorted circular doubly-linked list	insert	92	30.4	3, 16	0	0	0
AVL tree	insert, delete	2708	1118.7	无循环	5	4	32
AA tree	insert	153	116.6	无循环	6	13	8
binary search tree	insert, delete	137	40.4	3, 15	7	10	6
treap	insert, delete	276	95.3	无循环	5	6	18
splay tree	insert	43927	2687.9	9, 43888	5	8	9

5.2 三个例子

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。此外，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。程序员需要提供谓词之间的并与程序有关的归纳性质，给自动定理证明器以提示。因为基于演绎推理的证明器推导不出这类性质。

本小节介绍规模不大，但有助于理解本文的 3 个例子。

例 1 有序单向链表合并函数，返回合并链表的指针。代码和断言见图 15。为描述有序性，程序员提供自定义谓词和性质定理分别如下，其中谓词中的变元 p 和 q 都是节点指针型。

order(p) ≡ p==NULL ∨ p->next==NULL ∨ p->data <= p->next->data ∧ order(p->next)

order_seg(p, q) ≡ p==q ∨ p->data <= p->next->data ∧ order_seg(p->next, q)，和

order_seg(p, q) ∧ q->data <= q->next->data ⇔ order_seg(p, q->next)

order_seg(p, q) ∧ order(q) ⇒ order(p)

图 15 代码中第 1 个循环的循环不变形状图见图 16。在图 16 中，h1′和 h2′是系统根据图 14 的规则添加的代表对应实参的节点，j、j1 和 j2 是系统添加来分别代表循环迭代次数以及条件语句中两个分支语句执行次数的虚拟变量。断言 j == j1 + j2 是 3 者之间满足的关系。程序员在断言中使用的、未在程序中声明的变量称为逻辑变量，例如函数前条件中的 m 和 n。断言 length(h1, next) == m 相当于断言∃m:Z.length(h1, next) == m。

```
typedef struct node {Node*: LIST next; int data;}Node;
Node* merge(Node* h1, Node* h2) {
  assertion order(h1) ∧ length(h1, next) == m ∧ order(h2) ∧ length(h2, next) == n;
  Node* h; Node* p; Node* p1; Node* p2; Node* new;
  h = NULL; new = NULL; p = h; p1 = h1; p2 = h2;
  while( (p1 != NULL) && (p2 != NULL) )
    loop_inrariant order(p1) ∧ order(p2) ∧ order_seg(h, p) ∧ order(p) ∧
                        (length(h, next) + length(p1, next) + length(p2, next) == m + n);{
    if (p1->data < p2->data) {
        new = malloc(Node); new->data = p1->data; new->next = NULL; p1 = p1->next;
    } else {
        new = malloc(Node); new->data = p2->data; new->next = NULL; p2 = p2->next;
    }
    if (p == NULL) {p = new; h = p; } else {p->next = new; p = p->next; }
  }
  if (p1 == NULL ) { p1 = p2; }
  while( p1 != NULL )
    loop_inrariant  order(p1) ∧ order_seg(h, p) ∧ order(p) ∧
                                    (length(h, next) + length(p1, next) == m + n); {
    new = malloc(Node); new->data = p1->data; new->next = NULL;
    if( p == NULL ) {p = new; h= p; } else { p->next = new; p = p->next; }
    p1 = p1->next;
  }
  return h;   assertion order(h) ∧ length(h, next) == m+n;
}
```

图 15 合并有序链表函数的代码和断言

因该循环的不变式中有 length 函数的应用，因此在产生循环入口的验证条件时，系统会从图 16 的形状图上，把验证条件中所涉及的各个声明指针 x 的 length(x, next)等于某个表达式的断言加入验证环境，并把各浓缩节点下的断言也加入证明环境。

已开发的各种程序验证器[12-16]，都未能做到：免除程序员提供有关指针的函数前后条件和循环不变式并且还能比较全面地验证多种易变数据结构的形状和数据性质。

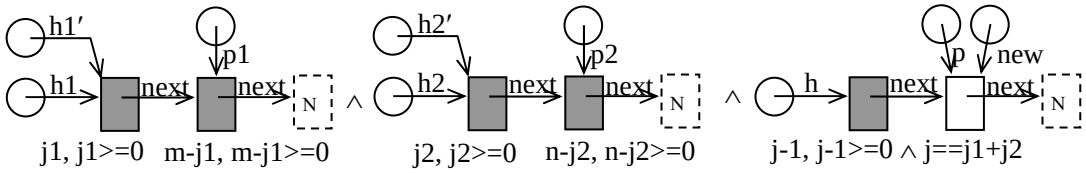


图 16　例 1 第 1 个循环的循环不变形状图

例 2 有序双向链表倒置函数，返回倒置后的链表指针。代码和断言见图 17，函数后条件只表达倒置后的双向链表仍然是有序的。与例 1 不同，本例使用全称量词断言来表示有序性。

由于不允许外来指针指向双向链表的浓缩节点，即便是对双向链表遍历的循环，其循环不变形状图比单向链表的也复杂得多。图 18 是本例第 1 个循环的循环不变形状图。

```
typedef struct listnode {int data; ListNode* :DLIST, nxt; ListNode* :DLIST, pre; }ListNode;
ListNode* invert( ListNode* list) {
assertion length(list, nxt) == n ∧ ∀i:1..n-1.list(->nxt)^(i-1)->data <= list(->nxt)^i->data;
    ListNode* last; ListNode* ptr; ListNode* tmp;
    int k;
    last = list; k = 0;
    if(last != NULL) {
        while(last->nxt != NULL)
        loop_inrariant ∀i:1..k.list(->nxt)^(i-1)->data <= list(->nxt)^i->data ∧
                        ∀i:1..n-k-1.last(->nxt)^(i-1)->data <= last(->nxt)^i->data; {
            last = last->nxt; k = k +1;
        }
        ptr = last; k = 0; tmp = NULL;
        while(ptr != NULL)
        loop_inrariant ∀i:1..n-k-1.list(->nxt)^(i-1)->data <= list(->nxt)^i->data ∧
                        ∀i:1..k.last(->nxt)^(i-1)->data >= last(->nxt)^i->data; {
            tmp = ptr->nxt; ptr->nxt = ptr->pre; ptr->pre = tmp; ptr = ptr->nxt;
            k = k +1;
        }
    }
    return last;
    assertion length(last, nxt) == n ∧ ∀i:1..n-1.last(->nxt)^(i-1)->data >= last(->nxt)^i->data;
}
```

图 17 有序双向链表倒置函数的代码和断言

双向链表在第 2 个循环的迭代过程中暂时被破坏，再加上 list 和 last 分别指向两端结构节点，使得循环不变形状图更为复杂，共有互不蕴涵的 8 种情况，限于篇幅不在此给出。

函数返回时的形状图见图 19。由于返回指针和实参指针分别指在链表的两端，导致不能在此将 3 种情况统一成双向链表的谓词应用。

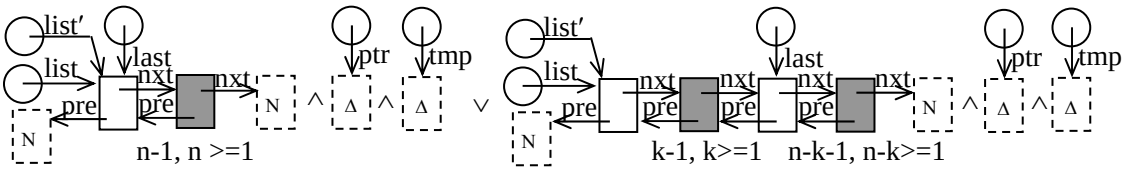


图 18　例 2 第 1 个循环的循环不变形状图

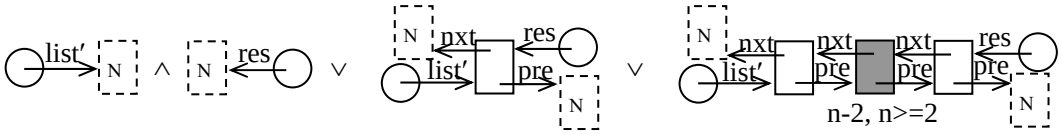


图 19　例 2 返回点的形状图

例 3 二叉排序树插入函数，返回插入节点后的二叉排序树指针。代码和断言见图 20。为描述有序性，使用的自定义谓词和性质定理如下，其中变元 p 是节点指针型，x 和 y 是整型。

order(p) ≜ p==NULL ∨ order(p->l) ∧ order(p->r) ∧ gt(p->data, p->l) ∧ lt(p->data, p->r)

gt(x, p) ≜ p==NULL ∨ x > p->data ∧ gt(x, p->l) ∧ gt(x, p->r)

lt(x, p) ≜ p==NULL ∨ x < p->data ∧ lt(x, p->l) ∧ lt(x, p->r)

x < y ∧ lt(y, p) ⇒ lt(x, p)

x > y ∧ gt(y, p) ⇒ gt(x, p)

本例是递归函数，形状分析阶段推断得到的函数返回时的形状图见图 21。

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

图 20　二叉排序树的插入函数

调用语句 p->l = insert(p->l, data)的前断言是：

order(p->l) ∧ p->data > data ∧ gt(p->data, p->l) ∧ z < data ∧ lt(z, p->l) ∧ gt(y, p->l) ∧ order(p->r) ∧ gt(y, p->r) ∧ lt(z, p->r) ∧ lt(p->data, p->r) ∧ y > p->data ∧ z < p->data ∧ y > data

上述第 1 行的断言都与指针 p->l 有关，它们蕴涵递归调用的前断言，因而得到后断言：

order(p->l) ∧ gt(p->data, p->l) ∧ lt(z, p->l)

上述第 2 行的断言根据 frame rule 可以加到这个后断言上。用谓词定义及其性质定理可从整个后断言推出 order(p) ∧ gt(y, p) ∧ lt(z, p)。

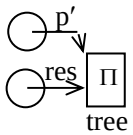


图 21　返回时的形状图

6、相关工作比较

6.1 形状图逻辑与分离逻辑的比较

形状图逻辑和分离逻辑[17]都是 Hoare 逻辑的一种扩展，都可用来对使用易变数据结构的命令式程序进行推理。分离逻辑适用于对任意指针程序进行推理，而形状图逻辑是针对使

用易变数据结构的指针程序设计的。

形状图逻辑的形状图比分离逻辑的断言分离方式包括了更多可用于程序验证的信息。比较分离逻辑和形状图逻辑的框架规则

$$\frac{\{P\}C\{Q\}}{\{P*R\}C\{Q*R\}} \quad \text{和} \quad \frac{\{W_1\}C\{W_2\}}{\{G[W_1]\}C\{G[W_2]\}}$$

分离逻辑强调分离性，导致在推导 C 引起被操作堆块的性质变化时，难以关心其他堆块与这些被操作堆块之间的联系。形状图逻辑总揽分离性和整体性，上下文描述的堆块与窗口描述的堆块是分离的，但跨越窗口的指针体现两者的联系。例如，用形状图逻辑可以保证程序不会出现内存泄漏。而用分离逻辑时，有多少个指针都指向某个堆块是不直观的，除非从每个堆块的断言中去收集，而这种收集用分离逻辑的推理规则难以表达，导致用分离逻辑难以判断内存泄漏。为了解决这个问题，Reynolds 等提出了精确断言（precise assertion）概念[18]，可惜，它的定义基于抽象机器，而不是基于断言的语法特征。

分离逻辑中访问路径的语法限制和推理规则的专门设计可防止访问路径的别名影响程序推理。在语法上，禁止使用像 C 语言 p->next->next 这样涉及多个堆块的访问路径，使得别名仅可能出现在同一个堆块的断言中。例如，若有 pure 断言 p==q，则相应的堆块可以有 $p \mapsto 5 \wedge q \mapsto 5$，这时 $p \mapsto$和 $q \mapsto$相当于 C 中互为别名的*p 和*q。若某语句要访问该堆块，在上述两个断言都存在时，分离逻辑没有规则可用来对该语句进行推理。

形状图逻辑的策略是，以形状图作为推导形状以外的性质的推理环境，用环境信息来保证在使用 Hoare 逻辑的赋值公理等规则时，断言和语句中不出现访问路径别名，以此来保证规则使用的可靠。在产生验证条件交给定理证明器时，同样可保证验证条件中无别名，使得产生的验证条件中无须分离合取这样特殊的逻辑连接词，因而不必使用专门的定理证明器，方便了验证条件的自动证明。目前，面向分离逻辑的定理证明器中[19, 7]，其 separation logic inference rules are obtained from an existing proof system for separation logic with list segments[20].

6.2 与其他形状分析方法的比较

Shape analysis attempts to discover the shapes of data structures which can be regarded as invariants in pointer programs. There has been a lot of work that builds shape analysis based on shape graphs, in which heap cells are represented by graph nodes. In particular, the elements of potentially unbounded data structures are grouped into a finite number of graph nodes by using summary nodes [22, 23, 24, 25]. This approximation of memory states leads to loss of information about the shapes of recursive data structures. 一种改进精度的方式是 associate grammars, which finitely summarize run-time heap structures, with the summary nodes of the shape graphs[26]. 由形状声明的提示和对指针操作的限制，本文的形状分析是一种精确的指针分析。在本文的形状图中，通过使用浓缩节点和谓词节点以及它们附带的个数信息来得到 potentially unbounded data structures 的精确有穷图形表示。浓缩节点是表段[17]的图形表示，但比其多了个数信息，由此来保证不丢失信息。

Chang *at al*. propose a shape analysis [27, 28] based on a shape graph representation which abstracts memory cells by edges. They describe memory states in a manner based largely on separation logic. For comparison, 他们形状图上 complete checker edge 和 partial checker edge 分别相当于本文的谓词节点和浓缩节点，更大的区别是，他们只是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是在符号断言上进行，而本文直接把形状图作为断言并直接在形状图上进行演算和推理。In addition, the shape analysis in [27, 28] is guided by invariant checkers supplied by programmers, 它们相当于易变数据结构的定义。It seems

more flexible than our method which provides only several fixed shapes. 但该文的方法很难定义循环双向链表的 inductive invariant checker，也很难分析利用数据结构的归纳性质的程序，例如用一个节点的 in-order successor (the left-most child of its right subtree)或者 in-order predecessor (the right-most child of its left subtree)来代替该节点的二叉排序树删除算法。该文给出的 analysis statistics 表中也未见有带环链表例子和二叉排序树的删除函数。Laviron *at al*. 基于[27, 28]的形状图，create an analysis that is capable of reasoning about low-level C features, such as unions, while not unnecessarily complicating the analysis of higher-level, Java-like code[29]. 而我们的形状图尚未在 low-level C features 中尝试。

6.3 与其他指针程序验证方法的比较

有不少在指针程序的分析过程中进行程序验证的研究。例如，Reps等在[23]中描述了如何用静态分析来验证处理链表结构的程序, 比如验证用链表实现的各种排序算法。但它难以验证更多的性质，例如本文例1有关链表长度的性质。因为若要验证这样的性质，系统必须添加core predicates和instrumentation predicates, 还需要为它们提供predicate-update formulae。Podelski and Wies的研究展示了the techniques developed in CEGAR(counterexample-guided abstraction refinement) scheme[30] in software verification and the focus operator[24] in shape analysis can be fruitfully integrated to enhance one another for the inference of quantified invariants[31]. They applied tool Bohne[31] to verify operations on a diverse set of data structures implementations, checking a variety of properties. Their experiments cover data structures such as (sorted) singly-linked lists, doubly-linked lists, two-level skip lists, trees, and trees with parent pointers, but do not include 循环单向链表和循环双向链表。本文采用先形状分析、后程序验证的方式，使得验证能很好地利用分析获取的信息，而要验证的性质又能较方便地表达。本文还通过使用形状系统来约束程序的行为，免去程序员提供函数前后形状图和循环不变形状图。

Madhusudan *at al*. develop a new recursive extension of first-order logic[32].The recursive logic over trees, DRYAD, is essentially a quantifier-free first-order logic over heaps augmented with recursive definitions of various types defined for location that have a tree under them. Based on this methodology, a sound and terminating procedure can prove a wide variety of algorithms on tree-structures written in an imperative language fully functionally correct. 该文的footprint由symbolic heap和DRYAD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。该文的symbolic heap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，未见处理循环链表的例子。

7、结论

In program verification, the highest expectation of shape analysis is to make the users avoid having to write loop invariants or even pre/post specifications for procedures: these are inferred during analysis [33]. 本文通过采用形状图逻辑和形状系统来进行形状分析和程序验证，免去程序员提供函数前后形状图和循环不变形状图，并体现出推理逻辑直观和推理过程简单、形状图把握指针信息全面并对自动定理证明有较好支持等优点。

今后的工作是克服本文方法只能使用几种形状的数据结构来编程的局限。我们拟对形状进行分类，不同类的形状用不同的方法来解决。本文已讨论的几种形状称为基本形状，第一步的拓展考虑基本形状各节点除了有维持本基本形状的指针外，还可以有附加指针。

（1）附加指针指向其他基本形状

附加指针是另一个独立的被嵌套数据结构的指针，例如双向链表的每个节点都指向一个

独立的单向链表，形成形状嵌套。在目前系统上进行允许嵌套形状的扩展并不困难。

（2）附加指针指向本基本形状上的节点

分成各节点附加指针的指向明确和不明确两种情况，它们将用不同方式解决。

（a）附加指针都有明确的指向。例如带父节点指针的二叉树，left-child right-sibling tree with two kinds of backward links，跳表和队列等。我们将允许程序员用指针相等断言来规范这些附加指针的特点，并考虑从这样的描述生成检查附加指针是否满足规范的代码。在需要形状检查的程序点，在完成基本形状检查后，执行对附加指针检查的代码。

（b）附加指针没有明确的指向。例如：作为基本形状的双向链表形成请求队列，其中若干节点用附加指针构成的单向链表形成就绪队列。我们需要考虑简单可行的编程约束，以保证可静态检查部分节点被附加指针构成单向链表，其他节点的附加指针等于 NULL。

参考文献

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

[2] The Coq Development Team. The Coq Proof Assistant Reference Manual (Version 8.2), 2009. URL http://coq.inria.fr.

[3] Nanevski, Aleksandar, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*: September 20-28, 2008, Victoria, BC, Canada, ed. J. Hook, 229-240. New York, N.Y.: ACM Press.

[4] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *CASSIS 2004, LNCS vol. 3362*, pages 49-69. Springer, 2004.

[5] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 234-245, 2002.

[6] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *4th Proceedings of FMCO 2005*, LNCS vol. 4111, pages 115–137. Springer, 2006.

[7] Dino Distefano and Matthew J. Parkinson. jStar: Towards practical verification for java. In Proceedings of. OOPSLA 2008, pages 213–226. ACM, 2008.

[8] Gareth Carter, Rosemary Monahan, and Joseph M. Morris. Software Refinement with Perfect Developer. In *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*. Pages: 363-373. Sep. 07-09, 2005.

[9] 技术报告：形状图理论的定理证明及其在指针程序验证中的应用，见 http://kyhcs.ustcsz.edu.cn/SGL

[10] 程序验证器原型，见 http://kyhcs.ustcsz.edu.cn/~zpli/sgltool.zip

[11] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver, Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Budapest, Hungary, volume 4963 of LNCS, pages 337–340, 2008.

[12] S. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *Proc POPL'06*, pages 115-126. ACM Press, 2006.

[13] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, doi: 10.1016/j.scico.2010.07.004, 2010.

[14] S. Lahiri and S. Qadeer. Back of the future: revisiting precise program verification using

SMT solvers. In *Proc. POPL'08*, pages 171-182. ACM Press, 2008.
[15] A, Bouajjani, C Dragoi, C. Enea, and M. Sighireanu.A logic-based framework for reasoning about composite data structures. In *CONCUR'09*, vol. 5710 of *LNCS*, pages 178-195. Springer, 2009.
[16] P. Madhusudan, G. Parlato, and X. Qiu. Decidable logics combining heap structures and data. In *Proc. POPL'11*, ACM Press, 2011.
[17] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55-74, July 2002.
[18] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proc.31th ACM Symposium on Principles of Programming Languages*, pages 268–280, 2004.
[19] Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI'11*, pages 556-566. ACM 2011.
[20] J. Berdine, C. Calcagno, and P. W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS*, vol 3328 of LNCS, pages 97-109, 2004.
[21] 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，Vol.21(No.3):124-137。（英文版见 http://kyhcs.ustcsz.edu.cn/~zpli/pl.pdf）
[22] 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.
[23] 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.
[24] 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.
[25] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proc. of PLDI'90*, pages 296-310, June 1990.
[26] 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.
[27] B. E. Chang, X. Rival, and G. C. Necula. Shape analysis with structural invariant checkers. In *Proc. of SAS'07*, pages 384-401, 2007.
[28] B. E. Chang and X Rival. Relational inductive shape analysis. In *Proc. POPL'08*, pages 247-260, 2008.
[29] V. Laviron, B. E.Chang, and X. Rival, Separating Shape Graphs. In *Proc. of ESOP '10*, LNCS vol. 6012, pages 387-406. Springer, 2010.
[30] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV'00*, pages 154–169, 2000.
[31] Andreas Podelski and Thomas Wies. Counterexample-guided focus. In *Proc. POPL'10*, pages 249~260, 2010.
[32] P. Madhusudan, X. Qiu, and A. Stefanescu. Recursive Proofs for Inductive Tree Data-Structures. In *POPL'12*, pages 123-135. ACM, 2012.
[33] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *Proc. POPL'09*, pages 289-300, 2009.
[34] P. Fradet and D. L. Metayer. Shape types. In *Proc. POPL'97*, pages 27-39, 1997.