安全 C 语言使用手册

（版本 V1.0）

安徽中科国创高可信软件有限公司

Copyright © 2017-2021 版权所有

前 言

本手册供熟悉 C 语言，并准备用安全 C 语言的程序验证器（科创验证器），进行程序验证的软件技术人员学习安全 C 语言时使用。当然，使用者还需要学习标注语言手册（《安全 C 的规范语言 SCSL 使用手册》），才能逐步胜任程序验证工作。

第一章介绍怎样用尽量少的编程约束，把不安全的 C99（C 语言标准 ISO / IEC 9899：1999）语言限制成安全 C 语言的设计思路。这主要是把程序验证中使用赋值公理时关注的是否存在变量别名，和安全语言设计中关注的类型是否可靠，这两件表面上面向不同需求但实质上紧密相关的事情综合起来，给出设计方案。本章主要通过一些简单的实例来展示变量别名和类型可靠两者的相关性。

第二章以第一章的设计方案为基础，对 C 语言的各种类型，介绍具体的编程约束。从中可以看到，为保证安全性，代码中缺少的一些信息，是由程序员在程序标注中进行补充而得到的。程序标注是程序员为程序代码写一些说明，作为给验证器的提示，以提高验证器判断程序安全性和正确性的能力。本章还简略介绍了各种主要的程序标注。

第三章介绍为易变数据结构设计的形状系统。各种易变数据结构的名称就是它们形状的名称。形状系统仿照类型系统，把形状分成基本形状和构造形状，构造形状分成嵌套形状和含附加指针的形状。本章围绕单态命名基本形状和形状的分类，对形状系统进行初步介绍，让大家对形状系统有大体的了解，并了解形状图在验证操作易变数据结构的代码上的优点。以方便学习标注语言和为代码书写标注。

程序验证是以 C 的源文件为单位，各源文件分别验证，然后再以进行集成验证。对于每个源文件，验证器对源文件的每个函数，自上而下地按照各种语句的推理规则进行演绎推理，并在重要的程序点产生验证条件。若一个函数的所有验证条件都得证，则该函数得证。

重要备注：对于第三章介绍的为易变数据结构设计的形状系统，目前的科创验证器已实现 3.1 节介绍的单态命名基本形状。3.2 节和 3.3 节的内容在相关形状的实现过程中可能还会修改。

目 录

第 1 章 安全编程语言

上世纪 70 年代，C 语言的设计目的是用来写 Unix 操作系统。C 语言提供指针算术运算和对指针的类型强制，它们可用来直接访问原始内存（*raw memory*），以便程序员编写灵活的内存操作以获得程序运行的高性能。这就使得程序员很容易用 C 语言取代汇编语言，编写许多如操作系统、设备驱动程序和编程语言的运行时系统等低层的系统程序。时至今日，C 语言仍然是编写操作系统、虚拟机监视器、编程语言的运行时系统、数据库管理系统、嵌入式软件和 web 浏览器等的一种主要编程语言，还包括运行在英特网上的各种服务器上的程序。

C 语言的指针操作没有任何保护，这给 C 语言带来很多不安全因素，导致 C 程序容易出现安全缺陷。特别是在程序运行时，通过悬空指针（*dangling pointer*）解引用（*dereference*）或数组越界访问破坏了内存中的数据结构后，程序接下去的行为可能会完全不同于从程序正文得到的想象。恶意攻击者就是根据程序中的这类错误，通过刻意准备的输入数据，有可能达到操纵程序行为的目的。近年来报告的大多数安全漏洞都是源于程序中这样的不端行为。

简言之，C 语言不是一种安全语言。在本手册中，编程语言简称为语言。

1.1 安全编程语言的定义

安全语言并没有一个统一的定义，在此通过下述几个概念[1]定义安全语言。程序运行时出现的错误称为**执行错误**（*execution error*）。有些执行错误，如非法指令错误、非法内存访问错误和除数为零错误，在它们出现时操作系统都会立即停止计算，报告发现错误的位置和错误性质。这类错误称为**会被捕获的错误**（*trapped error*）。还有一些难以捉摸的执行错误，它们引起数据遭到破坏但操作系统未能发现，因而也不会立即报告错误。例如，在没有越界检查的情况下访问超越数组边界的数据。另一个例子是程序跳到一个错误的地址，该地址开始的内存正好代表一个指令序列，使得该错误可能会有一段时间未引起会被捕捉到的事情。这类错误叫做**不会被捕获的错误**（*untrapped error*）。一个程序是**良行为的**（*well behaved*），如果它的运行过程中不出现不会被捕获错误。所有合法程序都是良行为的语言叫做**安全语言**（*safe language*）。显然，C 语言不是安全语言，因为某些 C 程序对应的目标程序在运行过程中会引起不会被捕获的错误。

保证语言安全性的通常做法是为语言设计一个类型系统（*type system*）。类型系统由一组定型规则（*typing rule*）构成，这组规则用来给构成一个程序的各种语言构造（如变量、表达式、函数和模块等）指派类型。非形式描述的定型规则的例子有：若 *M* 和 *N* 都是整型表达式，则 *M*+*N* 也是整型表达式。根据语言的类型系统，编译器或者其他程序分析工具通过静态（例如编译时）检查，动态（运行时）检查或静态检查与动态检查混合的方式来拒绝一切有**类型错误**的程序（指无法根据类型系统给其中某个语言构造指派类型的程序），例如含表达式 3 + true 的程序。如果**良类型的程序**（即经**类型检查**后无任何类型错误的程序，也称合法程序）一定是良行为的，则称该语言是**类型可靠的**（*type sound*）语言。类型可靠的语言一定是安全语言。用安全语言编写的程序，运行时就没有不会被捕获的错误。

显然，C 虽然是一个类型化的语言，但它不是类型可靠的语言。在共用体类型、含灵活数组结构体类型（结构体的最后一个域是未指定长度的数组）和参数个数可变的函数类型等方面，C 语言都有类型不可靠之处。1.2 节会给出一些简单的例子。这部分的内容可见参考文献[1]。

1.2 从杜绝别名和类型可靠相结合的角度考察C语言的安全性

别名是基于演绎推理进行程序验证的重要概念，类型可靠是安全语言的重要概念。这两个概念是紧密相关的，在程序验证过程中要做到能彻底消除别名，需要可靠的类型系统来支持。本节阐明两者之间的关系。

1.2.1 别名是程序验证中关注的一个重要问题

在编程语言中，站在语法或语义的角度，符号名字（泛指 C 语言中代表内存地址的左值表达式）分别叫做访问路径和变量。若是指针类型的变量则称指针变量或直接称指针。

若两个名字被联系到相同的内存单元（在后面会放宽一点，包括有部分内存单元重叠的情况），则称它们互为别名。通过其中一个名字修改该单元的数据意味着同时也修改了另一个名字的值，通常这不是程序员所希望的。别名的存在使得理解、分析和优化程序等变得困难。通常所说的别名分析就是试图获取程序中对判断别名有用的信息并用于判断别名。

下面先介绍对程序进行演绎推理的 Hoare 逻辑的赋值公理，然后再解释别名对使用该公理的影响。Hoare 逻辑的赋值公理如下：

{Q[E/x]}x = E{Q}

其中 x = E 是赋值语句（其中 E 是表达式），语句之后花括号中的 Q 是断言，断言就是逻辑表达式（或称逻辑公式）。语句之前花括号中的 Q[E/x]也是断言，其含义是 Q 中的 x 用表达式 E 代换后的结果断言。一段代码以及其前后的断言称为 Hoare 逻辑的三元式。上述赋值公理的含义是，若希望赋值语句 x = E 执行之后的程序状态满足 Q，则该赋值语句执行之前的程序状态必须满足 Q[E/x]，并且 Q[E/x]是保证执行赋值 x = E 之后的程序状态满足 Q 的最弱前断言。

上述赋值公理的一个实例是

{m+1 > 8}m = m+1{m > 8}，即{m > 7}m = m+1{m > 8}

该三元式的含义简单说就是，要保证 m>8，则在执行 m= m+1 之前，m 至少要大于 7。

在使用 Hoare 逻辑对程序进行演绎推理的过程中，每当使用赋值公理时，必须保证后断言 Q 和赋值语句 x = E（E 是表达式）中没有潜在的别名，否则 Q[E/x]是最弱前断言的结论不可靠。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是 q->data == 5，但实际上该语句的最弱前断言是

p == q || p != q &&q->data == 5

在此用赋值公理未能得到最弱前断言的原因是，访问路径 p->data 和 q->data 被认为是代表不同的程序对象，而实际上若 p 等于 q 时，访问路径 p->data 和 q->data 互为别名，它们代表同一个程序对象。当对 p->data 赋值时，q->data 的值也被修改了。由此可见，程序中出现别名增加了程序验证的难度，把握每个程序点的别名信息对基于演绎推理的程序验证是重要的。

像上面这个简单的别名现象，通过对断言和代码进行分析大都可以推断。一些复杂的别名现象是不可推断的，并且其不可推断的根源往往是语言的类型系统不可靠。例如，对数组元素 a[e1]赋值，若下标表达式 e1 没有越界，则它最多可能和其它未越界的数组元素 a[e2]互为别名。但若下标表达式 e1 越界，则它可能会与当前程序状态中的其它变量互为别名，和谁形成别名，依赖于编译器对程序变量的存储分配方式，即与语言的实现有关。靠基于语言语义的演绎推理是无法推断的。

别名推断对程序验证来说很重要。从上面所举的事例知道，C 的类型系统管不住下标表达式的越界，因而会出现不可推断的别名。其它不可推断的别名也都源于 C 的类型系统不可靠。因此，设计安全 C 语言时，把杜绝别名和类型可靠结合起来考虑便是件自然的事情。下面两小节分别考虑可推断的别名和不可推断的别名，并考虑如何通过程序验证者（通常就是程序员，统称程序员）所提供的标注（*annotation*）来加强安全 C 的类型系统的可靠性。

由于逆向推理与程序员静态检查自己代码的正向推理习惯不一致，在进入后面的章节之前，先介绍进行正向演绎推理的赋值公理：

{Q} x = E {∃x′.Q[x′/x] && x == E[x′/x]}

该公理的含义是，若断言 Q 在执行赋值语句 x = E 之前的程序状态中成立，则可以找到 x′，使得 Q[x′/x] && x == E[x′/x]在执行该赋值语句之后的程序状态中成立。利用该公理进行正向推理，得到的是最强后断言。下面不再使用逆向的赋值公理。

1.2.2 可推断的别名

可推断的别名是指两个名字是否互为别名可以根据语言的语义推断出来。下面按类型来逐个讨论。

1．数组元素之间的别名

限定在下标表达式都没有越界的场合，用例子来解释。

例 1.1 现有赋值语句 a[i] = 10，它的前断言是 P && a[i]==5 && a[j]==y，其中 P 是指除了 a[i]==5 && a[j]==y 以外的其它断言。怎么用正向赋值公理计算该语句的后断言？

首先需要推断 a[i]和 a[j]是否互为别名。

（1）若从 P 能够证明 i != j，即 P ==> i != j，则 a[i]和 a[j]不互为别名。这时把 a[i]代换成 5，用赋值公理得到的后断言见下面第 3 行。

{P && a[i]==5 && a[j] == y}

a[i] = 10

{P[5/a[i]] && 5==5 && a[j] == y && a[i] == 10}　备注：断言 5 == 5 可以略去。

（2）若 P ==> i == j，则把 a[j]改名为 a[i]，然后仍然把 a[i]代换成 5，用赋值公理得到的后断言见下面第 3 行。

{P && a[i]==5 && a[i] == y }

a[i] = 10

{P[5/a[i]] && 5==5 && 5 == y && a[i]==10}

基于前断言能得出 y == 5 的信息不会丢失。

（3）若 P ==> i != j 和 P ==> i == j 都不可证，但它们都可满足，则改前断言为

{i == j && P && a[i] == 5 && a[j] == y || i != j && P && a[i] == 5 && a[j] == y}

然后再依据（1）和（2），分成两种情况继续向下演绎推理。

（4）若系统所用的自动定理证明工具，例如 SMT（*Satisfiability Modulo Theories*）求解器 Z3，对

P ==> i != j 和 P ==> i == j

的证明都回答 unknown（指证明不了），则像（3）这样分成两种情况也解决不了问题。

由上述四种情况可知，在下标表达式都没有越界的情况下，同一个数组的两个元素是否互为别名是可推断的，除非遇到上述情况（4）。对于（4），有待于自动定理证明工具的证明能力提高或其它方式来消除这种情况。实际实现还得包括 P ==> i != j 和 P ==> i == j 两者中一个可满足，另一个是 unknown 的情况。

另外，为保证下标表达式没有越界，代码中每出现对数组元素的访问时，需要证明下标

表达式没有越界。□

2．指针指向对象之间的别名

类似于数组元素之间可能互为别名，以指针型名字为前缀的访问路径之间也可能互为别名。例如，先前已有例子，若 p 等于 q，则 p->d 和 q->d 互为别名。若 d 域是指针域，则 p->d 和 q->d 成为指针别名。若两个指针互为别名了，则它们加上任何相同的后缀都继续互为别名。类似数组元素，这些都属于可推断的别名。

另外，函数指针也会引起别名。但它指向的是函数，不是数据对象，与赋值公理的使用没有关系。

3．共用体变量各个域之间的固有别名

同一个共用体中的不同域名一定互为别名，因为它们共享存储单元。这是一种特殊的基于语言语义的别名，一个程序对象可能因被看成共用体的不同域，因而具有不同的类型并被解释为不同的值。当初 C 语言引入共用体类型，让不同类型的数据共享存储单元，可能是从节省内存考虑的。但是忽视了它可用于躲开类型检查，提高代码的执行效率。

例 1.2 下面是一段有共用体变量的代码

```
union{long m;char a[4];}data;            // m 和数组 a 互为别名
long x; char c1, c2, c3, c4;
data.m = x;
c1 = data.a[0];   c2 = data.a[1];   c3 = data.a[2];   c4 = data.a[3];
```

这段代码毫不费劲就把 long 类型变量 x 的 4 个字节取出来的，无需使用移位或掩码操作。它利用了 C 类型系统的不可靠，避开类型检查，达到提高效率的目的。根源在 C 的类型系统不能保证在对 x 域赋值后，对 a 域任何元素赋值前的这段时间内，拒绝对 a 数组的任何访问。□

对于程序验证来说，它在从头到尾遍历代码的演绎推理过程中，收集用断言表示的每个程序点的状态信息（有可能是若干个状态的析取）。在一个程序点某状态的信息中，不会同时出现一个共用体变量不同域的断言。对于例 1.2，在第 4 行遇到赋值语句 c1 = data.a[0]时，验证器知道当前状态中只有 m 域，该语句要取 a[0]是不可以的，因而报告错误。若赋值语句是 data.a[0] = c1，则该语句合法。得到的后断言中，有关 m 域的断言消失，新增断言 data.a[0] == c1。

与此类似的情况有，两个指针以不同的类型观点指向同一块数据。

例 1.3 一段 C 代码如下：

```
long* p; char* q;
long m = 123456789;
p = &m;
q = (char*)p;
```

执行这段代码后，*p 和*q 互为别名，但是*p 代表整个 m，其值是 123456789，而*q 仅代表 m 最左边的那个字节，其值肯定与*p 不一样，是 21。靠例 1.1 的消除别名的方式是不可行的。好在这种危险的别名是可以推断的，代码中出现这样的类型强制就报错。□

4．函数调用使用不恰当的指针型实参引起被调用函数的函数体中出现别名

以标准库函数中的串复制（即串拷贝）函数 strcpy.c 为例来展示这个问题。

例 1.4 历史上，串复制函数是

```
char* strcpy(char* dest, char* src){
    char* p;
    p = dest;    *dest = *src;
    while(*src != 0){
```

```
            dest = dest + 1;    src = src + 1;    *dest = *src;
        }
        return p;
    }
```

若调用 strcpy(s1, s2)的实参 s1 和 s2 并非指向不同的字符数组，而是指在同一个字符数组上，则该调用会引起两个形参指向同一个字符数组而破坏源字符串 s2。若两个实参之间的关系是 s2 == s1 + k（k 在合理的范围内），则调用的结果是源字符串的前 k 个字符被删除。

由于 C 的类型系统并没有限定指针型实参 s1 和 s2 必须指向不同的字符数组，因此 s2 == s1 + k 是允许的。从另一个角度说，编写一个函数，让其具有字符串复制和删除源字符串前 k 个字符这两个功能也是可以的，使用哪个功能由调用函数所给实参的情况决定。

对于函数 strcpy 的验证来说，在使用赋值公理对语句*dest = *src 进行推理时，类似例 1.1 的下标变量赋值那样，需要考虑 dest 和 src 指向的字符数组不重叠和重叠两种情况。若程序员在这个函数的标注中允许不重叠和重叠两种情况或只允许其中一种情况，则都可以判断 dest 和 src 指向的对象是否互为别名。这是用标注来补足用 C 的类型系统尚不能表达的有关参数的信息。

顺便说一句串复制函数现在的原型是

```
char* strcpy(char'* dest, const char* src);
```

它只有串复制一个功能。 □

类似的场合还有，以指针型形参为前缀和以全局变量（全局变量指外部变量和静态外部变量）为前缀的访问路径之间，也可能出现互为别名的情况。

1.2.3 不可推断的别名

无法推断的别名是指两个名字是否互为别名无法根据语言的语义推断出来。更实际的说法是，我们知道一个名字和其它某个名字形成别名，但是基于语言的语义无法知道后者是哪个名字。

1．带灵活数组的同类型结构体之间整体赋值引起的不可推断的别名

例 1.5 在 C 语言中，若变量 p 和 q 属于相同的结构体类型，那么结构体赋值 p = q 是可以的。在下面的 C 程序中，record 是含灵活数组的结构体类型，p 和 q 同属于 record 类型。

```
#include <stdio.h>
typedef struct{int n; float a[ ];} record;
record p = { 2, {1.0, 2.0} }, q = { 3, {1.0, 2.0, 3.0} };
main() { p = q;
    printf("%d,%f,%f,%f\n", p.n, p.a[0], p.a[1], p.a[2]);
    printf("%d,%f,%f,%f\n", q.n, q.a[0], q.a[1], q.a[2]);
}
```

该程序经 GCC 编译后，运行结果如下：

```
3,          1.000000,   2.000000,   3.000000
1077936128, 1.000000,   2.000000,   3.000000
```

问题是，q.n 的值为什么被破坏了？

由于 C 语言中类型系统的无奈，导致这两个属于同一类型但需要存储空间大小不一样的结构体之间能够赋值，出现这样一个不会被捕获的错误。该编译器为 p 和 q 连续分配存储空间，引起 p.a[2]和 q.n 共享存储单元，它们互为别名。赋值 p = q 之后，p.a[2]和 q.n 把这个共享单元的内容分别解释为浮点数和整数。这是基于实现而不是基于语义可以推断的别名。

类型系统的无奈是出于编译器在类型检查时只知道变量的类型，对带灵活数组的结构体类型也是这样。带灵活数组的结构体，其类型信息中不包括需要存储空间的大小，导致不能依据需要存储空间是否相同来决定是否赋值相容。 □

程序验证通过要求程序员用标注方式来增加与类型相关的信息。对例 1.5 带灵活数组的结构体类型 record，程序员必须提供类型不变式，例如\length(a) == n。该类型不变式表示，record 类型的结构体的域 n 存放的是该结构体中灵活数组 a 的长度。类型不变式是这个类型的所有元素都有的性质。有了这个类型不变式，对于例 1.5，在对赋值语句 p = q 进行演绎推理前，在证明 p.n 是否等于 q.n 时，可发现 p 和 q 的存储空间大小不一样。

2．目标模块（也称为可重定位模块）的类型信息不足引起的不可推断的别名

C 代码的编译是以源文件为单位分别编译成目标模块，最后通过连接器把这些目标模块组织成可执行程序。由于目标模块的符号表中保存的类型信息不完整，会导致连接器发现不了一个外部变量在定义它的源文件和引用它的源文件中的类型差异，从而导致程序的运行结果与期望的不一致。

例 1.6 下面左右两边分别是两个源文件的内容，k 的类型在这两个文件中有区别。

```
char.c:                    short.c:
char k = 2;                #include <stdio.h>
char j = 1;                extern short k;
                           main( ) {printf("%d\n", k);}
```

在 GCC 系统上，编译这两个文件后，运行结果是 258，而不是所期望的 2。编译 char.c 时为 k 和 j 分配两个连续的字节，用位串把它们连续表示出来则是 0000000100000010。由于 short.c 认为 k 是 short 类型，因而把这两个字节的内容当成一个短整数输出。这也是不会被捕获的错误。

这两个文件中的 k 代表运行时静态数据区中同一个对象。因两个 k 的类型不一样，也可认为是两个不同名字，因而就是不同源文件中类型不同的两个 k 互为别名。上面这两个文件中的外部变量很少，互为别名的关系可以推断。对于复杂情况，若出现这样的类型错误，互为别名关系无法推导，因它不是基于 C 语言的语义，而是基于编译器的存储分配策略。 □

程序验证也是以源文件为单位，分模块进行验证的，之后也要进行连接各模块的集成验证。针对编译器输出的目标模块中，符号表中各符号的类型信息不足的弱点，程序验证为每个源文件的外部变量和外部函数输出完整的类型信息，以保证集成验证时不会放过任何类型错误，保证集成验证成功的程序，在编译和连接时无错可报，也不会出现运行时的错误而被非正常终止（存储空间不足除外）。

3．形参有 void*类型的函数引起的不可推断的别名

下面用有 void*类型的函数给出 C 类型系统不足的典型例子。

例 1.7 C 的标准库函数中有一个可用于对任意类型的数组进行排序的函数 qsort，只要调用者能提供待排序数组的元素类型的比较函数就可以。下面的 comp_int 函数是 int 变量的比较函数。main 函数调用 qsort，但误把 double 数组 b 当作整型数组进行排序，导致结果不对。

```
int comp_int(const void* a, const void* b){
    return *(int*)a - *(int*)b;
}
int main(void){
    int i;
    double b[10] = {3.0, 5.0, 1.0, 8.0, 6.0, 7.0, 2.0, 9.0, 4.0, 0};
    qsort(b, 10, sizeof(int), comp_int);
    printf("\n after sorting:\n");
```

```
        for(i = 0; i < 10; i++){ printf("%f\t",b[i]); }
        return 0;
    }
```

输出：　0.000000　0.000000　1.000000　5.000001　8.000002
　　7.000000　2.000000　9.000000　4.000000　0.000000

上述代码中，qsort 函数的原型如下：

```
void qsort(void* p, size_t len, size_t size,
                                int (*compare)(const void* q, const void* t));
```

其中 void* p 直接作为 qsort 的形参，void* q 和 void* t 是作为形参的函数 compare 的形参。qsort 函数对调用者的要求是，所提供的对应形参*p、*q 和*t 的实参必须类型相同。但是这个要求并未体现在 qsort 的原型中，因此对调用语句进行类型检查时，不会发现上面代码中 qsort 调用的第 1 个实参类型不对，导致未能报错而排序的结果不对。结果不对是因为 b 数组的前 5 个 double 类型的数组元素被当成 10 个 int 类型的数组元素进行排序。这又是同一批数据被两种不同的观点在使用，它也是一种不可推断的别名。

若指望通过类型系统发现上述类型错误，则需把 qsort 函数原型中*p、*q 和*t 的类型都用同一个类型变量来表示，调用语句中对应的实参就必须类型相同，否则就是类型错误。但是，类型系统中一旦引入类型变量，则至少就相当于一个二阶的类型系统，这意味着 C 的类型系统要做很大的改变，这是不现实的。

对程序验证来说，解决上面问题的办法仍然是要求程序员用标注把缺少的类型信息补足。基于先前给出的 qsort 原型：

（1）第 4 个形参 compare 的函数协议需要增加断言

```
\typeof(*q) == int && \typeof(*t) == int
```

（2）qsort 的函数协议需增加断言

```
\typeof(*p) == int && \length(p) == len && size == sizeof(int)
```

在这个例子中，碰到了在标准库<typedef.h>中定义的类型。例如：

```
typedef __SIZE_TYPE__  size_t;
```

类型 size_t 是一个与机器相关的类型，它是依赖于实现的一个 unsigned 类型，其大小足以保证存储内存中该类型的任何对象。由于 size_t 是依赖于实现的类型，一般场合下，安全 C 禁止使用 size_t 类型，像本例这种把 size_t 作为函数形参类型的场合是个例外，还有就是 size_t 作为函数返回值类型的场合。把这两种例外都放到 SCSL 手册 4.8 节再进一步解释。

4．不可推断的别名还会出现在下面这些场合（但不止这些场合）

（1）数组元素 a[e]的下标表达式 e 越界，导致 a[e]不知和谁会构成别名。因此验证时，每遇到一个下标变量都要检查下标表达式是否越界。

（2）指针算术运算 p+e 的结果越界，导致*(p+e)不知和谁会构成别名。对于某个类型 t 的有效指针 p（指针有效是说该指针指向已分配且尚未释放的内存块，否则就是无效指针），验证器需知道 p 指向的区间有多少个 t 类型的元素以及 p 当前指在第几个元素上，才能完成 p+e 是否越界的检查。

（3）无效指针 p 的解引用导致*p 不知和谁会构成别名。验证器在验证过程把握每个指针分属于有效指针、NULL 指针还是悬空指针就是很重要的事情了。

（4）由有效指针 p 的类型强制转换引起的别名。若对指针 p 的类型强制，使得 p 由指向 t1 类型的变量变成指向 t2 类型的变量。若 t2 类型的变量所占空间比 t1 类型的变量大，就会使得*p 的后一部分存储空间不知和谁会构成别名。这样的类型强制必须禁止。

从本章的介绍，可得下面两点有关安全 C 语言设计的认知。

1．无法推断的别名对程序验证来说是有害的，因为程序验证是基于语义对程序进行分

析推理，而无法推断的别名并非源于语义，是验证过程中无法把握，因而无法消除的。为保证程序验证的可行和可靠，必须提高合法程序的门槛，这个新门槛的设计主要就是如何利用程序标注来使 C 的类型系统可靠，从而程序中不会出现无法推断的别名。这就是从源头上把含有不会被捕获错误的程序排除出合法程序。

2．鉴于自动定理证明工具的能力还有待提高。对 C 语言程序还要增加一些合理约束，把可推断的别名限制在易于推断的范围内，以减轻程序验证的负担。

对于程序验证来说，标注语言的设计也是验证器设计中的重要部分。从本章的介绍已经知道，标注可用来弥补类型系统的不足。程序验证中更多的事情需要标注语言来支持。

例 1.8 图 1.1 是平衡二叉树的删除节点函数的代码框架，节点的 bf 表示平衡因子，其值可取 1、-1 和 0，分别代表左右子树的高差为 1、-1 和 0。代码的第 15 和 16 行访问 t->r->bf，在先前的代码中已判断 t 是有效指针，指针 t->r 在此是否有效？未见有判断它是否有效的代码。单纯的指针分析，甚至带有简单标注的指针分析都难以排除 t->r 是无效指针的可能性，因为 C 语言和标注语言都不具备平衡二叉树的知识，因而不知道节点的 bf 值与左右子树高差之间的关系，也难以通过演绎推理发现这个性质。验证器只有得知平衡二叉树的这个性质，并且得知函数每次被调用时，第一个实参指向平衡二叉树，才可能推断出第 15 和 16 行的指针 t->r 一定是有效指针。

标注语言是一种逻辑语言，程序员用这种语言把平衡二叉树的定义和相关性质描述清楚，验证器就可以据此进行推理。 □

```
typedef struct node{int data; int bf; struct node *l, *r;}Node;
bool lower;
Node * AVLdelete(Node *t, int data) {
    … …
    if (t == NULL) { …; lower = false   // 若是空树，什么也不做
    } else {
        if (t->data == data) {   … …   // 找到了应该删除的节点，并进行删除
        } else if (data < t->data) {// 到左子树删除节点
            t->l = AVLdelete (t->l, data);
            if (lower) {// 在左子树上删除了节点，且左子树变矮，lower 是全局变量
                if (t->bf == 1) {t->bf = 0; lower = true; // 原左子树比右子树高，现左右子树等高
                } else if (t->bf == 0) {t->bf = -1; lower = false;
                                                    // 原左右子树等高，现右子树比左子树高
                } else {// 原右子树比左子树高，右子树一定非空，现需对右子树做平衡处理
                    if (t->r->bf == -1) { t = rBalance(t); lower = true;
                    } else if ( t->r->bf == 1) { t = rBalance(t); lower = true;
                    } else { t = rBalance(t); lower = false;
                    }
                }
            }
        } else { … …// 到右子树进行节点删除
        }
    }
    return t;
}
```

图 1.1 平衡二叉树的删除函数

对 C 代码进行验证，既要掌握安全 C 语言，也要学会用标注语言编写恰当的程序标注。

第 2 章 面向验证的安全 C 语言的设计

在 C99 的基础上所设计的面向验证的安全 C 语言有如下重要特点。

1．对 C 语言的各种类型增加一些编程限制（简称编程约束），使得程序对这些类型的变量的操作表现得较为规矩。

2．规定程序中需要有的各种标注，它们以一定的语法格式出现在程序注释中，是对程序的补充说明。这些补充说明主要是被验证程序的功能描述，其余的是减轻验证器负担的有益提示。

3．为 C 语言设计一个形状系统，限制利用动态存储分配和指针能够构造的易变数据结构的种类，并约束程序操作这些易变数据结构的行为。本手册的易变数据结构专指通过节点的指针域来链接的像单向链表（即通常称的单链表）、双向链表和二叉树等数据结构，易变是指在程序执行过程中，数据结构的结构完整性和节点个数等性质很容易引起变化。单向链表、双向链表和二叉树等都称之为是易变数据结构的一种形状。

4．这个设计面向顺序程序，没有考虑并发引起的数据竞争等问题。即安全 C 语言的当前版本只用于顺序程序的验证。

为描述方便，把分配在栈上的局部自动变量都称为栈数据区变量（简称栈变量），把通过 malloc 等函数动态分配的变量都称为堆数据区变量（简称堆变量），把分配在静态数据区的外部变量、静态外部变量和静态局部变量都称为静态区变量。静态区的一部分是只读数据区，存放只读数据，例如程序中出现的字符串常量。把指向栈数据区、堆数据区和静态数据区（包括只读数据区）的指针分别称为栈指针、堆指针和静态区指针（无需细分时，只读数据区指针也称为静态区指针）。在比较堆变量（指针）和其他两类变量（指针）时，合称栈变量（指针）和静态区变量（指针）为其他变量（指针）。一般情况下，把指向变量 v 的指针简称为 v 的指针。

上述每一种数据区都由若干个数据块构成。为每个外部变量、静态外部变量和静态局部变量在静态区分配的空间称为一个数据块，为每个自动变量在栈区分配的空间也称为一个数据块。数组类型的数据块还可以看成由若干个同类型的数据单元组成，数据块的长度就是数据单元的个数。其他数据类型的数据块只有一个数据单元。

2.1 编程约束

根据禁止无法推断别名和限制可推断别名的指导思想，所设计的编程约束从宏观上看主要有如下几点，大都围绕着指针类型。

1．把堆分配类型及其指针和非堆分配类型及其指针尽量区别开，尽量不要出现可同为两者的类型和指针。

2．堆区和其他数据区的关系简单，尽量避免出现从堆区指向其他数据区的指针，不滥用从其他数据区指向堆区的指针。

3．禁止对不是指在数组区间（包括动态分配的某类型的若干个元素构成的区间）的指针使用指针算术运算。

好的编程习惯基本上符合这几点。例如，对于指针类型，实际编程至少有下面这些特点。

1．不会把堆变量与其他变量混合链接成易变数据结构。

2．堆变量一般不会作为栈指针，偶尔会作为静态区指针。

3．一般来说，不会用一种含自引用的结构体类型构造不同种类的易变数据结构。含自引用的结构体类型是指它包含指向本身结构体类型的指针域。

4．用取地址算符&取栈变量、静态区变量和堆变量的地址通常用于下面的目的：

（1）一些函数要求调用者提供参数的地址，以便该函数修改实参的值。库函数 scanf 就是这样的例子。在程序员定义的函数中，例如把易变数据结构的指针 p 的地址作为实参，以便被调用函数修改 p 的值。这时 p 的地址是指针的指针。

（2）作为数组元素的指针，常用于通过指针算术运算有规律地访问一批数组元素。

对于数组类型，上述 4(2)已经给出它的一个特点，实际编程至少还有下面这些特点。

1．若结构体类型中有灵活数组作为其最后一个域，则该结构体类型通常还有记录灵活数组长度的整型域。

2. 含灵活数组的结构体类型只作为不含灵活数组的结构体类型的最后一个成员的类型。

下面先介绍对各种类型都有的约束，然后介绍对指针、结构体、共用体和数组专有的约束，最后介绍对副作用、控制结构和作用域等的限制。

2.1.1 对各种类型都有的约束

1．取地址算符&不能用于堆变量。

这个约束使得指针一般只会指向堆块的起始位置，不会指向堆块的其余位置，便于对指向堆数据区的指针的分析。例外情况由堆块中的数组名（简称堆数组名）引起。堆数组名本身就是堆块中的一个地址，在操作堆块中的数组时，使用堆数组名是常有的事。对堆数组名的使用约束见 2.1.3 节和 2.1.6 节。

2．对任何类型 t，若动态分配 t 的一个变量，必须用(t*)malloc(sizeof(t))的形式。若分配 n(n>0)个变量，必须用(t*)calloc(n, sizeof(t))的形式。对于扩大先前用 p = (t*)calloc(n, sizeof(t))分配的空间，则必须用 p = (t*)realloc(p, sizeof(t)*m)的形式，并且 m > n。

3．对有 const 修饰符的变量（包括形参）声明，不允许通过别名方式修改该变量的值。

4. 基本类型的 char、short、int、long、long long、float 和 double（以及它们带前缀 unsigned 的情况）的变量可以在这些类型之间进行强制，还有枚举类型和整型的变量也可以在这两个类型之间进行强制，但被强制的值必须在结果类型能表达的范围之内。此外，除了下面提到的一些特殊类型强制之外，不允许其他的类型强制。

5．允许把常数 0 强制为指针类型，作为 NULL 指针的值（备注：这样的隐式类型强制是存在的）。不允许把任何非 0 常数强制为指针类型。

6．指针变量的值可以强制为另一个指针类型的值来使用或者赋给后者类型的某个指针变量，但必须满足两个前提。其一是被强制指针所指向的数据块有长度性质和偏移性质断言，该数据块上没有任何验证需要关注的，被强制指针指向部分（数据块的部分数据）的性质断言。其二是强制不能造成有两个不同类型的访问路径代表这个数据块（不满足这个条件的例子见例 1.3）。在这两个限制下，这种强制能用的场合很少，基本上只能用在不关心字节序列的内容时，为操作方便，把指针的值从 char*类型强制为 short*类型的值或反方向强制。例如，在 gzip 的源码中有下面这样的代码（暂且不管下面表达式中的副作用）：

```
while (*(unsigned short*)(scan+=2) == *(unsigned short*)(match+=2) &&
        *(unsigned short*)(scan+=2) == *(unsigned short*)(match+=2) &&…)
```

其中 scan 和 match 都是 unsigned char 类型的指针，它们指向字符数组的不同位置。这里只是暂时把无符号字符数组看成无符号短整数数组，达到一次比较两个字节的目的。

7．有关变量的静态置初值

- 对于外部变量、静态外部变量和静态局部变量，C 语言的默认初值不被接受，即必须由程序显式地置初值。
- 各种类型的静态外部变量和静态局部变量的声明不仅必须带初值，而且必须各有类型

不变式或变量不变式来表达它们的不变性质，即用断言表示它们的不变性质。类型不变式和变量不变式概念和用途见标注语言 SCSL 手册的第 8 章。

- 对于外部变量和自动变量，若使用在变量声明时为变量置初值，则遵守声明时置初值方式的语义，例如，long a[10] = {0}和 long b[10] = {1}表示 a 数组的所有元素的初值都是 0，而 b[0]的初值是 1，b 数组的其它元素的初值都是 0。

2.1.2 对各种构造类型都有的约束

除了 2.1.1 节已经指出的约束外，对各种构造类型还有如下的约束。

1．结构体类型、共用体类型和数组类型这三种构造类型都可用于堆分配，并用程序标注来指明其堆分配特征（见 2.2 节），有这种标注的类型称为堆分配类型。没有这种标注的构造类型不能用于堆分配。堆分配类型不允许作为任何构造类型的成员类型。

2．堆分配的构造类型的变量只能由动态存储分配生成，非堆分配的构造类型的变量只能通过变量声明来引入。这样，堆变量和其他变量不会属于同一个构造类型。

上述对构造类型的约束并不排除非构造类型用于堆分配。例如，p 是 int 类型指针，则

p = (int*)calloc(100, sizeof(int)) 和 p = (int*)malloc(sizeof(int))

都是可行的。由于除 void 以外，其他非构造类型既可以像这两个语句一样用堆分配（且无需对类型进行标注）引入变量，又可用声明方式引入变量，因此不能称它们为堆分配类型。注意，这里的指针 p 是堆指针。

2.1.3 对指针类型的约束

除了 2.1.1 节已经指出的约束外，对指针类型还有如下约束。

1．对类型定义和变量声明的约束

（1）void*只用于表示函数形参的类型。

（2）可以出现指针的指针，但不能出现指针的指针的指针。

- 若 type 不是指针类型，则可以有声明 type *p 和 type **p，但不允许出现 type ***p。
- 若 type 是指针类型，则可以有声明 type *p，但不允许出现 type **p。
- 若 type 是指针的指针类型，则可以有声明 type p，但不允许出现 type *p。

（3）字符串常量的指针，必须声明成诸如 const char *p = “12345”的形式，而不允许是 char *p = “12345”的形式。由多个源文件构成程序时，类似的约束见 2.1.12 节。

（4）堆指针和其他指针一般情况下都不等价，无论 C 认为它们类型等价与否。这样的规定有助于避免堆指针和其他指针之间的混淆。一种例外情况在下面叙述。

- 堆指针有额外的标注，以便区别其他指针。堆指针的标注出现在其所属类型的类型声明的标注中，或者出现在堆指针自身声明的标注中，见 2.2 节。
- 因为不等价，堆指针和其他指针不能相互赋值，有助于避免堆指针和其他指针之间的混淆。这仅是对指针的限制，它不延伸到指针指向的对象。
- 所提到的例外情况是：若函数的实参和形参分别是类型等价的堆指针和其他指针，则认为它们仍然等价，但相反情况（实参和形参分别是其他指针和堆指针）则被看成不等价。因为能够应用于堆指针的操作多于能够应用于其他指针的操作，例如 free 函数，禁止后一种情况可避免对实参施加它不能接受的操作。

2．对操作的约束

先介绍有效指针 p 的属性\length(p)和\offset(p)，它们是 SCSL 语言的两个内建函数，用于下面描述对指针操作的约束。有效指针是指有指向对象的指针，即不是 NULL 指针也不

是悬空指针。\length(p)等于有效指针 p 所指向的数据块的长度，\offset(p)等于有效指针 p 所指向的单元在数据块中的偏移。

若有效指针 p 指向某个类型的单个变量，则断言\length(p) == 1 和\offset(p) == 0 成立。对于赋值 p = (int*)calloc(10, sizeof(int))（这时 p 必须是指向堆数据块的堆指针），若分配成功，得到相当于数组类型的数据块，赋值后\length(p) == 10 和\offset(p) == 0 成立。若有赋值语句 q == p + 3，赋值后\length(q) == 10 和\offset(q) == 3 成立，表示 q 指向有 10 个元素的数据区中的第 4 个元素（偏移为 3）。对于数组 int a[100]，则断言\length(a) == 100 和\offset(a) == 0 成立。有关数组的这两个属性在 2.1.6 节还有进一步的介绍。

显然属性\length(p)和\offset(p)是用于防范 p 越界。

（1）指针赋值运算的限制。

- 只要字符指针 p 指向字符串常量，p 的值就只能赋给有同样 const 修饰符的指针型变量 q。
- 在函数调用场合，若这样的 p 是实参，则对应形参 q 必须有同样的 const 修饰。

（2）指针关系运算的限制：

- 各类指针都可以进行相等与否比较运算。
- 指在同一个数据块的栈指针和静态区指针，可以进行大小比较运算，否则不行。
- 堆指针不能进行大小比较运算。

（3）栈指针和静态区指针的赋值和算术运算受到如下限制。

首先肯定一点，生存期短的指针可以指向生存期长的数据，但反之不行。

具体来说，对于指针赋值语句 p = q、p = q + e 和 p = &r（其中 p 和 q 是指针类型的访问路径，r 是访问路径，e 是整型表达式，“+”泛指加减运算符）：

- 若赋值号右部表达式的结果指针指向静态区变量或常量，则 p 可以是全局指针、局部指针或者堆上的指针。
- 若赋值号右部表达式的结果指针指向局部变量，则 p 只能是局部指针。
- q 可以是声明变量的地址、数组名、指针变量，甚至是存储在只读静态区的字符串常量的指针，如语句 p = “abc”。
- 若 q 是形参，稳妥的做法是默认其对应实参是调用函数的局部指针。

对于指针算术运算，允许求两个指针的差 p - q，限定 p 和 q 是为同一个变量所分配的存储区域的指针，例如是同一个数组的数组元素的指针。

在用指针算术运算 p + e 的结果作为被访问变量的地址时，必须保证\offset(p) + e ≥ 0 && \offset(p) + e < \length(p)。

（4）禁止对堆指针进行算术运算。

这是为了保证堆指针都指在堆块的起始位置。

（5）指针作为形参（包括形参的域或元素是指针的情况）时，在函数协议中的函数前条件中，需要有标注指明它是有效指针、悬空指针还是 NULL 指针。

（6）在有效指针作为形参时，对应实参还有进一步的限制。

- 若形参 a 是指针但不是指针的指针，对应的实参为 b，则它们必须满足：0 <= \offset(b) <= \length(b) - 1 && \offset(a) <= \offset(b) && \length(a) - \offset(a) <= \length(b) - \offset(b)，以保证在函数中对 a 指向数据块的操作不会引起调用时越出 b 指向的数据块。
- 若形参 a 是指针的指针，对应的实参为 b，这时并不要求*b 和所有可能的*(b+e)都是有效指针。
- 任何指针型实参之间以及指针型实参和被调用函数可能赋值的指针型全局变量（或全局变量的左值）之间都不能相等，也不能一个是另一个的前缀，也不能虽不相等但指向同一个数据块（例如是同一个数组的不同数组元素的地址）或指向同一个易变数据结构（例如都

指向同一个单向链表的节点），除非在函数协议的前条件中已经指出了这种相等或指向同一个数据块或指向同一个易变数据结构。简单地说，任何两条以形参名开始的不相同指针型访问路径默认为不相等，除非函数协议的前条件已经指明它们相等。这是为了保证函数调用不会引起从函数前条件推导不出的别名。有关函数协议的内容见标注语言手册第 4 章。

（7）无效指针不能解引用。

（8）动态存储分配库函数的使用要加以约束：

• malloc 等堆分配函数调用的结果只能赋给堆指针，下一个语句必须是判断 malloc 返回值是否等于 NULL 的语句。

• free 函数只能用于指向堆数据块的有效指针，它不能用于其值等于堆数据块的起始地址，但其类型不是堆数据块指针类型的有效指针。

例 2.1 下面是涉及动态存储分配的一个简单 C 程序。

```
#include <stdlib.h>
typedef int array[100];   // 以后要用标注指明用于堆分配
void main(){
    array *p;             // 以后根据类型标注可知，p 是堆指针
    p = (array*) malloc(sizeof(array));
    (*p)[5] = 10;
    free(*p);   // free(&(*p));
}
```

在上述的程序中，p 是 array 类型的指针，*p 是 array 类型的变量，*p 是数组名。GCC 编译器接受该程序，安全 C 语言不认可。安全 C 语言拒绝 free(*p)，因为 p 是 array 类型的指针，而*p 是 array 的元素类型的指针，不是堆数据块的指针，虽然它们的值一样。安全 C 语言也会拒绝 free(&(*p))，虽然 p 和&(*p)的类型是一样的，但&(*p)在取堆块的地址，这是安全 C 所禁止的。 □

（9）与输入输出有关的库函数的使用要加以约束：

• 对于有返回值的输入输出库函数的调用，若返回值的各种可能取值中，有指示出错的返回值，则其下一个语句必须是判断返回值是否指示出现错误的语句。

• 只有 printf、fprintf(stdout, ...)和 fprintf(stderr, ...)的使用可以例外，因为这些函数调用的执行结果是否出现异常，与代码的验证无关。

• 对于 fopen 和 fropen 调用得到的 FILE*类型的文件指针 p，不允许通过 p 修改文件描述符的内容。

（10）有关指针形参的修饰符

• 对于函数的堆指针型形参 p，若 p 的声明中没有限定它是不能被赋值的 const 修饰符（即不是像 int* const p 这样的声明），则必须在函数的前条件中有某个逻辑变量等于 p 的断言，其缘由在例 2.10 中有解释，逻辑变量的含义见标注语言手册第 7 章。

• 对于函数的其他指针型形参 p，若 p 的声明中没有限定它是不能被赋值的 const 修饰符，则上述强求可放宽到仅在下述场合是必须的：若也没有 const 修饰符限定 p 所指向的对象不能被赋值（例如像 const int* p 这样的声明），除非函数后条件中的断言不涉及 p 在函数入口处所指向的对象。

• 若有限定形参指针 p 不能被赋值的 const 修饰符，则上述两种情况下的逻辑变量都不是必须的。

（11）return e 语句中的指针表达式 e 不能是指向局部变量的指针。若函数后条件中没有指明，则 e 也不能等于某个全局指针。

（12）对于指向只读数据区的字符串指针，不能修改其指向的字符串。

例 2.2 下面是有运行错误的一个简单的字符串复制程序。

```
#include <string.h>
void main() {
    char p[] = "abcd";    char *q = "efgh" ;
    strcpy(q, p) ;          // 运行时报告 Segmentation fault
}
```

p 是一个字符数组，q 是字符指针，字符串"efgh"存放在只读数据区。strcpy(q, p)引起段错误，因为它试图修改存放在全局只读数据区的字符串"efgh"。而反向的 strcpy(p, q)不会引起运行时的错误，因为"abcd"不是存放在只读数据区。 □

main 函数的形参 argv 的一组指针被认为是指向只读数据区的字符串指针。

对操作易变数据结构的程序的特殊约束在第 3 章介绍。

2.1.4 对结构体类型的约束

除了 2.1.1 和 2.1.2 节已经指出的约束外，对结构体类型还有如下约束。

1．对类型定义和变量声明的约束

（1）在堆分配的结构体类型中，所有的指针域都是堆指针或静态区指针。

（2）含自引用的或相互引用的结构体类型只能作为堆分配类型。

（3）含灵活数组域的结构体类型必须有类型不变式（见 2.2 节），类型不变式表示某个整数域的值始终（从初始化之后程序点开始算起）等于灵活数组的长度。

2．对操作的约束

（1）对于堆分配的含形状标注的结构体类型，每次只允许动态分配其一个变量。

（2）除了用带初值来声明含灵活数组的结构体外，只能用 malloc 函数为含灵活数组域的结构体类型 t 的变量分配空间，它必须是(t*)malloc(sizeof(t) + e * sizeof(t_a))的形式，其中 t 是结构体的类型，t_a 是其中灵活数组 a 的元素类型，e 是灵活数组长度表达式。不管是静态声明还是动态分配，结构体的初始化要保证类型不变式中的\length(a)等于某个整数域的断言成立。并且以后每次修改该结构体时都要维持该不变式。

（3）对于含灵活数组域 a 的结构体类型 t，t 类型的结构体整体赋值时，所涉及的两个结构体的 a 数组的长度必须一样。

2.1.5 对共用体类型的约束

除了 2.1.1 和 2.1.2 节已经指出的约束外，对共用体类型还有如下约束。

1．对类型定义和变量声明的约束

（1）和结构体类型的（1）一样。

（2）和结构体类型的（2）一样。

（3）不允许共用体类型有灵活数组域，也不允许它有含灵活数组域的结构体域。

2．对操作的约束

（1）和结构体类型的（1）一样。

（2）若 a 和 b 是共用体变量 u 的域，那么 a 和 b 互为别名。它有别于通常的别名：在给 u.a 赋值后，在没有给 u.b 赋值前，不能有对 u.b 的访问。

2.1.6 对数组类型的约束

除了 2.1.1 和 2.1.2 节已经指出的约束外，对数组类型还有如下约束。

1．对类型定义和变量声明的约束

（1）对于堆分配数组，可以像例 2.1 那样先声明类型，然后再动态分配。也可以未声明堆分配的数组类型，而直接动态分配堆数组。例如，若 p 的类型是 long*，则语句

p = (long*)calloc(e, sizeof(long));

使得 p 指向长度为 e 的 long 数组（e 是整型的线性表达式），但要用标注指明 p 是堆分配的数据块的指针（见 2.2 节）。

（2）数组的成员若含堆分配类型的指针，则这些指针自动受一种弱类型不变式（见 2.2 节）的约束：在使用这种数组的函数的入口和出口，这些指针都指向相应易变数据结构的标准形状（见第 3 章）的实例。若是链表指针时，这些指针也可以受另一种弱类型不变式的约束：链表长度等于所在成员中另一个域的值。

（3）数组名也有长度属性和偏移属性，偏移属性恒为 0（不管显式出现还是默认），长度属性分成以下几种情况。

- 对于指定大小的数组声明，例如 int a[10][20]，断言\length(a) == 10 成立。
- 对于未指定大小但带置初值的数组声明，例如 int a[] = {1,2,3,4}或 char a[] = “abc”，则断言\length(a) == 4 成立。
- 对于不作为形参的未指定大小的数组声明，例如 int a[]，a 是无效指针，无长度属性。
- 对于作为形参的数组声明和未指定大小的数组声明，例如 void f(int a[10][20], ...)和 void f(int a[][20], ...)，它们本质上与指针形参声明 void f(int(*a)[20], ...)一样。它们的长度和偏移属性都由函数前条件给出（这时的偏移不一定是 0）。在这个场合，长度和偏移属性的默认值分别是\length(a) == 1 和\offset(a) == 0。这个默认仅对形参 a 本身，不延伸到 a 指向的对象。例如，对于 void f(int** p, ...)，仅默认\length(p) == 1，不默认\length(*p) == 1。

这两个默认值是由验证器添加到函数前条件、后条件、循环不变式和程序点断言等标注中的。有一点需要注意，对一个函数来说，若上述这些标注中（包括多协议和多命名行为协议的场合，多协议和多命名行为协议见《安全 C 的规范语言 SCSL 使用手册》第 4 章）任何地方出现有与\offset(a) == 0 相矛盾的断言，则系统放弃自动添加，由程序员自己写出各处的\offset 断言。对\length 断言也是这样。

2．对操作的约束

（1）对数组名的算术运算的限制同于对栈指针和静态区指针的算术运算的限制。

（2）在数组作为形参时，对相应实参的限定如下。

对形参数组声明 a，例如 void f(int a[10][20], ...)，或 void f(int a[][20], ...)，或 typedef int array [10][20]; void f(array a, ...)，由于它们本质上与 void f(int(*a)[20], ...)一样，因此函数调用 f(b, ...)中的实参 b 必须满足的条件也是 0 <= \offset(b) <= \length(b)-1 && \offset(a) <= \offset(b) && \length(a) - \offset(a) <= \length(b) - \offset(b)。

（3）对于堆变量中数组名的使用需要注意：堆分配的数组，包括出现在堆分配的结构体或共用体中的数组，不允许把数组名赋给任何堆数据块的指针，不管该数组名的值是否等于某个堆数据块的起始地址。

例 2.3 图 2.1 是把静态区数组 a 的内容复制到堆分配数组的两个程序，两个程序都是安全 C 的程序。对于左边程序来说，赋值 q = *p 把堆数组名赋给了同类型的指针变量，而不是堆数据区的指针。右边程序也是安全 C 的程序，它通过函数调用来完成数组的复制。在函数 f 中，形参 p 没有标注为堆指针；实参也不是堆数据块的指针，只要不出现数组越界，则不会影响程序的安全执行。 □

```
#include <stdlib.h>
typedef int array[100];   //标注为堆分配类型
int a[100];
void main( ){
      array *p;   //p 标注为堆指针
      int m, *q, *s;
      p = (array*)malloc(sizeof(array));
      q = *p; s = a;
      for(m = 0; m < 100; m++)
            *q++ = *s++;
}
```

```
#include <stdlib.h>
typedef int array[100];   //标注为堆分配类型
int a[100];
f(int *p){ //标注\length(p)==100 && \offset(p)==0
      int m, *q, *s;
      q = p; s = a;
      for(m = 0; m < 100; m++) *q++ = *s++;
}
void main( ){
      array *p;     //p 标注为堆指针
      p = (array*)malloc(sizeof(array));
      f(*p);
}
```

图 2.1 数组复制的两个不同程序

（4）不允许函数调用引起别名。不同名字可能引起别名已经在 2.1.3 节第 2 点加以限制，同一个数组的不同数组元素指针可能引起别名也已经在 2.1.3 节第 2 点加以限制。

2.1.7 对位运算的约束

位运算的运算对象称为位向量，位向量的长度按其占用的字节数统计。对于位向量的类型限制如下。

1．参与位运算（包括移位运算）的位向量必须长度一致，所允许的长度就是 C 语言整型（char、short、int、long 和 long long）的字节数。

2．移位运算的左右操作数都必须为无符号数或者非负的有符号数，并且计算结果不能发生溢出。对其他位运算操作符没有特别限制。

3、对于位运算和算术运算的混合运算（包括大小比较运算）没有限制，但同样也要求计算结果不能发生溢出。

对于移位运算，不要忘记 C99 对移位运算的约束。其约束是禁止那些结果没有定义或依赖于实现的左移和右移，具体是如下 3 点。

1．右操作数为负数或者右操作数大于等于左操作数的位宽，行为未定义。

2．对于左移<<：左操作数须为无符号数或者非负的有符号数，并且结果可在结果类型中表示，否则行为未定义。

3．对于右移>>：左操作数须为无符号数或者非负的有符号数，若为有符号负数，则结果依赖于实现。

在实际代码中使用位运算时，还需要考虑避免使用可能出现数值上溢的操作。

1．移位运算常用来获取一个位向量的某些连续位。由于 C 语言并没有位向量类型，该数据仍然被看成整型数据。左移该数据很可能引起验证器报告数值上溢（除非从该数据的最大可能值和左移位数能判断没有上溢）。遇到这样的情况，需改用掩码方式来达到取位向量的连续若干位的目的。

2．若试图通过把多字节类型的整数强制为少字节类型的整数的方式，来略去前者的高位字节时，同样也可能出现数据上溢。也需要改用掩码方式来达到目的。

2.1.8 对含副作用的表达式的约束

副作用是指那些会改变执行环境的状态的操作，例如表达式 i++、赋值表达式和修改全

局变量的函数调用等。副作用与第 1 章讨论的别名没有联系。若一个表达式内部的计算次序没有明确规定，并且该表达式存在多个副作用，则其结果依赖于实现。从程序验证的角度来说，不允许程序中出现计算结果依赖于实现的副作用。下面先讨论拒绝出现其结果依赖于实现的表达式时，哪些是需要关注的表达式。

1．逗号表达式

逗号表达式 $e_1, e_2, \ldots, e_n$ 中的这些表达式有严格的计算次序，因此逗号表达式本身不属于所关注的表达式，但它的各个子表达式有可能属于被关注的表达式。有可能是指，子表达式 e_i（$1 \le i \le n$）可能被关注。当逗号表达式是更大范围表达式的子表达式且该逗号表达式中有被关注的子表达式，则该逗号表达式本身也属于所关注的表达式。

2．赋值表达式

（1）赋值表达式 $e_1 = e_2$ 中的赋值是最后执行的，但其中两个表达式的计算次序没有规定。因此赋值表达式本身及其子表达式都是需要关注的表达式。

对于复合赋值表达式，例如 x += a * b，同样，复合赋值表达式本身及其子表达式都是需要关注的表达式。

（2）连续赋值表达式 $e_1 = e_2 = \ldots = e_n$ 中这若干个赋值有严格的执行次序，但这些表达式的计算次序没有规定，因此连续赋值表达式本身及其子表达式都是需要关注的表达式。

3．条件表达式

条件表达式 $e_1 ? e_2 : e_3$ 的三个表达式有严格的执行或不执行的规定，要执行的两个表达式有严格的执行次序。因此条件表达式本身不属于被关注的表达式，但它的子表达式是。但是，当条件表达式是更大范围表达式的子表达式时，则条件表达式本身也可能属于所关注的表达式。

4．循环语句中的表达式

while(*e*) *s* 语句、do *s* while(*e*)语句和 for(e_1; e_2; e_3)语句中的表达式都属于要关注的表达式。

5．选择语句的表达式

选择语句（if 和 switch）的表达式属于要关注的表达式。

6．函数调用的实参表达式

函数调用 f($e_1, e_2, \ldots, e_n$)的各个实参表达式属于要关注的表达式。函数调用本身也是被关注的表达式。

对于函数调用，需要把与函数调用本身相关副作用解释一下。若函数调用 f($e_1, e_2, \ldots, e_n$)的计算修改某个由全局变量名开始的访问路径所代表的变量的值，或者修改某个由指针型实参表达式 e_i（$1 <= i <= n$）（包括数组实参）的名字开始（且并非只是这个名字）的访问路径所代表的变量的值，则该函数调用本身有副作用。

7．逻辑运算表达式

含逻辑运算&&、||和!的表达式属于要关注的表达式

8．return 语句的表达式

return e 语句的表达式 e 属于要关注的表达式。

9．声明语句中的置初值表达式。

置初值表达式属于要关注的表达式。

下面几条约束是安全 C 语言对副作用的限制。

1．逻辑运算的表达式不能有副作用。这个限制是为了避免逻辑运算中的短路计算给程序验证带来的麻烦。这就不必把逻辑运算表达式作为被关注的表达式。

2．while(*e*) *s* 语句和 do *s* while(*e*)语句中的控制表达式 *e* 不能有副作用。for(e_1; e_2; e_3)语句中的控制表达式 e_2 不能有副作用。这样，只需把 for 语句的 e_1 和 e_3 作为被关注的表达式。

3．对允许有副作用的被关注表达式 *e*（包括赋值语句 $e_1 = e_2$;中的 $e_1 = e_2$），若 *e* 中某处有通过++或--运算修改变量 v 的副作用，则除了在表达式 *e* 的计算中仅使用其左值的地方（例如，赋值语句 v = … ;中，赋值算符左边的 v，是仅使用 v 左值的地方），变量 v 不出现在 *e* 的其它地方。例 2.4 是解释最多只允许一个修改变量 v 的缘由的简单例子。

4．对允许有副作用的被关注表达式 *e*，若 *e* 中有通过函数调用 f(e_1, e_2, …, e_n)修改全局变量或指针型实参表达式 e_i（1 <= *i* <= *n*）（包括数组实参）的名字开始（且并非只是这个名字）的访问路径所代表的变量 v 的副作用，则除了在表达式 *e* 的计算中仅使用其左值的地方，在整个 *e* 中只允许这个函数调用访问变量 v，即 *e* 中其它地方没有 v，并且其它函数调用也不访问 v。例如，在表达式 f(e_1, e_2, …, e_n) + g(d_1, d_2, …, d_m) + …中，若函数调用 f(e_1, e_2, …, e_n)修改全局变量 v，则该表达式的剩余部分 g(d_1, d_2, …, d_m) + …即不显式出现 v，也不会通过函数调用访问 v。

例 2.4 下面的程序

```
#include <stdio.h>
void main(){
    long i = 0;
    printf("%ld\n", (++i)+(++i)+(++i));
}
```

在不同的机器上或使用不同的编译器对其编译，它可能有不同的输出。这是由于 C 语言没有规定表达式的计算次序引起的。禁止 i 在表达式(++i)+(++i)+(++i)中出现多次，就是为了保证该表达式的值是确定的，以避免不确定性给程序验证带来麻烦。 □

2.1.9 对控制结构的限制

1．在 switch 语句中，对于由标记 case 或 default 开启的每个分支，该分支的语句序列是指从标记之后的语句开始一直到达下一个标记或是到达 switch 语句结束之前的所有语句。每个这样的语句序列的最后一个语句必须是 break、return、continue、goto，abort、exit 或_exit，除非该语句序列为空或者是到达 switch 语句结束的最后一个语句序列。

2．一个 switch 语句的 case 或 default 标记不能出现在该 switch 语句体中的选择或循环语句中。也就是 switch 语句引起的执行跳转不能跳到选择语句和循环语句体中的某个语句。

3．goto 语句不得转到选择语句和循环语句体中的某个语句。逆向的 goto 语句不能从循环语句内跳转到循环语句外面。

4．逆向的 goto 语句可能会引起一段代码反复执行，这段代码被认为构成一个循环，同样不得有其它 goto 语句转到这个循环内部的某个语句。

这是限制一些非结构化的编程。

2.1.10 对变量作用域的限制

1．在函数体中，所有局部变量声明在作为函数体的那个程序块（*block*）中，也就是不认为函数体中可以有嵌套的作用域。

对名字作用域加这个限制，根源是 C 语言有作用域概念，标注语言 SCSL 虽也有作用域概念，但它不涉及程序变量。若函数体中嵌套的两个程序块都有变量 m 的声明，则在内程序块的程序点断言中出现 m，不知对应到哪个 m。

2．若函数形参与全局变量同名，则函数协议以及该函数中的所有断言中有关这个名字的断言都是关于该形参而不是那个同名全局变量的。

这是标注语言中有关函数协议的约定，因为它与 C 语言的作用域有关，因此列在这里。例如，若函数 A 有形参 m，A 通过调用函数 B 修改全局变量 m，则 A 的函数协议既要表达对形参 m 的约束，又要表达全局变量 m 的变化。这时难以判断 A 的函数协议中的 m 是代表形参 m 还是全局变量 m。程序员只有将其中的某个 m 换个名字才能解决问题。

2.1.11 不允许定义参数个数可变的函数

例 2.5 下面是一个参数个数可变的求和函数。

```
#include <stdarg.h>
double sum(int count, ...){
    va_list dp; int j; double total = 0;
    va_start(dp, count); // 让 dp 指向 sum 调用实参表中省略号部分的起始参数
    for(j = 0; j < count; j++) {total += va_arg(dp, double);}
    va_end(dp); // 置 dp 为 0
    return total;
}
```

其中宏 va_list、va_start 和 va_end 定义在文件 stdarg.h。参数个数可变的函数是容易引起无法推断别名的语言机制。隐患来源于函数定义参数表中的省略号。上述宏引用 va_arg(dp, double)隐含地要求所有的 sum 调用，除了第 1 个实参外，其余实参都是 double 类型。sum 函数还隐含地要求第 1 个参数的值等于随后参数的个数。 □

2.1.12 对多文件组成程序的限制

一个类型、不能修改的变量（变量声明中加了 const 修饰符，以下简称常量）、变量或函数，被多个 C 源文件（指以 c 为后缀名的源文件）用到（定义、赋值、引用或调用等）时，称它被多个 C 源文件共享，这些文件称为共享它的 C 源文件。对于被多个 C 源文件共享的类型、常量、变量和函数，它们定义或声明在程序中出现的位置规定如下，这些规定是为了保证程序验证能够以模块（C 源文件）为单位进行。

1．共享类型的类型定义必须出现在共享它的 C 源文件都包含的头文件中，不得分别定义在这些共享它的 C 源文件中。

2．共享常量的常量定义（如 const int m;）必须出现在共享它的 C 源文件都包含的头文件中，但给出常量值的定义（如 const int m = 100;）必须也只能出现在共享它的某个 C 源文件中。

3．共享函数的函数声明必须出现在共享它的 C 源文件都包含的头文件中，函数定义必须也只能出现在共享它的某个 C 源文件中。并且被共享函数调用的函数的函数声明也必须出现在这些 C 源文件都包含的头文件中。

4．共享变量的外部声明（有 extern 的声明）必须出现在共享它的 C 源文件都包含的头文件中，变量声明（无 extern 的声明）必须也只能出现在共享它的某个 C 源文件中。对于初值是字符串常量的共享字符指针变量或字符数组变量，例如 char *p = “12345”，其在相应头文件中的外部变量声明必须有 const 修饰符，例如 extern const char* const p，表示 p 指向的存储区域不能被修改并且 p 自身也不能被修改。即这个 p 只能用作字符串“12345”的指针。或者声明成 extern const char* p，表示 p 自身还是能修改的。

以共享变量为例，若没有这样的限制，则它们在共享 C 源文件中表现出的类型可能会不一样，导致程序的结果不是所预期的，见先前的例 1.6 和下面的例 2.6。

例 2.6 下面左右两边分别是两个 C 程序文件 file1.c 和 file2.c 的内容。

```
#include <stdio.h>                    #include <stdio.h>
char *p = “0123456789”;               extern char p[];
void f ( ) {                          void main ( ) {
      printf (“%s\n”, p);                   f ( ); p[1] = ‘1’; f ( );
}                                     }
```

用命令 cc file1.c file2.c 对这两个文件进行编译和连接。运行目标程序得到：

0123456789

Segmentation fault

从 file1.c 知道，p 指向的字符串分配在只读数据区。在 file2.c 第二次调用 f 前，对 p[1]的赋值试图改变只读数据区的内容，引起段错误。 □

5．共享变量的外部声明不允许有初始值。多个 C 源文件共享的常量或变量的带初值的声明，它只能出现在共享它的某个 C 源文件中，因而不能出现在它们共享的头文件中。

2.1.13 保证验证结果独立于编译器的限制

1．各种类型的取值范围

程序的正确与否依赖于某种数值类型的取值范围时，程序验证的结果并不独立于编译器。好在各种数值类型可取的最大值和最小值定义在 C 的标准库 limits.h 中，在验证器使用的标准库和编译被验证程序所使用的标准库一致时，不用为数值类型的取值范围而担心验证结果是否可靠。

2．各种类型占用的字节数

根据 C99 的规定，char、short、int、long 和 long long 类型（包括带 unsigned 的这些类型）占用的字节数分别为 1、2、2、4 和 8，若使用 int8_t 和 int16_t 等类型名则更清楚。float、double 和 long double 类型分别占用 4、8 和 16 字节。

（1）若程序代码中出现算符 sizeof 应用到上述类型或这些类型的变量，则其值分别对应到上述字节数。

（2）若程序代码中出现 sizeof 应用到其它类型或它们的变量 t，这样的 sizeof(t)就有可能出现在断言中。

- 若程序的某些性质依赖于这些 sizeof(t)的具体值，则这些性质可能验证不了。
- 若程序的待证性质并不依赖于这些 sizeof(t)的具体值，则这样的程序可以验证。正确性并不依赖于这些 sizeof(t)的具体值是指：sizeof(t)在代码和标注中都不得参与数值计算和比较运算，包括不得出现在代码的赋值表达式以及循环语句和分支语句的条件控制表达式中。

3．2.1.8 节对表达式副作用的限制完全是为了保证验证结果独立于所用的编译器。有某个变量有多个副作用的表达式，由不同编译器生成的目标代码的运行结果可能不一样。

4．不允许将任何指针值强制为整数类型的值，原因也在于此。

2.1.14 程序中使用的标识符不能与 SMT-LIB 的保留字重名

程序中所用的标识符，变量名、类型名、结构体和共用体的域名和函数名等，不能和 Z3 所采用的 SMT-LIB 中的内建符号（*buildin symbol*）重名。Z3 的内建符号有下面这些，它们有些作为保留字（*reserved word*）使用，有些是作为关键字（*key word*）使用。虽然程序中的标识符可以和 Z3 的某些关键字重名，建议程序员还是不要用，避免给分析 Z3 的代码带来干扰。Z3 的内建符号如下。

- 类型：Int、Real、Bool、Array、BitVec、String、FloatingPoint、NUMERAL、DECIMAL、STRING。
- 特殊值：true、false、RNE、RNA、RTP、RTN、RTZ。
- 运算符：select、store、ite、and、or、xor、not、let、par、as、distinct、forall、exists、rem、bvshl、bvashr、div、mod、bvlshr、bvand、bvor、bvxor、bvxnor、bvadd、bvsub、bvneg、bvmul、bvudiv、bvurem、bvsrem、bvsmod、bvnot、bvule、bvult、bvuge、bvugt、bvsle、bvslt、bvsge、bvsgt、bveq、int2bv、bv2int、concat、extract、to_fp、to_fp_unsigned、nil、cons。
- 命令名：assert、simplify、echo、eval、eufi、help、include、labels、exit、push、pop、display、maximize、minimize、reset。

2.2 程序标注

安全 C 语言要求程序员为程序代码写一些说明，作为给验证器的提示，以提高验证器判断程序安全性和正确性的能力。这些说明称为程序标注，或简称标注。在第一章已经见到用标注给验证器提供信息，以弥补 C 类型系统不足的一些例子。因此标注语言的的设计也可看作为安全 C 语言设计的一部分。程序标注写在 C 源文件的注释中，以保持源文件仍然可编译。作为标注的注释必须以/*@开始并以*/结束，或者以//@开始到本行结束。这两类注释中的其余部分都被认为是标注。注释也可以出现在标注中，它们只能使用以//开始到当前行结束的形式。

程序标注本身必须遵守一定的语法，为此设计了面向程序验证的安全 C 语言的规范语言 SCSL。程序标注分成全局标注和语句标注两类。本节简略介绍程序标注，并给出一些简单的例子，程序标注的详细介绍在标注语言手册中。

2.2.1 全局标注

全局标注有以下几种。

1．函数协议

函数协议包括函数的前条件和后条件。其中函数前条件是指函数形参和在函数中使用的全局变量在函数入口点必须满足的性质；而函数后条件是指在函数前条件得到满足的情况下，函数保证返回点必定满足的性质。函数的前条件和后条件也称为函数的前断言和后断言。

例 2.7 图 2.2 是二分查找函数及其标注。在第 11 行开始的标注构成该函数的协议，其中关键字 requires 和 ensures 之后的断言分别是该函数的前条件和后条件。后条件中的\result 表示函数值。前条件主要表示数组中的数据必须有序。后条件主要表示，若存在某个数组元素等于 val，则函数返回该元素的下标，否则返回-1。另外，后条件还表示了数组的有序性仍然被保持等。

该例中有关引理的标注和有关循环不变式的标注在后面解释。 □

函数协议还包括：

（1）函数可能修改的全局变量。提供这个信息是为了方便程序验证。

（2）函数终止的条件。程序验证检查该条件是否能保证函数终止。

（3）函数异常终止时的特性。ensures 子句并未包括函数异常终止状态（调用 exit 之后的状态）时的特性，因此有 exit 调用的函数需要给出函数在异常终止时具有的性质。

2．类型不变式

类型不变式是指该类型的任何变量都具有的性质。例如对于例 1.1 中带灵活数组的结构体类型：

typedef struct{int n; int a[];} record

可以有类型不变式\length(a) == n，用以指明灵活数组的大小。并非一定需要为类型不变性质给出不变式标注，但是有这样的标注给验证器和程序员本身都带来方便（在标注语言手册中介绍）。

类型不变式有强弱之分。强类型不变式必须在程序执行的任何时候都有效。弱不变式必须在函数的边界（函数的入口和出口）都有效，但不要求在这两点之间保持有效。

3．变量不变式

变量不变式可用于全局变量，也可用于静态局部变量。例如一个整型全局变量 a，在其生存期内始终不会小于 0，则可以为它标注不变式 a >= 0。同样，并非一定需要为变量的不变性质给出不变式标注。

变量不变式也有强弱之分。

4．形状声明

类型不变式可用于描述结构体和共用体等类型的同类型变量都具有的不变性质。对于易变数据结构来说，其形状不变性不是由节点类型的类型不变式可以表达的，因为刻画形状特征涉及多个节点。安全 C 语言要求用形状声明来表达被标注的类型用于构造何种形状的易变数据结构。这部分放在第三章单独介绍。

```
// 这是尽量通用的二分查找。这时数组长度必须作为形参。
/*@
lemma property1: \forall int *b. \forall int value. \forall int k:[0..\length(b)-1].
        \length(b) > 0 && (\forall integer n:[0..\length(b)-2].b[n] < b[n+1]) && value > b[k]
    ==> (\forall integer n:[0..k].value > b[n]);
lemma property2: \forall int *b. \forall int value. \forall int k:[0..\length(b)-1].
        \length(b) > 0 && (\forall integer n:[0..\length(b)-2].b[n] < b[n+1]) && value < b[k]
    ==> (\forall integer n:[k..\length(b)-1].value < b[n]);
*/
#define MAX_LEN 10000
/*@ requires 0 < len <= MAX_LEN && \length(a) == len && (\forall integer n:[0.. len-2]. a[n] < a[n+1]);
    ensures 0 < len <= MAX_LEN && \length(a) == len && -1 <= \result && \result <= len-1 &&
        (\result >= 0 && a[\result] == val || \result == -1 && (\forall integer n:[0.. len-1]. a[n] != val));
*/
int bsearch(int* const a, int const len, int const val){
    int i, j, k;
    i = 0;   j = len-1;
    /*@ loop invariant
        0 < len <= MAX_LEN && \length(a) == len && 0 <= i <= len && -1 <= j <= len-1 &&
            (\forall integer n:[0..len-2]. a[n] < a[n+1]) &&
            (j-i >= -1 && (\forall integer n:[j+1..len-1]. val < a[n]) && (\forall integer n:[0..i-1]. val > a[n]) ||
        j-i == -2 && k == i-1 && val == a[k]);
        loop variant j - i + 1;
    */
    while(i <= j) {
        k = i + (j - i)/2;
        if(val <= a[k]) j = k - 1;
        if(val >= a[k]) i = k + 1;
    }
    if(j - i == -1) k = -1;
    return k;
}
```

图 2.2 二分查找函数的代码及标注

5．全局逻辑声明

C 语言的逻辑表达式不足以表达要验证的程序性质，从例 2.7 看到，全称量化逻辑表达式被用来表达数组的有序性。从例 2.3 看到，内建函数\length 和\offset 被用来表达数组和指针的性质。全称量化断言和内建函数\length 等都是 C 语言没有的，因此描述程序性质的语言必须强于 C 语言的逻辑表达式语言的表达能力。

全局逻辑声明，包括逻辑的常量、类型、变量、函数、谓词和引理等的定义或声明，用来扩展使用在标注中的逻辑表达式语言，使该语言的表达能力大大加强。

例 2.8 以操作平衡二叉树的程序的验证为例（见图 2.3），来对全局逻辑声明和定义建立一个初步印象。

在图 2.3 中，归纳谓词 Gt 和 Lt 用于定义二叉排序树的归纳谓词 BST。满足 balance 谓词的二叉排序树就是平衡二叉树。图 2.3 中的两个引理 LtProperty 和 GtProperty 只是验证中需要用到的引理的一部分。

有了这些谓词和逻辑变量声明，就可以给出图 1.1 的平衡二叉树的删除函数的函数协议，见图 2.4。为简单起见，协议中没有给出实参指针在函数返回点的特征。 □

```
typedef struct node{int data; int bf; struct node *l, *r;} Node;   // bf：节点的平衡因子
//@ shape l, r : tree;                                    // 形状声明，指针域 l 和 r 用于构造二叉树

bool taller, lower;                                       // 用来表示插入或删除节点后，树是否长高或变矮

/*@  inductive Gt(int x, Node* p) =                       // 归纳谓词定义：x 大于二叉树上所有节点的数据
        p == \null || p != \null && x > p->data && Gt(x, p->l) && Gt(x, p->r);
     inductive Lt(int x, Node* p) =                       // 归纳谓词定义：x 小于二叉树上所有节点的数据
        p == \null || p != \null && x < p->data && Lt(x, p->l) && Lt(x, p->r);
     inductive BST(Node* p) =                             // 二叉排序树的归纳谓词定义
        p == \null || p != \null && BST(p->l) && BST(p->r) && Gt(p->data, p->l) && Lt(p->data, p->r);
     inductive balance(Node* p, int m) =                  // 二叉树平衡性的归纳谓词定义，m 代表树的高度
        p == \null && m == 0 ||                           // bf 等于 1、0 和-1 分别代表左高、等高和右高
        p != \null && p->bf == 0 && balance(p->l, m-1) && balance(p->r, m-1) ||
        p != \null && p->l != \null && p->bf == 1 && balance(p->l, m-1) && balance(p->r, m-2) ||
        p != \null && p->r != \null && p->bf == -1 && balance(p->l, m-2) && balance(p->r, m-1);
     lemma LtProperty: \forall int x. \forall int y. \forall Node* p. x < y && Lt(y, p) ==> Lt(x, p);
     lemma GtProperty: \forall int x. \forall int y. \forall Node* p. x > y && Gt(y, p) ==> Gt(x, p);
     logic int m, n, k;                                   // 逻辑变量声明
*/
```

图 2.3 操作平衡二叉树的代码的验证所需要的部分全局逻辑声明

```
//@ logic Node* oldt;
/*@  requires BST(t) && balance(t, m) && Gt(n, t) && Lt(k, t) && oldt == t;
     assigns lower, *t;                                                    // 被修改的全局变量
     ensures BST(\result) && balance(\result, m) && Gt(n, \result) && Lt(k, \result) && !lower ||
             BST(\result) && balance(\result, m-1) && Gt(n, \result) && Lt(k, \result) && lower;
*/
Node * AVLdelete(Node *t, const int data) {... ...}
```

图 2.4 平衡二叉树节点删除函数的函数协议

需要对例 2.8 做进一步解释。没有 BST 和 balance 等归纳谓词定义，则难以写出平衡二叉树节点删除函数的协议。引理 LtProperty 和 GtProperty 是程序验证过程中需要用到的二叉排序树的归纳性质，若不提供这样的性质，基于演绎推理的验证器很难发现这样的归纳性质，从而导致验证的失败。

函数协议中出现的逻辑变量 m 在此的含义是：对于高度为 m 的平衡二叉树，调用该函数后，作为结果的平衡二叉树的高度仍然为 m，或者是 m - 1。逻辑变量 n 在此的含义是：对于任何大于平衡二叉树上所有节点的数据的 n，调用该函数后，n 仍然大于结果平衡二叉树上所有节点的数据。逻辑变量 k 的含义类似。逻辑变量仅可以使用在断言中，不能出现在程序代码中。

2.2.2 语句标注

语句标注有下面几种。

1．循环标注

循环标注有循环不变式和循环变式两种。图 2.2 二分查找函数中的 loop invariant 标注就是代码中循环的循环不变式，它表达该函数中 while (e) s 循环代码的不变性质。具体说，在

该循环的迭代计算过程中，每当到达计算 e 的程序点时，程序状态都满足循环不变式。

循环变式用于循环不会陷入无限次迭代执行循环体（可粗略地称为循环终止）的验证。循环变式 e 表示对正常终止或通过 continue 终止的每次迭代，在迭代结束时 e 的值必须比同次迭代开始时 e 的值要小。并且 e 的值在每次迭代开始时必须非负。循环变式是可选的，若不使用则表示不要求对循环进行这样的验证。

例 2.9 下面是循环变式的一个简单例子。

```
void f(int x) {
    //@ loop variant x;
    while (x >= 0) { x -= 2;}
}
```

□

2．程序点断言

对于每个函数，程序员必须提供函数协议，并为每个循环提供循环不变式。

程序员也可以在其他程序点给出断言。在某个程序点的断言 P 的含义是，P 在当前状态下必须成立。当前状态是指程序正好执行到该断言所在程序点的状态。程序点断言可标注在 C 语言的标号可以出现的任何地方，也可以正好在程序块的闭括号之前。

3．幽灵代码

先举例说明幽灵代码的应用。

例 2.10 图 2.5 给出有序单向链表的插入函数及其函数协议和循环不变式。先对其中的一些符号进行解释。

1．函数协议中的谓词\list(head)表示 head 指向标准的单向链表，内建函数\length(head, next)是指，从指针 head 指向的节点开始，顺着节点的 next 域一直走到 next 域等于 NULL 或者 next 域指向最初出发的节点为止，所经过的节点数，即链表的长度。循环不变式中的谓词\list_seg(head, ptr1)表示从 head 指向的节点到 ptr1 指向的节点构成一个表段。谓词\list(head)和\list_seg(head, ptr1)的定义见 3.1.2 节。

2. 函数协议和循环不变式中的 head->(next:i)->data 是 head->next…->next ->data 的缩写，其中“->next”重复 i 次。访问路径 head->(next:i)->data 称为带折迭域 next 的访问路径，其中的 i 是折迭表达式。循环不变式中的 ptr1 == head->(next:j-1)表示 head 指向的节点经过 j-1 个 next 指针到达 ptr1 指向的指针，也就是表段\list_seg(head, ptr1)共有 j 个节点。

3．oldhead 是逻辑变量，并且在函数前条件中有 oldhead == head。调用 listInsert 函数的指针实参 q 一定等于 oldhead，因为调用时实参 q 的值传给形参，即有 head == q，从而可得 oldhead == q。因为在函数中是不允许对逻辑变量 oldhead 赋值的，因此 oldhead 所起的作用就是一直指向实参指针 q 原来所指向的节点。

若形参 head 有 const 修饰符，则它在函数中不会被修改，这时逻辑变量 oldhead 没有什么用处，因为对应的实参 q 也不会被修改。当形参 head 没有 const 修饰符时，情况就不一样了。若 q 指向空链表或形参 data 小于等于第一个节点的 data 域，在这两种情况下，新节点都是作为链表的第一个节点。这时返回值指向链表的第 1 个节点，而实参 q 在第一种情况下等于 NULL，在第二种情况下指向链表的第 2 个节点。这就是逻辑变量 oldhead 的作用，把实参所指向节点和返回值所指向节点明显区别开来了。

```
#include <stdlib.h>
typedef struct node{struct node* next; int data;} Node;
//@ shape next: list;
//@ logic int m; logic Node* oldhead;
/*@  requires \list(head) && \length(head, next) == m && oldhead == head && 0 <= m <= 1000 &&
          (\forall int i:[1..m-1].head->(next:i-1)->data <= head->(next:i)->data);
assigns* oldhead;
exits \exit_status == 1;
ensures
     \list(\result) && \length(\result, next) == m+1 && oldhead == \result && 0 < m <= 1000 &&
        (\forall int i:[1..m].\result->(next:i-1)->data <= \result->(next:i)->data) || //插入节点不作为首节点
     \list(\result) && \length(\result, next) == m+1 && oldhead == \null && m == 0 &&
        (\forall int i:[1..m].\result->(next:i-1)->data <= \result->(next:i)->data) || //构成只有一个节点的表
     \list(\result) && \length(\result, next) == m+1 && oldhead ==\result->next && 0 < m <= 1000 &&
        (\forall int i:[1..m].\result->(next:i-1)->data <= \result->(next:i)->data);   //插入节点作为首节点
*/
Node* listInsert(Node* head, int data) {
     Node* ptr; Node* ptr1; Node* p;
     //@ ghost int j;
     p = (Node*)malloc(sizeof(Node)); if (p == NULL) {exit(1);}
     p->data = data; p->next = NULL;
     if (head == NULL) {
          head = p;
     } else if (p->data <= head->data) {
          p->next = head; head = p;      //  插在表头
     } else {
          ptr1 = head; ptr = head->next; //@ ghost j = 1;
     /*@ loop invariant   //分成插在表中和表尾两种情况。对链表分段描述，节点数据关系也分段描述。
               0 <= m <= 1000 && 1 <= j <= m && oldhead == head && \list_seg(head, ptr1) &&
               ptr1 == head->(next:j-1) && \list(p) && \length(p, next) == 1 &&
               ptr1->data <= p->data && p->data == data &&
               (\forall int i:[1..m-1]. head->(next:i-1)->data <= head->(next:i)->data) &&
               (      \list(ptr) && ptr != \null && ptr == ptr1->next && \length(ptr, next) == m-j &&
                        (\forall int i:[1..m-j-1].ptr->(next:i-1)->data <= ptr->(next:i)->data) ||
                    0 <= m <= 1000 && j == m && ptr1->next == \null && ptr == \null); */
          while( (ptr != NULL) && ( ptr->data < p->data ) ) {
               ptr1 = ptr; ptr = ptr->next;   //@ ghost j = j+1;
          }
          p->next = ptr1->next;   ptr1->next = p;
     }
     return head;
}
```

图 2.5 有序单向链表的插入函数

下面再看图 2.5 中的幽灵代码。标注 ghost int j 声明整型幽灵变量 j，标注 ghost j = 1 和 ghost j = j+1 都是幽灵赋值语句。从描述循环不变性质的角度看，变量 j 及相关语句是必要的，否则难以用全称量化断言来表达不变性（但可用归纳谓词方式）。变量 j 及相关语句并不影响函数的结果，它们又显得多余。把它们作为幽灵代码可以兼顾到双方的需要。 □

幽灵变量和幽灵语句类似于 C 的变量和语句，但它们仅可以出现在标注中，由关键字

ghost 引入。在程序验证过程中，幽灵代码被当成程序代码，而程序真正执行时是没有这些代码的。即幽灵代码只在程序验证时被关注。

显然，幽灵代码不能修改任何非幽灵变量。

从例 2.10 再次看到，当指针类型的变量作为实参时，虽然和其他类型的变量一样，它的值在被调用函数中不会被修改，但是它指向的节点在易变数据结构中的位置可能发生变化。若是删除节点的函数，它指向的对象有可能被释放。这就要求在函数的前条件中必须有逻辑变量等于指针形参的断言，以保证在函数后条件中可以体现实参指针的变化，以免造成隐患。

第 3 章 安全 C 语言的形状系统

验证操作易变数据结构的程序面临很多挑战。易变数据结构中严重的指针别名和指针指向对象的别名显著地复杂化了对操作这些数据结构的程序的分析和推理。

安全 C 语言在易变数据结构方面的设计完全根据参考文献[2]和[3]的形状图、形状图逻辑、形状系统和形状图理论的定理证明。先将它们简单概述如下。

形状是对易变数据结构的一种分类，这种分类反映易变数据结构节点之间的链接特点。例如双向链表和二叉树分别都是节点之间链接关系相同的一类易变数据结构的总称。形状图是形状的一种图形表示，形状图逻辑是基于形状图进行推理的一种逻辑，其推理规则都是用图形表示的。形状系统是一组规则，它指派形状属性给各种各样的易变数据结构。

为易变数据结构设计形状系统的目的是，限制程序中可以构造和使用的易变数据结构的种类，达到既能满足绝大多数实际软件编程的需要，又能禁止程序员随意构造难以分析和验证其性质的易变数据结构。通过这样的形状系统，提高堆指针操作程序的合法性门槛，控制住易变数据结构的复杂性，便于排除没有构造出（或操作在）程序员所声明形状上的程序，降低程序分析和程序验证的难度。

形状系统要求程序员声明结构体类型（或共用体类型等）用于构造何种形状的易变数据结构（形状声明已经在图 2.3 和图 2.5 中出现过），一个结构体类型不允许用于构造不同形状的实例。易变数据结构的形状声明可类比变量的类型声明，它有助于验证器发现程序在堆指针方面的错误。不同的是，形状声明比变量的类型声明复杂，并且形状声明以标注形式出现在程序注释中。

本章围绕单态命名基本形状和形状的分类，对形状系统进行初步介绍，让程序员对形状系统有大体的了解，并了解形状图在验证操作易变数据结构的代码上的优点。以方便学习标注语言和为代码写标注。

形状图之间的变换规则，作为 Hoare 一种扩展的形状图逻辑，以及形状图理论中的形状图等价和形状图蕴涵判定方法，它们与验证操作易变数据结构的代码的实现方法密切相关，而与程序员的关系不大，因此在本章不介绍或略微介绍。

3.1 单态命名基本形状

易变数据结构的形状分成单态基本形状、多态基本形状、嵌套的形状和含附加指针的形状等种类。基本形状可类比类型系统的基本类型，嵌套的形状和含附加指针的形状可类比从基本类型经类型构造子得到的构造类型。本节只介绍大家比较熟悉的单态基本形状中的单态命名基本形状，其余的在 3.2 节中介绍。只有了解了这些形状的严格定义及其性质和在标注中的声明方式，才能在标注中恰当地声明它们和描述它们的性质。

3.1.1 形状图

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。

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 3.1，图中给出了六种节点

的名称和语法。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 节点和悬空节点的含意稍后会提到。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。可理解为 e 和 a 分别是 m 和 $m \geq 0$。

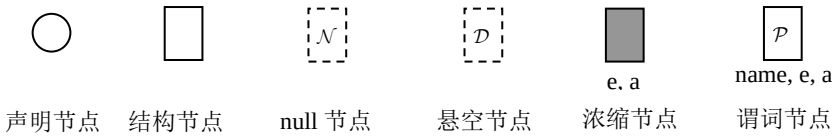

图 3.1 形状图的各种节点

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。

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。

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，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 $G_1, G_2, \ldots, G_n$ 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。其中，不含符号∧的形状图 G 被称为**形状子图**。

（3）若形状图 $G_1, G_2, \ldots, G_n$ 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。

从稍后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（略去在此不关心的非堆分配数据以及它们的指针，下同）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。下面若无特别说明，符号 G 仅表示不含符号∨的形状图。

受编程语言类型系统的约束，本手册涉及的形状图只是形状图定义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。

例 3.1 下面是遍历单向链表的程序片段，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;
}
```

图 3.2 是两个表示单向链表的形状图（在稍后介绍形状图的语义后会知道）。上述代码的循环不变特性由图 3.2(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，即两个浓缩节点的 e 分别是 m 和 n。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，由于 ptr != NULL，即 ptr 一定是有效指针，指向一个结构节点，而该结构节点的右边是浓缩节点数为 n（$n \geq 0$）节点，这个特点由图 3.2(b)概括。图 3.2(b)是由图 3.2(a)经形状图变换规则得到。 □

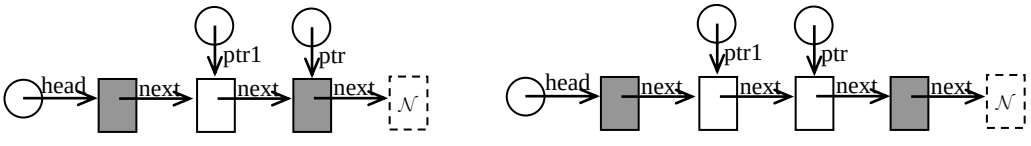

图 3.2　形状图的两个例子

验证器在验证了包含图 3.1 代码段的函数后，可以向程序员显示各程序点的形状图。例如，在循环语句入口点和循环体的入口点，若程序员需要，则可以分别看到图 3.2 的形状图(a)和(b)。这样的形状图有助于程序员排除代码中堆指针操作的错误，这是在本手册中介绍形状图的主要目的。

下面简单介绍形状图的语义。

对于像 C 这样有动态存储分配的语言，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，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。谓词节点的情况略微复杂一点，在 3.1.2 节再进一步介绍。

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。

（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：$s_d : DecVar \to AbsValue \cup \{\mathcal{N}, \mathcal{D}\}$ 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。下面用 s 或$\langle s_d, s_f \rangle$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。

形状图语义定义 1 形状图 G 所代表的机器状态集$\mathcal{S}[\![G]\!]$由下面几条规则定义。

（1）若 G 无浓缩节点和谓词节点，则$\mathcal{S}[\![G]\!]$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 $G_1,\ldots, G_k$，则$\mathcal{S}[\![G]\!] = \mathcal{S}[\![G_1]\!] \cup \ldots \cup \mathcal{S}[\![G_k]\!]$；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（$n \geq 0$）的形状图是 G_n，则$\mathcal{S}[\![G]\!] = \mathcal{S}[\![G_0]\!] \cup \mathcal{S}[\![G_1]\!] \cup \ldots$。

（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$\mathcal{S}[\![G]\!] = \mathcal{S}[\![G']\!]$或$\mathcal{S}[\![G]\!] = \mathcal{S}[\![G_1]\!] \cup \mathcal{S}[\![G_2]\!]$。

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 3.2(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。

形状图语义定义 2 形状图 G 所表示的断言集$\mathcal{A}[\![G]\!]$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 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 3.2(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。

3.1.2 单态命名基本形状的逻辑定义

上一小节提到，对于程序员来说，形状图的最大作用是有助于排除代码中的栈指针操作错误。程序员在代码中提供的标注，使用的是标注语言所规定的文字表达形式，不需要画形状图。本小节以单态命名基本形状为例，介绍在标注中如何声明形状和这些形状的内建逻辑定义。

单态命名基本形状有单向链表（list）、循环单向链表（c_list）、双向链表（dlist）、循环双向链表（c_dlist）、二叉树（tree）和数据块（data_block），一共 6 种。形状描述出现在相应的节点类型定义之后。

\length 是一个多态的内建函数，函数值一定是非负整数。在上述前 5 种形状中，\length(head, next)是指从指针 head 指向的节点开始，顺着节点的 next 域一直走到 next 域等于 NULL 或者指回最初出发的节点（后者构成环）的节点为止所经过的节点数。在例 2.10 中已经提到这一点。\length 也可用于后面介绍的各种形状中。

例 3.2 下面是在代码中的类型定义之后，为相应的类型定义标注上述前 5 种单态命名基本形状的例子。表示相应类型的变量用于构造哪种形状。

```
typedef struct node1 {int data; struct node1* next;} Node1;
//@ shape next: list;
```

```
typedef struct node2 {int data; struct node2* next;} Node2;
//@ shape next: c_list;
typedef struct node3{int data; struct node3* l; struct node3* r;}Node3;
//@ shape l, r : dlist;
typedef struct node4{int data; struct node4* l; struct node4* r;}Node4;
//@ shape l, r : c _dlist;
typedef struct node5{int data; struct node5* l; struct node5* r;}Node5;
//@ shape l, r : tree;
```

这些类型的变量的声明方式还是原来的那样。

数据块是由不含指针的单个节点构成的形状，它实际可以由某个类型的一个或多个元素构造。和例 3.2 的那 5 种单态基本形状不同，数据块可以因大小不固定而没有类型定义（见标注语言手册附录中的内存分配器例子）。在这种情况下，在声明数据块的指针变量时，需要在其声明之后加标注，以表明它们是指向动态分配的数据块。例如：

```
char *dp, *dq;
//@ shape dp, dq : data_block;
```

若形参、返回值或函数使用的全局变量是数据块指针，则标注加在参数表之后。例如：

```
char * f(char *p, long a, …) /*@ shape \result, p : data_block; shape q : data_block; */ {
    …
}
```

其中 q 是全局数据块指针。

同样，若仅是函数原型，形状标注仍然出现在参数表之后。例如：

```
char * f(char *p, long a, …)/*@ shape \result, p : data_block; shape q : data_block;*/;
```

数据块也可以通过如下形式的类型定义及标注来定义：

```
typedef int array [1000];
//@ shape : data_block;
```

有了这样的标注，那么这个类型的变量只能动态生成，而不允许通过静态声明获得。由于 array 类型已经描述为用于动态生成的数据块，因此其指针可以直接声明为

```
array *dp, *dq;
```

它们作为 data_block 形状的指针的标注可以省略。在这种情况下，作为数据块指针的形参、返回值和全局变量的描述方式类似，data_block 形状标注可以省略。

下面给出标注语言为单态命名基本形状提供的内建谓词和引理。对于不同的形状，Node 的类型可能不同，这里没有给出它们的定义，仅给出的结构体域指针采用本手册的习惯用的名字。

1．单向链表

（1）单向链表的谓词定义

- 完整链表

```
inductive \list(Node *p) = p == \null || \list(p->next);
```

- 表段

```
inductive \list_seg(Node *p, Node *q) =
    p == q && p != \null || p != q && \list_seg(p->next, q);
```

（2）单向链表的归纳引理

```
lemma p1: \forall Node *p, *q. \list_seg(p, q) && list(q) ==> \list(p);
lemma p2: \forall Node *p, *q. \list_seg(p, q) && q->next!= \null
        ==> \list_seg(p,q->next);
```

可以对比一下单向链表的上述归纳定义和用形状图表示的归纳定义，后者在图 3.3 中。两者本质上是一样的归纳谓词定义，分成空表和非空表两种情况。符号化的内建定义让程序员了解这些单态命名基本形状的准确定义，用形状图给出的定义用于演绎推理过程中形状图的等价变换和蕴涵变换等。

在图 3.2 中，指针 head 指向的节点到指针 ptr1 指向的节点部分，通常称为单向链表的一个表段。因此单向链表的符号断言中还有表段的归纳定义。如果指针 p == q，并且它们都有效指针，则它们共同指向的那个节点是仅有一个节点的表段，若 p 和 q 不相等，都有效，并且从 p 指向的节点到 q 指向的节点有一条路径，则它们指向的节点之间构成长度大于 1 的表段。对于例 3.1 的代码，程序员标注的循环不变式是：

m >= 0 && \list_seg(head, ptr1) && ptr1==head->(next:m+1) && ptr==ptr1->next &&
(n > 0 && ptr!=\null && list(ptr) && \length(ptr, next)==n || n==0 && ptr==\null)

系统依据上述循环不变式得到的是图 3.2(a)的循环不变形状图，再依据该循环不变形状图，在基于代码的演绎推理过程中逐步产生循环体中各程序点的形状图。

单向链表的两个归纳引理是显然的。

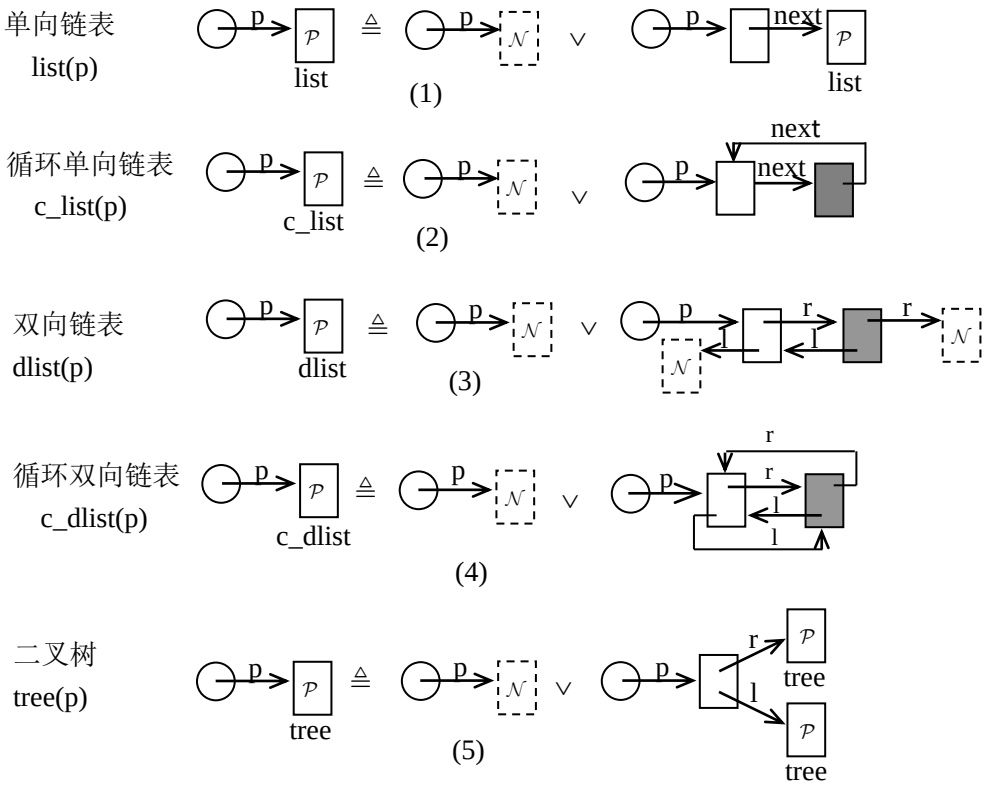

图 3.3 五种单态命名基本形状的谓词定义

对于其它单态命名基本形状的逻辑定义，不再对照图 3.3 的定义逐个解释。

2．循环单向链表

（1）循环单向链表的谓词定义

- 表段

```
inductive \c_list_seg(Node *p, Node *q) =
    p == q && p != \null || p != q && \c_list_seg(p->next, q);
```

- 完整链表

```
predicate \c_list(Node *p) =
    p == \null || \exists Node *q. \c_list_seg(p, q) && q->next == p;
```

（2）循环单向链表的归纳引理

```
lemma p1: \forall Node *p, *q.
    \c_list_seg(p, q) && q->next != \null && q->next != p ==> \c_list_seg(p, q->next);
lemma p2: \forall Node *p, *q.
    \c_list_seg(p, q) && \c_list_seg(q, p) ==> \c_list(p);
```

3．双向链表

（1）双向链表的谓词定义

根据先前的类型定义，双向链表节点的类型是 Node3，各节点有两个指针域 r 和 l。在给出定义前，我们需先考虑，在循环代码中遍历双向链表时，用哪个指针域来表达遍历的前进方向。还需要考虑在用量化断言或归纳谓词来描述相邻节点的数据关系（例如递增链表）时，让哪个指针域出现在这些节点的访问路径上。一般来说，这两件事选择的是同一个域，称其为 forward 域（详细解释见标注语言手册第 8 章）。在下面的定义中选择的是 r 域。

- 表段

```
inductive \dlist_seg(Node *p, Node *q) =
    p == q && p != \null || p != q && p == p->r->l && \dlist_seg(p->r, q);
```

- 完整链表

```
predicate \dlist(Node *p) =
    p == \null ||
    p->l == \null && \exists Node *q. \dlist_seg(p, q) && q->r == \null;
predicate \almost_dlist(Node *p) =
    \exists Node *q. \dlist_seg(p, q) && q->r == \null;
```

（2）双向链表的引理

```
lemma p1: \forall Node *p, *q.
    \dlist_seg(p, q) && q->r->l == q ==> \dlist_seg(p, q->r);
lemma p2: \forall Node *p, *q.
    \dlist_seg(p, q) && \almost_dlist(q) && p->l == \null ==> \dlist(p);
```

4．循环双向链表

（1）循环双向链表的谓词定义

这也是双向链表，同样需决定哪个指针域作为 forward 域，在此同样用 r 域。

- 表段

```
inductive \c_dlist_seg(Node *p, Node *q) =
    p == q && p != \null ||
        p != q && p == p->r->l && \c_dlist_seg(p->r, q);
```

- 完整链表

```
predicate \c_dlist(Node *p) =
    p == \null ||
    \exists Node *q. \c_dlist_seg(p, q) && p->l == q && q->r == p;
```

（2）循环双向链表的引理

```
lemma p1: \forall Node *p, *q.
    \c_dlist_seg(p, q) && q->r->l == q ==> \c_dlist_seg(p, q->r);
lemma p2: \forall Node *p, *q.
    \c_dlist_seg(p, q) && \c_dlist_seg(q, p) ==> \c_dlist(p);
```

5．二叉树

（1）二叉树的谓词定义

- 完整二叉树

```
inductive \tree(Node *p) = p == \null || \tree(p->l) && \tree(p->r);
```

- 树段

```
inductive \tree_seg_l(Node *p, Node *q) =
    p == q && p != \null || p != q && \tree_seg_l(p->l, q) && \tree(p->r);
inductive \tree_seg_r(Node *p, Node *q) =
    p == q && p != \null || p != q && \tree_seg_r(p->r, q) && \tree(p->l);
inductive \tree_seg_lr(Node *p, Node *q) =
    p == q && p != \null ||
    \tree_seg_lr(p->r, q) && \tree(p->l) ||
    \tree_seg_lr(p->l, q) && \tree(p->r);
```

内建谓词\tree_seg_l 和\tree_seg_r 名字中最后一个字母 l 和 r，分别指相应自引用结构体类型中用于构造二叉树的第一和第二个指针域（按它们在类型定义中出现的次序），不管类型定义中两个指针域使用什么名字，三个内建函数的名字不变。

对于二叉树，同样可以使用带折迭域的访问路径，例如 q = p->(r : 5)。并且还可以用 q = p->(l, r : 5)来表示 p 经过 5 个节点的 l 或 r 域指针到达 q。注意，其中的指针域名需要和相应结构体类型定义中的一致。

（2）二叉树的引理

```
lemma p1: \forall Node *p, *q.
    \tree_seg_r(p, q) && \tree(q->l) && q->r != \null
  ==> \tree_seg_r(p, q->r);
lemma p2: \forall Node *p, *q.
    \tree_seg_l(p, q) && \tree(q->r) && q->l != \null
  ==> \tree_seg_l(p, q->l);
lemma p3: \forall Node *p, *q.
    \tree_seg_lr(p, q) && \tree(q->l) && q->r != \null
  ==> \tree_seg_lr(p, q->r);
lemma p4: \forall Node *p, *q.
    \tree_seg_lr(p, q) && \tree(q->r) && q->l != \null
  ==> \tree_seg_lr(p, q->l);
lemma p5: \forall Node *p, *q.
    \tree_seg_r(p, q) && \tree(q) ==> \tree(p);
lemma p6: \forall Node *p, *q.
    \tree_seg_l(p, q) && \tree(q) ==> \tree(p);
lemma p7: \forall Node *p, *q.
    \tree_seg_lr(p, q) && \tree(q) ==> \tree(p);
```

6．数据块

数据块的谓词定义是

```
\data_block(char *p) = p == \null || p != \null
```

在断言中很少用，需要时直接使用 p == \null 或者 p != \null。

程序员在为操作单态命名基本类型的函数书写循环不变式和程序点断言时，使用这些谓词会给书写带来方便，使用这些谓词和引理会给相关验证条件的证明带来方便。

注意，上面所给出的定义没有考虑节点有数据域的情况。在验证操作单态命名基本类型的函数时，若验证的性质包括节点数据域的断言，则很可能要基于相应的基本形状的定义，

重新定义这种带数据域的基本形状。例如，二叉排序树的定义是（见例 2.8）

inductive BST(Node *p) = p == \null ||

p != \null && BST(p->l) && BST(p->r) && Gt(p->d, p->l) && Lt(p->d, p->r);

显然 BST(p) ==> \tree(p)。因此在书写循环不变式和程序点断言时，有了断言 BST(p)，就不必再写断言\tree(p)。有了断言 BST_seg_r(p, q)时，就不必再写断言\tree_seg_r(p, q)。带节点数据断言的谓词通常蕴涵这里相应的内建谓词，只要前者是在后者的基础上扩大的。

3.1.3 操作单态命名基本形状的代码的验证

3.1.1 节已经介绍了形状图的语义，本节还需要围绕语义问题再深入一步，才能全面了解使用形状图和形状图逻辑的优点。

对于二叉树的归纳定义：

inductive \tree(Node *p) = p == \null || \tree(p->l) && \tree(p->r);

除了所有的二叉树满足这个定义外，允许有多个父节点的二叉无环有向图也满足这个定义。上述定义把后者排除在外，是因为一个节点若有多个父节点，则从这些父节点出发的这多个指针相等，但是基于上述归纳定义，是推不出任何两个指针相等的。这是基于代数规范（*algebraic specification*）初始语义（*initial semantics*）的一个重要性质：基于定义能证明为相等的两个项则相等，否则它们就是不相等。

例 3.3 再举个简单例子来体会初始语义。下面是关于自然数的一个代数规范。

sorts: *nat*

fctns: $0 : nat \quad S : nat \to nat$

$+ : nat \times nat \to nat$；

eqns： [x : nat, y : nat]

$x + 0 = x$

$x + S(y) = S(x + y)$

其中的 *nat* 是类别的符号，0、*S* 和+是零元、一元和二元的运算符号，最后两行是它们满足的等式。若把上面这些符号分别指派到自然数类型，自然数 0，自然数上的后继运算和自然数上的加运算，那么自然数类型及其上的这些运算构成该代数规范的一个初始模型。初始模型唯一到同构，就是说，对于上面这个代数规范，若还有初始模型的话，它一定与上述自然数模型同构。集合{$0_{模5}$，$1_{模5}$，$2_{模5}$，$3_{模5}$，$4_{模5}$}以及模 5 的后继运算和模 5 的加运算，也构成该代数规范的一个模型，因为它满足 eqns 部分的两个等式。但是它不是初始模型，它是宽松语义（*loose semantics*）的一个模型，因为它还满足基于上面代数规范的等式推不出的等式，例如，$4_{模5} +_{模5} 1_{模5} = 0_{模5}$。若用宽松语义，它的模型不止一个同构类。 □

对于二叉树的归纳定义，不需要像例 3.3 那样考虑符号的映射，因为那儿的符号 Node 等都已有明确含义。但还是要决定，按初始语义还是宽松语义的方式来得到二叉树归纳定义确立的二叉树集合。由于演绎推理都是基于已经存在的性质进行推理，推不出的性质就不存在，这与初始语义的方式一致。所以二叉树的归纳定义确定的集合只包含节点数不同的二叉树，不包含有多个父节点的二叉无环有向图。

3.1.2 节中的单向链表和二叉树的归纳定义，本质上定义的是代数数据类型。以图 3.4 的单向链表为例，把该链表各节点的内容（a, b, c, d, e, f, g）抽出，构成一个表，见图 3.4。

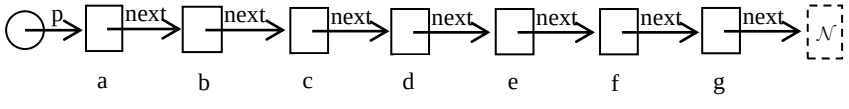

图 3.4 单向链表的示意图

一般来说，表类型 List 和表元类型 Atom 上至少有下列函数：

a, b, c, d, … : Atom;

Nil : List;

Cons : Atom × List → List;

Car : List → Atom; // 从表中取第一个表元。

Cdr : List → List; // 忽略第一个表元后，得到剩余元素构成的表。

在这些函数之间至少有下列等式（其中 x 和 l 分别是 Atom 类型和 List 类型的变量）：

Car(Cons(x, l)) = x;

Cdr(Cons(x, l)) = l;

把图 3.4 中整个链表各节点的 Atom 元素和略去第一个节点后的各节点的 Atom 元素，依次序排列成表，分别记为 L 和 L′，则 L = abcdefg，L′ = bcdefg，并有下列两个等式：

Car(L) = a 和 Cdr(L) = L′

所以，把易变数据结构单向链表的指针忽略，节点其余数据域的内容就是一种代数数据类型 List 的表元类型 Atom 的元素。C 语言不直接提供 List 类型，主要是因为内存管理上的困难。List 类型中不同长度的表需要的存储空间是不一样的，按最大需求为 List 类型的变量分配空间显然是不合适的。

对于双向链表、循环单向链表和循环双向链表，情况就没有这么简单。对于图 3.4 的单向链表，忽略各节点的指针域后，第 n+1 个节点的数据域领头形成的表是第 n 个节点的数据域领头形成的表的子表。再看图 3.5 的循环单向链表，因最后一个节点的 next 指针指向第一个节点而形成环。若还用忽略指针域而构造 List 表的方式，就会出现一种特殊要求：一个表的子表是这个表本身（由于形成环的指针的存在）。它就不属于代数数据类型的范畴了。

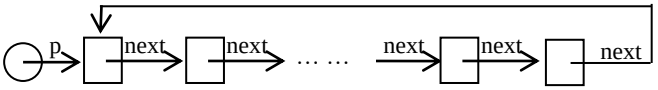

图 3.5 循环单向链表的示意图

双向链表虽然没有图 3.5 这样的大环，但是相邻节点之间相互指向对方的小环同样不可能把它抽象到某个代数数据类型。循环双向链表的情况更是如此。

演绎推理的程序验证所用的自动定理证明器，都是基于能证明为相等则相等，能证明为真的则为真。例如可满足性模理论求解器，其中的理论就是一些基于初始语义的代数数据类型的理论。

由于循环单向链表、双向链表和循环双向链表，都没有相应的基于初始语义的代数数据类型作为它们的模型，操作这些形状的代码很难验证。因此，到目前为止，无论是实验室研发的验证器原型，还是工业界使用的验证器，在介绍验证操作易变数据结构的程序的功能时，大都只涉及单向链表和二叉树。

本系统的验证器采用形状图和形状图理论来克服这个困难。在验证操作易变数据结构的函数时，验证条件生成器所产生的验证条件的一般形式是：

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

其中 $G_{i,j}$ 是形状图，$Q_{i,j}$ 是符号断言。在这里非形式地介绍时，用一种简单的情况

$$(G_1 \wedge Q_1) \Rightarrow (G_2 \wedge Q_2) \qquad (1)$$

进行解释。

验证条件（1）的证明分成两步，首先用形状图理论的定理证明方法证明

$$(G_1 \wedge Q_1) \Rightarrow G_2$$

该证明需用到 Q_1 中那些限定 G_1 中浓缩节点所代表的结构节点个数的符号断言。形状图理论的定理证明方法的可靠性也有模型支持的，该模型就是 3.1.1 节所介绍的形状图的语义模型。

然后的证明用 SMT 求解器来证明

$$(G_1' \wedge Q_1) \Rightarrow Q_2$$

注意，这里是 G_1'而不是 G_1。若用 G_1，例如，若 G_1 是双向链表的形状图，仍有双向链表不是代数数据类型的问题。这一步是验证节点数据断言的性质，要求这时 Q_1 和 Q_2 中的节点访问路径都只用 forward 指针域，这是可以做到的。因为只用 forward 指针域，则可以把 G_1 的另一个指针域忽略，简化成只有 forward 指针域的单向链表，单向链表是有对应的代数数据类型的。这个证明结果和$(G_1 \wedge Q_1) \Rightarrow Q_2$ 是一致的。

这样分两步进行证明的方式，解决了循环单向链表、双向链表和循环双向链表没有对应的代数数据类型问题。并且只给程序员带来很少限制，就是用 forward 指针域描述节点的访问路径，但并不限制在代码中使用另一个指针域（若存在另一个指针域的话）。

3.2 易变数据结构的形状分类

3.1 节介绍了单态命名基本形状，本节介绍形状分类中其他形状。先通过两个例子来观察对复杂易变数据结构的实际需求。它们都可以用本节所介绍的形状系统来描述。

例 3.4 图 3.6 是一个内存分配器所用的各种易变数据结构及相互关系的示意图，它们共同维护动态分配的若干数据块（*block*），每个数据块被分成若干小块（*chunk*）。图 3.6 左边的标识符是程序所用的各种堆块的数据类型，箭头边上的标识符 data 是域指针的名字。block 的域指针 data 指向的数据块被分成了若干小块，这些小块分别是数据块的不同部分。属于同一个数据块的各小块形成一个 chunk_list 单向链表，它们由薄片（*slice*）结构来维护。维护各数据块的薄片构成 slice_list 单向链表。因此，实际上图 3.6 中只有最下面一行的数据块存放数据，其余行的块都是维护这种结构的数据。指针的特别之处是，slice 与其维护的 block 相互有指向对方的指针，共享数据块的各 chunk 指向同一个 block。 □

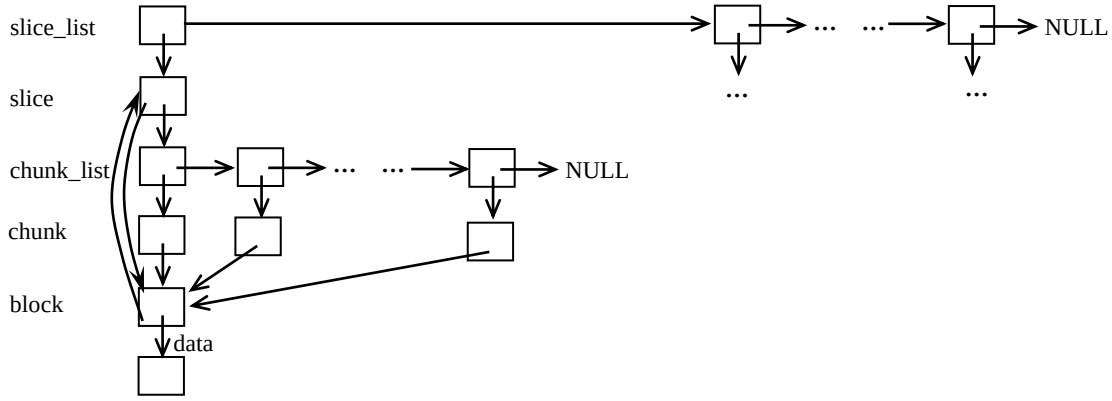

图 3.6 一个内存分配器所用数据结构

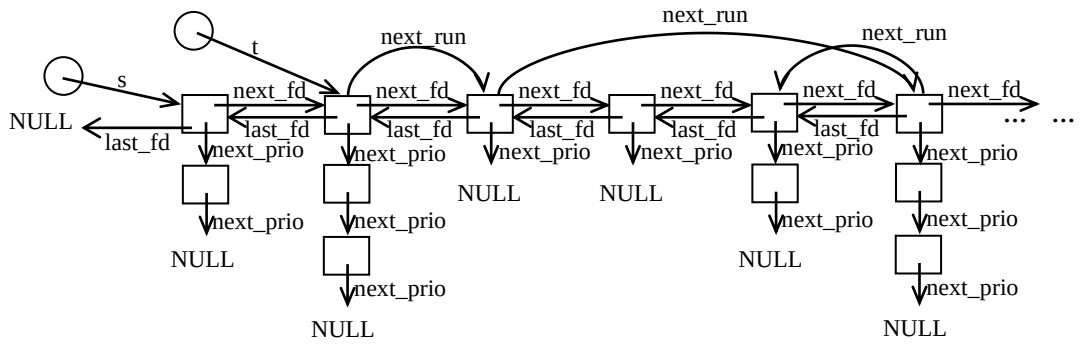

图 3.7 实现异步 I/O 操作所使用的数据结构的示意图

例 3.5 图 3.7 是 GNU C Library 中实现异步 I/O 操作所使用的易变数据结构的示意图。从 s 开始，由节点的 next_fd 和 last_fd 指针链接的双向链表（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）。 □

3.2.1 单态无名基本形状

3.1 节定义的 6 种单态命名基本形状并没有囊括编程中可能会用到的所有基本形状，即使再增加几种有时会用到的形状，也未必能囊括。还有，若某个结构体类型标注为用于构造双向链表，则该类型的结构体不允许用作二叉树的节点。但在编程中，不符合这个限定的情况是可能发生的，例如对于线索二叉树，其线索化函数就是把二叉树变为线索二叉树，两者的指针特性不完全一样。

为适应这样的需求，形状系统提供单态无名基本形状子类。单态无名基本形状的每个节点的指针不少于两个，表达节点之间相互指向关系的形状特性由程序员用标注提供。对这样的形状没有公认的形状名称。

例 3.6 以线索二叉树[4]为例来介绍如何标注和使用单态无名基本类型。

线索二叉树有如下几个特点：

（1）线索二叉树的叶节点没有 NULL 指针。因此不能把它看成是单态命名基本形状中的二叉树。

（2）在程序中，通常是先构造二叉树 *A*，然后把它线索化成线索二叉树 *B*，并且 *A* 和 *B* 的节点属于同一种结构体类型。即使在单态命名基本形状集中增加线索二叉树也不能支持从二叉树到线索二叉树的线索化。所以，线索二叉树需要用单态无名基本形状来描述。

（3）线索二叉树与二叉树的指针特性不完全一样。因此需要用多个谓词，而不能用类型不变式来定义线索二叉树的指针之间的关系。

线索二叉树节点的数据类型定义及其标注如下：

```
typedef enum pTag {Link, Thread};
typedef struct node {long data; struct node *l; struct node*r; pTag l_tag, r_tag;}Node;
//@ shape l, r : no_name, predicate;
```

该标注表示，结构体用来构造一种无名形状，由谓词来定义指针的有效性和指针之间的相等性。在给出相关谓词前，先给出线索二叉树的定义需要用到两个逻辑函数：求线索二叉树最右下节点和最左下节点的逻辑函数。

```
//@ logic Node* rightMost(Node* p) = p->r_tag == Thread ? p : rightMost(p->r);
//@ logic Node* leftMost(Node* p) = p->l_tag == Thread ? p : leftMost(p->l);
```

从头节点开始的线索二叉树（中序）的定义如下：

```
/*@ predicate hTree(Node* p) =
    p != \null && p->l == \null && p->l_tag == Thread &&
                                p->l == p && p->r_tag == Thread && p->r == p ||
    p != \null && p->l != \null && tTree(p->l, p, p) && p->r == rightMost(p->l);
*/
```

从根节点开始的非空线索二叉树的定义如下，谓词变元 pred 和 succ 分别是 p 所指向的线索二叉树的前驱节点指针和后继节点指针。

```
/*@ inductive tTree(Node* p, Node* pred, Node* succ) =
    (p->l_tag == Link && tTree(p->l, pred, p) ||
        p->l_tag == Thread && p->l == pred && p == leftMost(pred->r)) &&
```

```
(p->r_tag == Link && tTree(p->r, p, succ) ||
    p->r_tag == Thread && p->r == succ && p == rightMost(succ->l));
*/
```

tTree 体现出与单态命名基本形状一个有区别的特点：节点指针的指向依赖于节点其他数据域的值。

线索化之前的二叉树的定义如下：

```
/*@ inductive Tree(Node *p) = p == \null ||
      p! = \null && p->l_tag == Link && p->r_tag == Link &&Tree(p->l) && Tree(p->r);
*/
```

归纳谓词 Tree 用于线索化函数的前条件中，表示线索化之前的二叉树的特点。谓词 hTree（用到归纳谓词 tTree）用于线索化函数的后条件中，表示线索化的结果是线索二叉树。 □

B 树和 B+树等也可以描述为单态无名基本形状，只不过它的所有指针用一个数组来描述而不是分别起名字的域。

大多数单态基本形状的尾部节点或最下节点的指针等于 NULL，循环链表和线索二叉树是尾部或最下节点的指针不等于 NULL，循环单向和双向链表分别只有一个和两个尾部指针，它们逆向回指到循环链表的第一个节点。线索二叉树的逆向指针的多少与树的大小有关，并且一棵树的前驱和后继节点，不是从它的根节点沿着正向指针能够到达的。

对于单态无名基本形状和下面的多态无名基本形状，也只能用 malloc 函数每次产生一个节点。

3.2.2 多态基本形状

若易变数据结构的各个节点都属于同一种含共用体域或灵活数组域的结构体类型 t，t 中的指针域都属于 t 的指针类型，但是共用体域或灵活数组域的存在导致各节点的指针个数可能不一样，那么这样的易变数据结构归为多态基本形状类。多态基本形状也分成两个子类。

1．多态命名基本形状

只有数据块 1 种，它与单态数据块类似，区别是同一结构体或共用体类型的数据块可以有不同的数据域。

2．多态无名基本形状

节点所含指针域的域名和个数并非都相同的基本形状称为多态无名基本形状。在多态无名基本形状中，表达节点之间相互指向关系的指针特性由程序员用标注提供。这类多态基本形状没有公认的形状名称。

例 3.7 一种抽象语法树有二元运算节点、一元运算节点、常量节点和变量节点，它的类型定义与标注如图 3.8。

left、right 和 operand 指针虽然都在共用体层面，但仍然都看成是外层结构体的指针。polymorphic_no_name 表示这些指针用于构造多态无名基本形状的实例。描述形状特征的类型不变式体现出节点的指针值依赖于节点其他数据的值。 □

多态无名基本形状节点的类型定义有如下几点规定：

1．描述节点的结构体类型中必须有共用体类型的域或灵活数组域，并且其中含指向该结构体类型的指针。若使用共用体，则共用体的各个成员代表多态节点的不同形态；若使用灵活数组域，则大小不同的数组代表不同的形态。

2．若使用共用体，则该结构体类型中必须有唯一的枚举类型的标志域，枚举值的个数要和上述共用体的成员个数相同,并且按序对应。在本例中，nodeTag 是标志域，若节点的 nodeTag 等于 binary，则表明节点有 binOp、left 和 right 三个域。这种对应是节点类型自动

有的强类型不变性。

3．若使用灵活数组，则按照安全 C 语言的规定，该结构体类型必须有整型的标志域，用于指明该结构体类型的各具体结构体中该数组的实际大小，作为该类型的强类型不变式。

```
typedef enum nodeKind {binary, unary, constant, variable};
typedef enum binaryOpKind {…};
typedef enum unaryOpKind {…};
typedef struct node {
        nodeKind nodeTag;
        union {
                struct {binaryOpKind binOp;struct node *left, *right;}; //  二元运算
                struct {unaryOpKind unaryOp; struct node*operand;}; //  一元运算
                int number;          //  常量值
                int index;           //  变量在符号表中的下标
        };
}Node;
/*@  shape nodeKind: tag;   // nodeTag 是标志域
        shape left, right, operand: polymorphic_no_name, invariant;
        inductive nonempty_syntax_tree(Node *p) =
                (p->nodeTag == binary ==> syntax_tree(p->left) && syntax_tree(p->right) )&&
                (p->nodeTag == unary ==> syntax_tree(p->operand)) &&
                (p->nodeTag == constant ==> true) && (p->nodeTag == variable ==> true);
        type invariant syntax_tree(Node *p) = p == \null || nonempty_syntax_tree(Node *p);
*/
```

图 3.8 一种抽象语法树的类型定义和形状标注

例 3.8，图 3.9 是一种使用灵活数组的多态无名形状的节点类型声明和标注。其中有 strong 前缀的类型不变式是强类型不变式，其后的弱类型不变式刻画形状特征。 □

```
typedef struct node{int n; struct node* a[ ];}Node;
//@ shape a: polymorphic_no_name, invariant;
/*@  strong type invariant inv_Node(Node node) = \length(node.a) == node.n && 1<= node.n <= 3;
       type invariant some_tree( Node *p) = p == \null ||
              (p->n == 1 ==> some_tree(p->a[0])) &&
              (p->n == 2 ==> some_tree(p->a[0]) && some_tree(p->a[1])) &&
              (p->n == 3 ==> some_tree(p->a[0]) && some_tree(p->a[1]) && some_tree(p->a[2]));
*/
```

图 3.9 使用灵活数组的多态无名形状的节点类型声明和标注

单态和多态基本形状合称为基本形状，其中单态命名基本形状是系统预定义的。

3.2.3 嵌套形状

复杂形状分成嵌套形状、含内部附加指针的形状、含外来附加指针的形状和相同形状实例的序列四种，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实例的指针、各节点都有指向本形状节点的附加指针、在形状的特殊位置上的节点有来自形状外部的附加指针和同一种形状的若干实例构成的形状。先说嵌套形状。

一个形状上各节点都有指向独占或共享的内嵌形状实例的指针，就构成一种嵌套形状。在嵌套场合下，它们分别被称为主形状和次形状。若次形状还有内嵌形状，则形成多层嵌套的形状。

数据块也可以有内嵌形状实例的指针，这时把它称为过渡块比较合适。

例如图 3.6 是一个多层嵌套形状的实例。单向链表 slice_list 的每个节点都有指向各自过渡块 slice 的指针。每个 slice 有指向自己的单向链表 chunk_list 的指针。各 chunk_list 的每个节点指向各自的过渡块 chunk。属于同一个 chunk_list 的各个 chunk 指向同一个过渡块 block，体现出 chunk_list 的各个节点共享一个 block 块。每个 block 指向自己的数据块 data。

```
typedef struct requestqueue{
        struct requestqueue* next_prio;
}Requestqueue;
//@ shape next_prio : list;

typedef struct requestlist{
        struct requestlist* last_fd; struct requestlist* next_fd;
        Requestqueue* request_ptr;
}Requestlist;
/*@  shape last_fd, next_fd: dlist, primary;
        shape request_ptr : list, secondary;
*/
```

图 3.10 一种嵌套形状的类型声明和标注

注意，在图 3.6 中，过渡块 slice 和过渡块 block 除了有上一段提到的指向内嵌层的一个指针外，还有指向另一个内嵌层的指针，即图 3.6 上那一对相互指向对方的指针。

例 3.9 图 3.10 的两个递归结构体类型构成的形状是双向链表的每个节点都有一个指向各自附带的单向链表的指针，它源于图 3.7。 □

在图 3.6 的形状标注中，用于构造嵌套形状中主形状的指针的特征是 primary，而指向嵌套在其中的次形状的指针的特征是 secondary。

例 3.10 图 3.6 中，带两个第 2 层指针的过渡块 block 的类型定义和形状描述见图 3.11。

```
typedef struct _memory_block {
        unsigned int size, next, used;
        char* data;
        struct _memory_slice* slice;
} memory_block;
/*@  shape : data_block, primary;
        shape data: data_block, secondary;
        shape slice: data_block, secondary;
*/
```

图 3.11 带第 2 层指针的过渡块的类型声明和标注

注意，图 3.11 第 1 行标注中没有指针域名，因为这是过渡块，没有指向自身类型的指针。 □

3.2.4 含内部附加指针的形状

含内部附加指针的形状是指，各节点都有指向本形状节点的附加指针。内部附加指针仅限于用到单态有名基本形状（不包括数据块），其用途是让外来的、指向易变数据结构的指针能方便地调整其所指向的节点。例如，含父节点指针的二叉树，还有含两个逆向指针的左孩子右兄弟树等。

一种分类方法是把这些形状都归入基本形状集。其缺点是基本形状集变大，但仍可能没有囊括程序员使用附加指针的各种情况。另一种办法是，提供一种在现有基本形状上描述附加指针的方式，让程序员描述附加指针与构造基本形状的指针（简称基本指针）之间的关系。这种关系就是相应数据类型的一种弱类型不变式。根据这种关系的繁简不同，又可把附加指针分成三种。

1．确定的附加指针

若附加指针与基本指针之间的相等关系可以用简单指针断言（严格定义见标注语言手册）来描述，则称其为确定的附加指针。

例 3.11 以含父节点指针的二叉树为例，来展示如何描述确定的附加指针的性质，见图 3.12。含父节点指针的二叉树的基本形状是二叉树，附加指针 parent 指向父节点，这个指向可用弱类型不变式 pTree 定义，pTree 的定义要用到归纳谓词 p_Tree。pTree 和 p_Tree 构成

含父节点指针的二叉树的形状特征断言。这样的定义适合于用循环而不是递归代码来操作带父节点指针的二叉树。若用递归函数，图 3.12 的 pTree 和 p_Tree 定义需要调整。 □

```
typedef struct node{int data; struct node *l, *r, *parent;}Node;
/*@  shape l, r : tree, primary;      // 用于构造基本形状的指针
     shape parent: tree, invariant; // parent 作为附加指针
                                          // invariant 表示用类型不变式来表示 parent 指针的性质
     inductive p_Tree(Node *s) =
          (s->l == \null || s->l->parent == s && p_Tree(s->l)) &&
               (s->r == \null || s->r->parent == s && p_Tree(s->r));
     type invariant pTree(Node *s) = s == \null || s->parent == \null && p_Tree(s);
*/
```

图 3.12 含父节点指针的二叉树的父节点指针的性质的描述

2．部分确定的附加指针

若附加指针与基本指针之间的相等关系需要用量化指针断言（严格定义标注语言手册）来描述，则称其为部分确定的附加指针。

例 3.12 以两层跳表为例来展示如何描述部分确定的附加指针的性质，见图 3.13。

两层跳表的基本形状是单向链表，附加指针用于联系单向链表上的节点，它所指向的方向是明确的，但跨越多少个节点不确定。不确定源于下面归纳谓词 nonempty_skip_list 的定义中，存在量化断言约束变元 n 的不确定。

skip_list 谓词是该类型的弱类型不变式。 □

例 3.12 体现的一个特点是，顺着 skip 指针找下一个节点时，要经过基本形状上多少个节点是不确定的，但行走方向是确定的。而例 3.11 中顺着 parent 指针找父节点时，不仅行走方向确定，而且走过的节点数也是确定的。

```
typedef struct node{struct node *next, *skip;} Node;
/*@  shape next : list, primary;
     shape skip : list, invariant;
     inductive nonempty_skip_list(Node *s) =
          s->skip == \null && \exists integer n.
               (n >= 0 && (\forall integer i:[1..n].s->(next:i)->skip == \null)&& s->skip == s->(next:n+1)) ||
          s->skip != \null &&
               \exists integer n. (n >= 0 && (\forall integer i:[1..n].s->(next:i)->skip == \null) &&
               s->skip == s->(next:n+1) && nonempty_skip_list(s->(next:n+1)));
     type invariant skip_list(Node *s) = s == \null || s != \null && nonempty_skip_list(s);
*/
```

图 3.13 两层跳表的节点的类型定义和标注

对于含内部确定或部分确定附加指针的情况，在程序代码和程序断言中，访问路径的最后一个域名是附加指针域则代表附加指针，称为 a，否则是基本形状的指针，称为 p。注意，这一点与含内部不确定附加指针的情况有区别。

含内部确定或部分确定附加指针的编程约束有下面两点。

（1）禁止把附加指针 a 用于语句 a = (Node*)malloc(sizeof(Node))。

（2）禁止把附加指针 a 用于语句 free(a)。

这两点用于强化 a 作为附加指针的特点：a 用于方便访问数据结构的节点，但 a 不是数据结构的基本指针。

3．不确定的附加指针

若附加指针用来把基本形状上部分节点链接成附加单向链表，但节点的附加指针指向哪

个节点或者等于 NULL，单从基本形状看是完全没有规律的，则称该附加指针为不确定的附加指针。例如节点的附加指针的指向取决于节点上的数据，而各节点并非按这些数据有序排列。

在图 3.7 中，由 t 指向的、由节点的 next_run 指针构成的就绪队列就是一个附加单向链表。附加指针在此与情况 1 和 2 都不相同：行走方向和走过节点数都不确定。

例 3.13 以图 3.7 的基本形状为例，来展示如何描述不确定的附加指针的性质。图 3.14 是图 3.7 请求队列节点的类型定义，其中 next_run 是附加链指针。

```
typedef struct requestlist{
        bool running;      /*  节点是否在附加链上的标记  */
        struct requestlist *last_fd, *next_fd, *next_run;
        Requestqueue *request_ptr; /*  附加链表指针  */
}Requestlist;
/*@   shape running : tag;
        shape last_fd, next_fd: dlist, primary;
        shape request_ptr : list, secondary;
        shape next_run: list, additional;
*/
```

图 3.14 请求队列节点的类型定义和标注

附加链的声明指针必须专用。下面全局变量声明后的标注表示 runlist 作为附加链指针，它不能与基本链表的指针混用。

```
Requestlist * requests;
        /*  基本链表的指针  */
Requestlist * runlist;
        /*  附加链表的指针  */
/*@ shape runlist:
          additional_list;
*/
```

若形参 p1 是附加链表的指针，可以如下描述。

```
void add_request_to_runlist(Requestlist* p1, …)
                /*@ shape p1: additional_list; */ {
    … …
}
```

若函数形参和返回值都是附加链表的指针，可以如下描述。

```
Requestlist* remove(Requestlist* p, int fildes)
            /*@ shape p, \result: additional_list; */ {
                …   …
}
```

由于形状特征 additional 的含义是内部定义的，因此不需要用类型不变式或谓词来定义其特性。

主链表的节点在附加链表上，当且仅当其标志域 running 为 true。这种对应是节点类型自动有的强类型不变性，程序不能破坏这种不变性。 □

含内部不确定附加指针的编程约束有下面 3 点。

（1）基本形状的指针和附加链表指针应分别声明。以基本形状声明指针开始的指针型访问路径（称为基本形状指针，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指针型访问路径（称为附加链表指针，下面用 a 表示）中不允许出现基本形状的指针域。

（2）任何情况下不允许使用 p = a 语句、a = malloc(…)语句和 free(a)语句。

（3）在一个函数体中，仅允许以基本形状指针 p 或者仅允许以附加链表指针 a 为前缀对节点上的数据域（非指针域）进行访问。

3.2.5 含外来附加指针的形状

外来附加指针也仅用于单态有名基本形状（不包括数据块），各外来附加指针指向易变数据结构上某特殊节点，并且可以用仅基于附加指针和基本指针之间的等式来定义。不确定的外来附加指针的情况不予考虑。

例如队列本质上是单向链表，单向链表的指针 p 就是队列的尾指针，队列的头指针 q 对单向链表来说就是一个外来的附加指针。p 和 q 之间的关系式是队列的不变式，但不是单向链表节点类型的类型不变式。

例 3.14 以队列为例来展示如何描述含外来附加指针的形状的性质。

图 3.15 是含外来附加指针的队列的节点类型定义及随后标注和 struct queue 类型的弱类型不变式。注意，类型 Queue 无形状描述，因此只能通过静态声明来获得该类型的变量。□

```
typedef struct node {int data; struct node *next;} Node;
//@   shape next: list;
typedef struct queue{Node *front, *rear;}Queue;
/*@   type invariant queuePred (Queue q) = q.rear == \null && q.front == \null ||
                \exists integer n.(\length(q.front, next) == n && q.rear == q.front->(next:n-1));
*/
```

图 3.15 队列的类型定义和标注

3.2.6 相同形状实例的序列

指向同一类易变数据结构的一组指针，若各指针指向各自的易变数据结构，所形成的这一组实例称为相同形状实例的序列。例如，由一组单向链表指针及所指向的一组单向链表所构成的散列表。

由于这一组指针指向相同的形状，因此声明这组指针时，只要指明每个指针的形状特点即可（若从类型可判断则无需声明），例如：

```
typedef struct node {char* name; char* desr; struct node *next;} Node;
//@ shape next: list;
Node * hashTable[100];
```

表示数组 hashTable 的每个成员都有单向链表的指针。

数组限定为一维数组，数组元素的形状没有限制。

这样的堆指针的数组自动受弱类型不变式的约束：在使用堆数组的函数的入口和出口，其所有成员都指向相应易变数据结构的标准形状的实例，标准形状概念见 3.3 节。

3.2.7 单个节点的使用

形状系统的形状检查（见 3.3 节）规定，在一般情况下，若函数声明中有易变数据结构的形参指针，则要求相应实参指向其标准形状的实例。特殊情况是指参数指向单个节点并且这单个节点并不构成标准形状。这时需要在函数前条件中，用内建谓词\singleton（而不是形状标注）来表示形参的这个性质。

单个节点的指针是编程中的一种需要，单个节点及其指针并不被认为是一种形状。在函数体中，动态存储分配语句为某种形状分配一个新节点时，分配成功后的节点也是单个节点。

例 3.15 把例 2.10 中单向链表插入函数的第 2 个参数进行修改，由提供待插入的数据改成待插入的节点。插入函数只把这个节点的数据复制到自己生成的新节点上，然后把第 2

个形参指向的节点释放，则节点类型、函数协议和函数原型见图 3.16。函数协议中出现两种描述单个节点的谓词\singleton 和\dangling。

```
typedef struct node{struct node* next; int data;} Node;
//@ shape next: list;
/*@  logic int m; logic Node *oldhead;
      requires \list(head) && \length(head, next) == m && oldhead == head && 0 <= m <= 1000 &&
          \singleton(p) && (\forall int i:[0..m-1].oldhead->(next:i-1)->data <= head->(next:i)->data);
      exits \exit_status == 1;
      ensures \list(\result) && \length(head, next) == m+1 && oldhead == \result && 0 < m <= 1000 &&
              \dangling(p) && (\forall int i:[1..m]. \result->(next:i-1)->data <= \result->(next:i)->data) ||
          \list(\result) && \length(\result, next) == m+1 && oldhead == \null && m == 0 &&
              \dangling(p) && (\forall int i:[1..m]. \result->(next:i-1)->data <= \result->(next:i)->data) ||
          \list(\result) && \length(oldhead, next) == m+1 && oldhead== \result->next && 0<m<= 1000 &&
              \dangling(p) && (\forall int i:[1..m]. \result->(next:i-1)->data<= \result->(next:i)->data);
*/
Node* listInsert(Node* head, Node* const p);
```

图 3.16 单向链表插入函数的相关类型定义和标注

谓词\singleton(p)表示，不管 p 是什么易变数据结构的指针，p 所指向节点的每一个域指针都默认为悬空指针，除非另有断言指明这些域指针的性质。这样，\singleton(p)意味着在函数体内，p 所指向节点的指针，若尚未有断言表明其性质的，必须先赋值后引用。\dangling(p)表示 p 是悬空指针，声明堆指针在尚未赋初值时，都被认为是悬空指针。进一步的解释见标注语言手册。

3.3 形状推断和形状检查

易变数据结构在程序执行过程中有如下这些特点：

1．所含节点数会变化，也会引起长度（链表）或高度（二叉树）等属性的变化。这种变化源于易变数据结构的插入或删除等操作。

2．易变数据结构的形状在这些操作的过程中会被暂时破坏。即在某些程序点，一个指针及所指向易变数据结构并未构成所声明的形状。

3．把尚未成形的易变数据结构当作已成形的来操作，是出现无法推断别名的一个重要根源。例如：

（1）若未成形的单向链表如图 3.17 所示，最后一个节点的 next 指针是悬空指针。若程序以节点的 next 指针是否等于 NULL 来判断是否已遍历并操作了链表所有的节点，则会因对悬空指针解引用而引起无法推断的别名。

（2）若单向链表如图 3.18 所示，有另一个声明指针 q 指向链表的第 2 个节点。若以 p 为参数调用单向链表的节点删除函数，很可能因碰巧第 2 个节点被删除而导致 q 成为悬空指针，形成隐患。

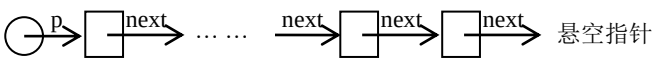

图 3.17 错误的单向链表

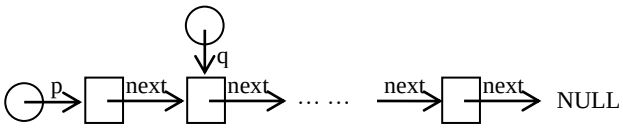

图 3.18 可接受的单向链表

显然，当单向链表作为实参时，宜保证只有指针指在头节点上，没有指针指在其他节点上，这是单向链表的标准形式。但是，在遍历单向链表的循环语

句的入口，有两个指针分别指在链表第 1 和第 2 个节点的情况还是有可能发生的，即图 3.18 在某些场合下又是可接受的形式。

易变数据结构的形状与变量的类型是有区别的。在任何程序点，变量的值必须属于它的类型，否则就是程序错误，因此对程序中有类型的程序构造的每个实例都要进行类型检查。易变数据结构的形状不同，在程序执行过程中，易变数据结构的形状经常遭到暂时破坏，因此形状检查宜在关键程序点进行。

形状检查就是检查指针所指向的易变数据结构是否形成了符合该指针声明所指示的形状。分析和推理指针指向的易变数据结构构成什么样形状的过程称为形状推断。形状推断是形状检查的先行步骤。

3.3.1 形状推断

先以单向链表为例，非形式地描述标准形状、可接受的形状和待完善的形状。即把易变数据结构在构造过程中出现的各种形态分类加以称呼。

1． 标准形状

图 3.19 是单向链表的标准形状，指针 head 等于 NULL，或者 head 指在单向链表的第一个节点上，链表的节点数没有限制，但没有其他外来指针指在该链表的节点上，最后一个节点的 next 指针等于 NULL。这也是单向链表的定义式能推导出的信息。

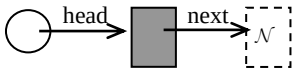

图 3.19 单项链表的标准形状

2．可接受的形状

图 3.18 是单向链表的可接受的形状。与标准形状的区别是，在标准形状的基础上，增加了若干指向该链表节点的其他外来指针。这些多出的指针对遍历形状各节点等操作来说是必要的。这些外来指针可能把整个链表分成若干段，循环不变式中有关形状部分的描述可以用整体描述或分段描述的方式（见标注语言手册）。

3．待完善的形状

若单向链表的指针 p 所指向的既不是标准形状也不是可接受的形状，则 p 所指向的就是待完善的形状。\singleton(p)和\dangling(p)的 p 所指向的形状是某些场合下我们给予特别关注的待完善形状。

对其他易变数据结构的形状，可以概述如下。

1．系统预定义的其他单态命名基本形状的标准形状、可接受形状和待完善的形状可以类似地区分。

2．对于其他基本形状，也可基于其定义和上面的标准加以区分。

3．对于嵌套形状，主结构是标准形状，主结构各节点所指向的独占或共享次结构也是标准形状，则该嵌套数据结构是标准形状。主结构是可接受形状，主结构各节点所指向的独占或共享次结构是标准形状（在下面两种情况可放宽到可接受形状），则该嵌套数据结构是可接受的形状。所放宽的两种情况是：

（1）次形状是共享的。

（2）对于非共享的次形状，凡是有外来指针指向的主结构节点，其次结构可以是可接受形状。

其他情况属于待完善的形状。

若当前代码仅操作某个次结构，则把该次结构当主结构来考察。

4．对于各种含附加指针的形状，若除各节点的附加指针外，构成的是标准形状或可接受形状，并且各节点的附加指针满足相应标注对其的约束，则整个形状是标准形状或可接受

形状（标准与可接收的区别与先前的一致），否则是待完善的形状。

5．对于相同形状实例的序列，若这一组形状都是标准形状，则该序列是标准形状或可接收形状（标准与可接收的区别与先前的一致），否则就是待完善的形状。

形状推断就是依据上面给出的标准，推断易变数据结构的指针所指向的易变数据结构是标准形状、可接受的形状还是待完善的形状。

3.3.2 形状检查

形状检查利用形状推断来判断指针所指向的易变数据结构的形状是否符合形状检查规则的规定。形状检查规则规定在哪些程序点、对哪些声明指针进行何种等级的检查。目前规定必须进行形状检查的有函数入口点、函数出口点、循环语句入口点、循环语句的出口点和函数调用点，检查等级分成标准形状、可接受形状和待完善形状的检查。为待完善形状进行的是宽松检查，但并非指任意待完善形状在任意程序点都可通过检查，它也有一些限制。下面按这五种程序点逐个介绍。

1．函数入口点

依据函数前条件中涉及堆指针形参 p 和堆指针全局变量 t 的符号断言，构造函数入口点的形状图。所构造的各形状图应该属于下列几种情况。

（1）p 和 t 分别指向各自的标准形状。

（2）若 p 和 t 或不同的 p 或不同的 t 指在同一个形状上，则该形状必须是可接受的形状。

（3）p 和 t 是单节点指针。这是函数入口所允许的待完善形状。

（4）p 和 t 是悬空指针。这是函数入口所允许的待完善形状。

2．函数出口点

返回值中可能有易变数据结构的指针，以 s 表示一个这样的指针。

依据函数后条件中涉及堆指针形参 p、堆指针全局变量 t 和返回值中的堆指针 s 的符号断言，构造函数出口点的形状图。所构造的形状图也是有 4 种可能，与函数入口点的一致，只是在此需要把指针 s 考虑进去。

3．循环语句的入口点

依据循环不变式中涉及堆指针的符号断言，构造作为循环不变式一部分的形状图。

（1）在循环体中被修改的声明堆指针，若并没有在循环体中最终被修改成 NULL 指针或悬空指针，则它们必须指向可接受的形状或者它们是悬空指针。

（2）在循环体中没有被修改的声明堆指针，它们可以指向任意形状。

4．循环语句的出口点

在循环语句的出口点，所要进行的形状检查与循环入口点的一致。

也可以改在循环体的结束点（而不是循环语句的出口点），进行和循环入口同样的形状检查。

5．函数调用点

对函数形参表中的每个易变数据结构的指针 p 对应的实参 p′：

（1）若函数前条件中有\singleton(p)，则 p′必须指向单个节点，即该节点上的指针都是悬空指针，除非函数前条件中有断言另行描述。

（2）若函数前条件中有\dangling(p)，情况类似（1）。

（3）若函数前条件中有指针相等断言把 p 与其他堆指针形参或全局堆指针 t 联系起来，则 p′指向的必须是有同样特点的可接受形状。

（4）其他情况下的 p′指向标准形状。

对于其它程序点，若程序员有检查请求时，则进行形状检查。若发现问题时，输出形状图作为警告信息。

3.3.3 形状系统给程序验证带来的好处

形状系统和类型系统看上去相似，其实它们有本质区别。

- 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，对类型系统而言，某个类型的变量只能保存该类型的值。对形状系统而言，指针并非时时刻刻都指向标准形状。
- 最后，类型系统的定型规则一般是结构化的，即根据某语言构造的各子构造的类型来确定该语言构造的类型，而在形状系统中，形状推断和形状检查是根据各节点的链接方式来推断所构成的形状以及该形状是否符合要求，它不是结构化的。

使用形状系统有如下好处：

1．形状系统限制程序中可以构造和使用的易变数据结构的种类，避免了程序员任意构造易变数据结构给程序验证带来的困难。另一方面，形状系统要求程序员声明所构造的易变数据结构的形状，它可免去程序验证为发现易变数据结构的形状所需做的分析推理工作。

2．形状系统对函数入口点、函数出口点、循环语句入口点和循环语句出口点的易变数据结构指针的限制，使得自动推断程序中有关易变数据结构的循环不变式（即循环不变形状图，限于操作单态命名基本形状和它们的嵌套形状的程序）成为可能，因而可以简化或免去程序员提供有关易变数据结构指针的循环不变式。即使不自动推断循环不变形状图，形状系统在这些程序点的限制也有助于及早发现循环不变式中有关易变数据结构指针的一些错误。

3．对自动推断循环不变形状图而言，形状检查有利于及时发现指针所指向的易变数据结构偏离所声明形状的情况，以保证推断过程的终止。

4．形状系统，还有形状图逻辑及其理论，使得自动验证操作易变数据结构的程序成为可能。

一些易变数据结构，例如双向链表和带环的各种链表，它们不是代数数据类型。对于操作这些易变数据结构的程序，Z3 难以证明其中一些复杂的验证条件。用形状图表示指针关系，用形状图理论及其定理证明方法判定形状图之间是否蕴涵，用形状图逻辑来进行有关堆指针操作语句的产生验证条件的演算，可以解决其中的困难。

参考文献

[1] 陈意云，张昱。编译原理（第三版），高等教育出版社，2014.9。

[2] 李兆鹏、张昱、陈意云，A Shape Graph Logic and a Shape System. *Journal of Computer Science and Technology*. 28(6):1063-1084, 2013.12. 见 http://staff.ustc.edu.cn/~yiyun/。

[3] 张昱、陈意云、李兆鹏，形状图逻辑的定理证明，计算机学报：39(12):2460-2480, 2016.12. 见 http://staff.ustc.edu.cn/ ~yiyun/。

[4] 严蔚敏，吴伟民。数据结构基础（C 语言版）（第 2 版），清华大学出版社，2007。