

安全 C 的规范语言 SCSL 使用手册

(版本 V1.1)

安徽中科国创高可信软件有限公司

Copyright © 2017-2021 版权所有

前言

本手册是安全 C 语言的规范语言 SCSL (*Safe C Specification Language*) 的参考手册, 用于安全 C 语言程序功能的形式化描述。该手册供熟悉 ISO C 编程语言, 了解安全 C 语言 [8], 并准备使用安全 C 语言程序的自动验证工具--科创验证器进行程序验证的软件技术人员参考学习。

SCSL 是安全 C 语言的行为接口规范语言 (*behavioral interface specification languages*), 可用于编写程序标注。程序标注加在程序注释中, 用来描述源代码的行为性质, 因此 SCSL 简称为标注语言。该语言的设计参考了 ACSL 语言 (*ANSI/ISO C Specification Language*) [1], 也体现出了中国科学技术大学计算机学院相关实验室的研究成果 [2, 3, 5]。

源文件中很多地方都有标注, 这些标注是对相关程序片段行为的描述, 它们综合起来就构成程序行为的描述。从宏观上看, 标注分成两大部分: 全局标注和语句标注。

全局标注主要有下面几种。

1. 函数协议 描述函数的行为。
2. 类型不变式 描述结构体和共用体等类型的不变性。
3. 变量不变式, 描述外部变量, 静态外部变量和静态局部变量的不变性。
4. 全局逻辑声明 逻辑的常量、类型、变量、函数、谓词及引理等的定义或声明。

语句标注有下面几种。

1. 程序点断言 描述该程序点状态必须满足的断言。
2. 循环标注 循环标注有循环不变式和循环变式, 分别描述循环的不变行为和循环结束的判定条件。
3. 幽灵代码 幽灵代码是仅能出现在标注中的 C 语言代码, 可以是幽灵变量声明、结构体的幽灵域声明、幽灵赋值语句、幽灵条件语句和幽灵循环语句等。幽灵语句只能修改幽灵变量, 不能修改 C 程序的变量和对 C 程序的代码产生任何影响。幽灵变量和幽灵语句的引入是为了书写标注的方便。

与 ACSL 和其他标注语言不同, SCSL 特有的部分是对易变数据结构的形状标注。它是依据形状图逻辑和形状系统 [2], 形状图理论的定理证明 [5], 以及上述全局标注和语句标注等的设计而设计出来的。在程序验证中, 形状声明标注有助于程序员在函数前后断言、循环不变式和程序点断言中描述单态和多态命名基本形状中各节点的指针特性, 还有助于程序验证系统发现程序中的指针错误。SCSL 所提供的易变数据结构的形状定义、它们的归纳引理和非归纳引理等的描述方式, 方便了操作易变数据结构的程序的形式化描述和验证。

※重要备注: 在《安全 C 语言使用手册》[8] 第三章中介绍的易变数据结构设计的形状系统, 目前仅实现了 3.1 节介绍的单态命名基本形状。因此本手册中有关其它形状的描述和例子仅作为参考, 后续的实现可能还会修改。

目 录

前 言	2
目 录	3
第 1 章 引言	5
1.1 标注概述	5
1.2 标注的语法设计	6
第 2 章 词法规则	7
第 3 章 逻辑表达式	8
3.1 算符优先级	10
3.2 语义	11
3.3 定型	11
3.4 整数算术和机器整数	12
3.5 实数和浮点数	13
3.5.1 实数与浮点数的联系与区别	13
3.5.2 对浮点计算程序的验证	15
3.6 数组和指针	17
3.7 结构体、共用体和数组	19
3.8 字符串	19
第 4 章 函数协议	24
4.1 内建构造 <code>\result</code> 和 <code>\old</code>	24
4.2 简化的函数协议	26
4.3 带命名行为的函数协议	30
4.4 异常终止	34
4.5 多协议	35
4.6 缺省协议和函数指针变量的协议	36
4.7 <code>main</code> 函数的前条件	36
4.8 对形参或返回值有 <code>void*</code> 或 <code>size_t</code> 类型的函数协议的特别要求	37
第 5 章 语句标注	40
5.1 程序点断言	40
5.1.1 <code>assert</code> 断言	40
5.1.2 <code>check</code> 断言	41
5.2 循环标注	41
5.2.1 循环不变式	41
5.2.2 循环变式	44
第 6 章 终止性	45
6.1 整数度量	45
6.2 一般度量	46
6.3 递归函数调用	47
6.4 不终止的函数	47
第 7 章 全局逻辑定义和声明	49
7.1 逻辑定义和声明的作用域	49
7.2 逻辑常量和逻辑变量	50

7.3 谓词定义和函数定义.....	55
7.4 引理声明.....	57
7.5 高阶逻辑构造.....	62
7.6 宏定义.....	64
7.7 常用逻辑定义和引理组成的逻辑模块及其使用.....	64
第 8 章 数据不变式和形状标注.....	66
8.1 类型不变式.....	66
8.2 变量不变式.....	67
8.3 易变数据结构的形状标注.....	68
8.3.1 描述形状性质的指针断言的种类及基本限制.....	69
8.3.2 各种易变数据结构的形状标注.....	71
8.3.3 对函数协议和循环不变式等的指针断言和节点数据断言的限制.....	77
8.3.4 形状标注和形状特征断言小结.....	82
第 9 章 幽灵变量和语句.....	84
9.1 幽灵代码的语法.....	84
9.2 对幽灵代码的限制.....	85
9.3 幽灵变量和逻辑变量的区别.....	90
附录一：供导入的逻辑模块.....	92
参考文献.....	97

第 1 章 引言

本篇是安全 C 的规范语言 SCSL (*Safe C Specification Language*) 的参考手册。SCSL 是安全 C 语言的行为接口规范语言 (*behavioral interface specification languages*), 可用于规范安全 C 源代码的行为性质。该语言的设计参考了 ACSL 语言 (*ANSI/ISO C Specification Language*) [1], 也包含了中国科学技术大学计算机学院相关实验室的研究成果[2, 3]。

本文档认为读者熟悉 ISO C 编程语言, 了解安全 C 语言[8]。

1.1 标注概述

程序的规范是以标注的形式直接写在 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。这类注释中的其余部分都被认为是标注。源文件中很多地方都有标注, 这些标注是对相关程序片段行为的描述, 它们综合起来就构成程序行为的描述。因此规范语言也称为标注语言。

全局逻辑声明和定义, 例如谓词定义和逻辑函数定义等, 其作用域是所在文件。若需要被多个文件共享, 则可用文件包含的方式, 让它们出现在这些文件都包含的头文件中。

标注分成全局标注和语句标注两类。其中全局标注有下面几种。

1. 函数协议 包含函数协议的标注正好插在函数声明或定义之前。

2. 类型不变式 类型不变式可用于描述结构体和共用体等类型的不变式, 其标注正好出现在被描述对象的定义或声明之后。

3. 变量不变式 变量不变式可用于全局变量, 也可用于静态外部变量和静态局部变量。例如一个整型静态局部变量 `b`, 在其生存期内始终不会小于 0, 则可以为它标注不变式 `b >= 0`。变量不变式的标注正好出现在变量声明之后。

4. 形状声明 形状声明用来表达自引用结构体类型 (也可能是其他类型) 用来构造哪类形状的易变数据结构。形状声明的标注正好出现在相应的类型声明之后。

5. 全局逻辑声明 逻辑的常量、类型、变量、函数、谓词及引理等的定义或声明可以标注在允许全局声明出现的地方。有些 (如引理) 也可以出现在允许局部声明出现的地方。

语句标注有下面几种。

1. 程序点断言 程序点断言可标注在 C 标号可以出现的任何地方, 或者正好在程序块的闭括号之前。

2. 循环标注 循环标注有不变式和变式, 它们可以出现在 `for`、`while` 和 `do...while` 循环语句之前。

3. 幽灵代码 幽灵代码是正规的 C 代码, 可以有幽灵变量声明、结构体的幽灵域声明和幽灵赋值语句等, 它们标注在变量声明、域声明和赋值语句等可以出现的地方。幽灵代码仅能出现在标注中, 并且它们只能修改幽灵变量。

标注中出现的名字也遵循先声明后使用的原则, 它们声明在 C 代码或者标注中。

若函数协议中出现程序变量 `m`, 该函数的形参中有 `m` 的声明, 另外该函数可访问的外部变量中也有 `m`, 则协议中的 `m` 被规定为指形参 `m`。

对于验证系统默认会验证并报错的性质, 例如数值溢出其类型规定的取值范围、对数据区的访问越界、除数为 0、访问悬空指针指向的对象和动态分配的堆数据区出现内存泄漏等, 程序员不必在程序标注中关注它们。

1.2 标注的语法设计

标注的文法必须仔细设计，以避免标注和代码相互影响。例如，在如下的代码中

```
if (c) //@ assert P;  
    c = 1;
```

语句 `c = 1` 必须理解为 `if` 语句的分支。

规范语言中的关键字，用于项和断言的那部分，都由反斜线开始，例如 `\forall`、`\exists` 和 `\result`，以免与程序中标识符混淆。其余的则无须用反斜线开始，例如 `ensures` 和 `requires`。它们都不受妨碍，可继续作为 C 程序的标识符。

对标注的分析可以分做两步，首先把标注从程序中剥离出来，然后再对其进行语法分析。

在本手册中，语法规则本质上用 BNF (Backus-Naur Form, 巴科斯-诺尔范式)。在语法规则中，使用额外的记号 e^* 表示 e 的零次、1 次或多次出现， e^+ 表示 e 的 1 次或多次出现， $e^?$ 表示 e 的零次或 1 次出现。为简单起见，一般的标注嵌在 `/*@ ... */` 风格的注释中，只有 1 行的标注 (单行标注) 可嵌在 `//@ ...` 风格的注释中。幽灵变量和幽灵语句的标注嵌在 `/*@ ghost ... */`，或 `//@ ghost ...` 风格的注释中。

第 2 章 词法规则

词法结构绝大部分遵从 ANSI C。少数区别必须注意。

1. 在注释 `/*@...*/` 和 `//@...` 中这两个位置的 `@` 符号仅表示本注释是标注。
2. 标识符可以由反斜线字符开始，但是仅限于内建的谓词和函数等。
3. 某些 UTF8 字符可用来代替某些逻辑或算术算符，它们列在表 2.1 中。（暂未实现）

表 2.1 可用 UTF8 字符代替的算符

逻辑或算术算符	UTF8 字符	UTF8 字符对应的编码
\geq	≥	0x2265
\leq	≤	0x2264
\Rightarrow	⇒	0x21D2
\Leftrightarrow	↔	0x21D4
$\&\&$	∧	0x2227
\parallel	∨	0x2228
$\wedge\wedge$	⊕	0x22BB
!	¬	0x00AC
- (unary minus)	-	0x2212
<code>\forall</code>	∀	0x2200
<code>\exists</code>	∃	0x2203

4. 注释也可以出现在 SCSL 的标注中，它们只能使用以 `//` 开始到当前行结束的形式。

第 3 章 逻辑表达式

本节展示在标注中用的表达式，称为逻辑表达式。图 3.1 和图 3.2 给出逻辑表达式基本构造的文法。

<i>term</i>	→ <i>literal</i> <i>id</i> <i>unary-op term</i> <i>term bin-op term</i> <i>term [term]</i> <i>term [term .. term]</i> <i>term [term .. \infinity]</i> <i>term . id</i> <i>term -> id</i> <i>term -> (id (, id) * : term)</i> <i>(type-exp) term</i> <i>id (term (, term) *)</i> <i>(term)</i> <i>term ? term : term</i> <i>\length (term , id)</i> <i>\length (term)</i> <i>\dimension(term , term)</i> <i>\offset (term)</i> <i>\string (term , term)</i> <i>\strlen (term)</i> <i>\prefix (term , term)</i> <i>\suffix (term , term)</i> <i>\index(term , term)</i> <i>\char(term , term)</i> <i>\result</i> <i>\lambda binders . term</i> <i>\old (term)</i> <i>ext-quantifier (term , term , term)</i>	//直接常量 //变量 //一元项 //二元项 //数组元素访问 //数组区间访问 //\infinity 的含义见第 9 章 //结构体域访问 //指向对象的域访问 //带折迭域的指向对象的域访问 //类型强制 //函数应用 //加括号的项 //三元的条件项 //链表长度函数 //一维数组长度和指针指向内存块长度函数 //多维数组中，求各维长度的函数 //指针指向位置在所属内存块中的偏移 //取字符数组某个长度的逻辑字符串函数 //计算物理字符串的长度 //字符串前缀函数 //字符串后缀函数 //字符串在另一个字符串中首次出现位置 //取字符串中某个位置的字符 //函数的结果 //抽象 //term 在函数入口点的值 //新增的
<i>literal</i>	→ <i>\true \false</i> <i>\null</i> <i>integer</i> <i>real</i> <i>string</i> <i>character</i>	//布尔常量 // NULL 指针常量 //整型常量 //实型常量 //串常量 //字符常量
<i>bin-op</i>	→ <i>+ - * / % << >></i> <i>== != <= >= > <</i> <i>&& ^</i> <i>& ^</i>	//算术运算符 //关系运算符 //布尔运算符 //按位运算符
<i>unary-op</i>	→ <i>+ -</i> <i>!</i> <i>~</i> <i>*</i> <i>&</i>	//一元加减符 //布尔否定符 //按位取反符 //指针脱引用符 //地址运算符
<i>ext-quantifier</i>	→ <i>\max \min \sum \product \numof</i>	//新增的范域词

图 3.1 项的文法

在此文法中，逻辑表达式分为项和断言，项对应到一阶逻辑的项，断言对应到一阶逻辑

的命题和谓词，并遵循它们在一阶逻辑中的通常区别。项对应到 C 语言中无副作用的表达式，另外，项还增加了一些算符和构造，其中由“\”开始的标识符都是系统内建的。

逻辑表达式是关于程序变量的性质断言，因此程序变量可以出现在逻辑表达式中，只要它们在逻辑表达式出现程序点是可见的。在第 7 章和第 9 章还会引入逻辑变量和幽灵变量，它们也可以出现在逻辑表达式中。

下面列出大部分新增的算符和构造，其余的在本章各节陆续介绍。

1. 新增逻辑连接词 C 的算符 `&&` (UTF8: \wedge)、`||` (UTF8: \vee) 和 `!` (UTF8: \neg) 作为逻辑连接词，新增逻辑连接词有蕴涵 `==>` (UTF8: \Rightarrow)、等价 `<==>` (UTF8: \Leftrightarrow) 和异或 `^^` (UTF8: \oplus)。这些逻辑连接词都有按位运算的对应算符，或者是 C 本身有的 (`&`、`|`、`~`和`^`)，或者是新增的 (`-->`和`<-->`，按位蕴涵和按位等价，暂未实现)。

<i>assert</i>	→	<code>\true \false</code>	//逻辑常量
		<code>term (rel-op term)⁺</code>	//关系运算
		<code>id (term (, term)[*])</code>	//谓词应用
		<code>\is_string (term , term)</code>	//字符数组是否构成字符串的内建谓词
		<code>\is_pstring (term , term)</code>	//字符数组是否构成物理字符串的内建谓词
		<code>\membership (term , term)</code>	//字符是否在字符串中的内建谓词
		<code>\contains (term , term)</code>	//字符串是否在另一个字符串中的内建谓词
		<code>\singleton (term)</code>	//内建谓词，见 8.3.7 节
		<code>\dangling(term)</code>	//内建谓词，见 8.3 节
		<code>\list(term) \c_list(term) \dlist(term)</code>	//内建谓词，见安全 C 手册 3.1 节
		<code>\c_dlist(term) \tree(term) \data_block(term)</code>	//和本手册 8.3 节
		<code>\list_seg(term, term) \c_list_seg (term , term) \dlist_seg(term, term)</code>	
		<code>\almost_dlist(term) \c_dlist_seg(term, term) \tree_seg_r(term, term) </code>	
		<code>\tree_seg_l(term, term) \tree_seg_lr(term, term)</code>	
		<code>\typeof(C-expr) == C-type-name</code>	//内建断言， <code>\typeof</code> 内建函数，见 4.8 节
		<code>(assert)</code>	//加括号的谓词
		<code>assert && assert</code>	//合取
		<code>assert assert</code>	//析取
		<code>assert ==> assert</code>	//蕴涵
		<code>assert <==> assert</code>	//等价
		<code>! assert</code>	//否定
		<code>assert ^^ assert</code>	//异或
		<code>term ? assert : assert</code>	//三元的条件谓词
		<code>assert ? assert : assert</code>	//三元的条件谓词
		<code>\forall binders . assert</code>	//全称量化
		<code>\exists binders . assert</code>	//存在量化
<i>rel-op</i>		<code>== != <= >= > <</code>	//关系运算
<i>binders</i>	→	<code>binder (, binder)[*]</code>	
<i>binder</i>	→	<code>type-expr binder-ranges</code>	
<i>binder-ranges</i>	→	<code>variable-ident (, variable-ident)[*]</code>	
		<code>variable-ident : [term .. term (, term .. term)[*]]</code>	
<i>type-expr</i>	→	<code>logic-type-expr C-type-name</code>	
<i>logic-type-expr</i>	→	<code>build-in-logic-type</code>	
<i>build-in-logic-type</i>	→	<code>boolean integer real</code>	
<i>variable-ident</i>	→	<code>id * variable-ident variable-ident [] (variable-ident)</code>	

图 3.2 断言的文法

图 3.2 中的终结符 *id*、*C-type-name* 以及各种字面常量就是 C 中对应的词法记号。

2. 连续的比较运算 带若干个连续比较算符的构造 $t_1 \text{ relop}_1 t_2 \text{ relop}_2 t_3 \dots t_k$ 是 $(t_1 \text{ relop}_1 t_2) \&\& (t_2 \text{ relop}_2 t_3) \&\& \dots \&\& (t_{k-1} \text{ relop}_{k-1} t_k)$ 的缩写。它要求所有的算符 relop_i 是同一个“方向”的，即都属于 $\{<, <=, ==\}$ 或者都属于 $\{>, >=, ==\}$ 。像 $x < y > z$ 和 $x != y != z$ 这样的表达式是不允许的。

为了避免混淆，规定缩写记号优先。也就是，符合缩写语法的表达式一定是看成连续比较运算的缩写。例如，若 a 、 b 和 c 是整型， $a == b < c$ 被解释成 $a == b \&\& b < c$ ，而不能按 C 那样解释，看成 a 等于 $b < c$ 。即使添加括号，写成 $a == (b < c)$ 也是不行的，因为逻辑表达式语言没有 `boolean` 和 `integer` 之间的隐式类型转换，也没有强制类型转换。

对于关系运算符来说，有一个需要注意加以区别的地方。它们出现在断言中则是谓词，而出现在项中则是布尔函数。另外，指向同一个数据块的两个指针可以进行 `<`、`<=`、`>` 和 `>=` 比较，否则不能比较。相等与否的比较不受这个限制。

例 3.1 若有下面的 C 函数，怎么写它的后条件？

```
int f(int const a, int const b) { return a < b; }
```

- 先前已经提到，`\result == a < b` 不是正确的后条件，因为它是 `\result == a \&\& a < b` 的缩写。

- 先前也提到，从 C 语言表达式的语义看，`\result == (a < b)` 是正确的后条件。逻辑表达式语言不接受这样的断言，因为 $a < b$ 是布尔类型，`\result` 是整型，两者之间没有隐式的类型转换。

- 同样，`\result == (integer)(a < b)` 在此作为后条件也是不合适的，因为逻辑表达式中没有这样的显式类型转换。写成 `\result == (a < b ? 1 : 0)` 是可以的。

- `(\result != 0) == (a < b)` 是不依赖于转换的等价后条件，这是可接受的后条件，两对括号都是必不可少的。

- `\result != 0 <==> a < b` 也是可接受的后条件，因为它是两个断言之间的等价。 □

3. 量化 全称量化由 `\forall \tau x.e` 或 `\forall \tau x:[a_1...b_1, \dots, a_n...b_n].e` 表示，存在量化由 `\exists \tau x.e` 或 `\exists \tau x:[a_1...b_1, \dots, a_n...b_n].e` 表示。对于前者（即没有区间描述的情况），其中 x 是 τ 类型的约束变元， τ 可以是任何逻辑类型和 C 的类型。没有区间描述时，全称则表示对类型 τ 的所有元素，存在量化则表示存在类型 τ 的一个元素。对于后者，各个 a_i 和 b_i 都是类型 τ 的表达式，表示一个区间。对于可以使用区间的类型，可以出现多个区间。全称量化和存在量化这时分别表示类型 τ 在这些区间中的所有元素和在这些区间中存在一个元素。

4. 条件表达式和条件断言 它们的语法形式一样，都是 $c ? e_1 : e_2$ 。它可以是布尔类型的项或逻辑断言。在断言情况下，构造 $c ? e_1 : e_2$ 的两个分支 e_1 和 e_2 都必须是断言，使得它的语义等价于 $(c ==> e_1) \&\& (!c ==> e_2)$ 。

5. 逻辑函数 逻辑函数和谓词在项和断言中的应用不同于 C 的函数应用，详见第 7 章。

6. 图 3.1 和 3.2 中以反斜线开始的内建函数在后面相关章节中介绍。还有一些内建函数和谓词并没有在图 3.1 和 3.2 中列出，它们大部分在讨论实数和浮点数的 3.5 节中。

3.1 算符优先级

C 的算符优先级在此得到保持，这里还需要加入新增算符，列在表 3.1 中。在该表中，算符按照优先级从高到低次序排列。同样优先级的算符则列在同一行。

表 3.1 算符优先关系和结合性

算符类	结合性	运算符
选择	左	<code>()</code> , <code>[]</code> , <code>-></code> , <code>.</code>
一元	右	<code>!</code> , <code>~</code> , <code>+</code> , <code>-</code> , <code>*</code> , <code>&</code> , <code>(type)</code> , <code>sizeof</code>
乘除	左	<code>*</code> , <code>/</code> , <code>%</code>

加減	左	+, -
左右移	左	<<, >>
关系运算	左	<, <=, >, >=
关系运算	左	==, !=
按位与	左	&
按位异或	左	^
按位或	左	
逻辑与	左	&&
逻辑异或	左	^^
逻辑或	左	
逻辑蕴涵	右	==>
逻辑等价	左	<==>
条件运算	右	... ? ... : ...
约束	左	\forall, \exists, \lambda

3.2 语义

SCSL 逻辑表达式的语义基于一阶数理逻辑,它是仅有全函数而无部分函数的二值逻辑。表达式绝不会没有定义,这是一个重要的设计选择,规范的书写者必须知道这一点。

因为仅有全函数,这意味着可以写诸如 $1/0$ 和 $*p$ (其中 p 是 `NULL` 或悬空指针) 的项。特别是,断言 $1/0 == 1/0$ 和 $*p == *p$ 是合法的,因为它们是一阶逻辑中公理 $\forall x, x = x$ 的实例。不必为此担心,因为从这样的断言不可能演绎出任何有用的东西。像通常情况一样,书写无矛盾的断言取决于规范的设计者。例如,当引入下面引理时, y 不能为零的前提应该加上。

```
/*@ lemma div_mul_identity:
   \forall real x. \forall real y. y != 0.0 ==> y*(x / y) == x;
*/
```

3.3 定型

逻辑表达式语言是一种类型化的语言。它包含 `C` 的类型和逻辑类型。逻辑类型由下面两点定义。

- 数学类型:无界的整数 `integer` 和实数 `real`,还有布尔类型 `boolean`(其值为 `true` 和 `false`)。
- 由程序规范的书写者引入的逻辑类型。见第 7 章。

对于数值类型,有隐式的类型强制:

• `C` 的整数类型 `char`、`short`、`int`、`long` 和 `long long`,无论有符号或无符号,都是类型 `integer` 的子类型。

- 类型 `integer` 是类型 `real` 的子类型。
- `C` 的类型 `float` 和 `double` 都是类型 `real` 的子类型。

有几点需要注意:

• 布尔类型和断言类型有区别。在项的位置出现的表达式 $x < y$ 是布尔型的,而同样表达式出现在断言的位置则是断言类型的。

• 和 `C` 不一样的是,这里不存在 `integer` 和 `boolean` 两类型之间的隐式强制,也不允许两者之间有显式类型强制。

- 先前已经提到,可以对任何逻辑类型和 `C` 类型进行量化,以得到量化断言。
- 对于量化断言,具有区间限制的约束变元的类型必须是整数类型或枚举类型。

3.4 整数算术和机器整数

加、减、乘和一元减这些算术运算都可用于数学整型 `integer`。C 整数类型的变量的值被提升为数学整数类型。因此在逻辑表达式中没有算术溢出。

除运算和模运算也是数学运算，它们与对应的 C 运算在 C 机器整数上一致，因此遵循 ISO C99 的习惯，不是通常的数学上欧几里德带余数的除法。一般而言，在有余数的情况下，除的结果取靠近 0 的值。若除数为 0，则结果没有说明；其余情况下，若 q 和 r 分别是 n 除以 d 的商和余数，则：

- $|d \times q| \leq n$ ， $|q|$ 对这个性质来说是最大的；
- 如果 $|n| < |d|$ ，那么 q 是 0；
- 如果 $|n| \geq |d|$ 并且 n 和 d 有同样的符号，则 q 是正的；
- 如果 $|n| \geq |d|$ 并且 n 和 d 有相反的符号，则 q 是负的；
- $d \times q + r = n$ ；
- $|r| < |d|$ ；
- r 等于 0 或者与 n 有同样的符号。

例 3.2 下面的例子说明除和模运算的结果依赖于它们变元的符号。

- $5/3$ 是 1， $5\%3$ 是 2；
- $(-5)/3$ 是 -1， $(-5)\%3$ 是 -2；
- $5/(-3)$ 是 -1， $5\%(-3)$ 是 2；
- $(-5)/(-3)$ 是 1， $(-5)\%(-3)$ 是 -2。 □

16 进制、8 进制和 2 进制常数总是非负的。C 常数的后缀 `u`（无符号整数）和 `l`（长整数）是允许的，但没有什么含义。

在逻辑表达式中，从数学类型到 C 的整数类型 `t`（如 `char` 和 `short` 等）的强制是允许的，并且解释成模 $2^{8 \times \text{sizeof}(t)}$ 的数学结果。例如，`(unsigned char)1000` 是 1000 模 256 的结果，即 232。但是，`(signed char)1000` 是 $((1000 + 128) \bmod 256) - 128$ ，也就是 -24。

若需要在这个逻辑中表示 C 的一个表达式的值，则必须显式地加必要的类型强制。例如 C 表达式 $x*y+z$ 的值由 `(int)((int)(x*y)+z)` 表示，其中 x 、 y 和 z 都是 `int` 类型。注意，没有从 `integer` 类型到 C 整数类型的隐式强制。

例 3.3 声明

```
//@ logic int f(int x) = x+1;
```

不被允许，因为 `x+1` 隐式提升为 `integer` 类型，必须把它强制到 `int` 类型，才能与结果类型一致。可以写成

```
//@ logic int f(int x) = (int)(x+1);
```

或者，若对结果是否溢出不介意，则可以定义成

```
//@ logic integer f(int x) = x+1; □
```

一种 C 整数类型的量化表示，对应到 `integer` 在相应区间上的量化表示。

例 3.4 断言

```
\forall char c. c <= 1000
```

等价于

```
\forall integer c:[CHAR_MIN .. CHAR_MAX]. c <= 1000
```

其中下界 `CHAR_MIN` 和上界 `CHAR_MAX` 的定义在 `limits.h` 中。 □

C 语言基本类型的大小（`size`）是依赖于体系结构的，构造类型的大小可能还和编译器有关。ACSL 也没有规定这些类型的大小。`sizeof` 算符可以用于标注中，并且该算符与 C 中的 `sizeof` 一致。例如，下面的简单函数应该是可以验证的。

```
/*@ ensures \result <= sizeof(int); */
```

```
int f() {return sizeof(char);}
```

枚举类型也解释为数学整数。在这个逻辑中，若把一个 `integer` 类型的值强制为一个枚举值，其结果与 C 代码完成这个强制的结果一样。

最后解释按位运算。按位运算可以用到任何数学整数：任何数学整数有唯一的无穷的二进制补码表示，其左边有无数个 0（对于非负数）或 1（对于负数）。按位运算可用于这种表示。

例 3.5

- $7 \& 12 == \dots 00111 \& \dots 001100 == \dots 00100 == 4$
- $-8 | 5 == \dots 11000 | \dots 00101 == \dots 11101 == -3$
- $\sim 5 == \sim \dots 00101 == \dots 111010 == -6$
- $-5 \ll 2 == \dots 11011 \ll 2 == \dots 11101100 == -20$
- $5 \gg 2 == \dots 00101 \gg 2 == \dots 0001 == 1$
- $-5 \gg 2 == \dots 11011 \gg 2 == \dots 1110 == -2$ □

由于验证器是关注算术运算是否溢出的，因此在标注中声明逻辑变量和使用逻辑表达式时，要注意下面几点。

(1) 声明的逻辑变量若紧密联系到程序变量，则逻辑变量的类型尽量和相应的程序变量一致，以避免考虑 `integer` 类型的逻辑变量显式向 C 整数类型的强制，就像例 3.3 和它之前那两段文字所说的那样。

(2) 在量化断言中，其约束变元一定是逻辑变量。由于是验证 C 程序的性质，没有必要考虑量化断言对整个整数集都成立，因此宜根据所表达性质的需要，用适当的 C 整数类型作为约束变元的类型。若对约束变元有具体的区间约束，则约束变元声明为 `integer` 类型也无妨，就像例 3.4 那样。

(3) 对于第 7 章的逻辑函数和逻辑谓词等，它们变元也宜声明为 C 的类型。

(4) 对于第 9 章的幽灵变量，一般情况下，宜声明为 C 的类型。但 `integer` 类型的幽灵变量有用武之地，在第 9 章再解释。

3.5 实数和浮点数

浮点计算程序的验证分成两步来学习。

1. 暂不涉及计算精度的验证。但关注代码计算过程的正确性，即验证得到的结果与被验证代码在机器上的计算结果一致。

2. 增加计算精度的验证，完善浮点计算的验证。

第一小节的内容，是上述两种情况下都要关注的内容。第二小节介绍在不涉及精度验证的情况下，对浮点计算我们能验证什么性质。

3.5.1 实数与浮点数的联系与区别

浮点常数及其上的运算解释为数学实数及其上运算。这样，出现在断言语言中的 C 语言浮点型和双精度型的常量和变量都隐式地提升为实型常量和变量。必要时，`integer` 类型的常量和变量也提升为 `real` 类型的常量和变量。通常的二元运算都解释成实数上的运算，在这样的情况下，计算不会出现任何舍入或溢出。

例如，在逻辑表达式中， $1e+300 * 1e+300$ 等于 $1e+600$ ，即使 $1e+600$ 大于双精度所能表示的最大数也不必在意，因为没有溢出。 $2*0.1$ 等于实数 0.2，而不是等于它的浮点近似，因为不存在“舍入”。

和把 C 的整数类型提升到数学整数相比，对于浮点类型，不一样的地方是，它有 3 个特殊浮点数是不能自然地映射到实数的，即 IEEE 754 中作为“非数” (*not a number*)、 $+\infty$ 和 $-\infty$ 的特殊值。稍后对这些特殊值有详细讨论。SCSL 逻辑只有全函数，但有一个例外，隐式提升函数对“非数”、 $+\infty$ 和 $-\infty$ 这 3 个值没有规定结果。

在逻辑表达式中，实数也可以用 C99 的 16 进制表示法，即 `0xhh.hhp±dd`，其中 *h* 是 16 进制数字，*d* 是 10 进制数字，`0xhh.hhp±dd` 表示 $hh.hh \times 2^{\pm dd}$ ，例如 `0x1.Fp-4` 是 $(1+15/16) \times 2^{-4}$ 。

常见的比较运算符 `==`、`!=`、`<`、`<=`、`>` 和 `>=` 可用于 `real`、`double` 和 `float` 类型，也就是它们是重载的算符，对于不同类型的运算对象，它们被解释成相应类型上的运算。其实，`float` 和 `double` 类型上的这些比较运算的结果，与把它们隐式提升到 `real` 类型后，在 `real` 类型上的比较结果是一样的。

从 C 的整数类型或单精度类型到单精度类型或双精度类型的强制和 C 中做法的一样，即应用同样的转换操作。但是，从实数到单精度或双精度值的转换依赖于由 IEEE 754 标准定义的不同的舍入模式。这些舍入模式可以由一个内建的逻辑类型来定义。先给出相关类型。

```
/*@ type rounding_mode = \Up, \Down, \ToZero, \NearestAway, \NearestEven
```

然后舍入一个实数可以由显式地使用下列两个内建逻辑函数之一来完成。

```
logic float \round_float(\rounding_mode m, real x);
```

```
logic double \round_double(\rounding_mode m, real x);
```

这两个函数给出实数 *x* 在模式 *m* 下的舍入结果，结果类型是 `float` 或 `double`。

强制操作 (`float`) 和 (`double`) 应用到数学整数或实数 *x* 等价于在上述函数中，把 `\NearestEven` 舍入模式（这是 C 程序缺省舍入模式）应用到 *x*。例如，`(float)0.1` 等于 13421773×2^{-27} ，后者等于 `0.100000001490116119384765625`。如果源实数太大，则导致结果是 $+\infty$ 和 $-\infty$ 中的一个。

还要注意，和整数不一样，浮点常数的后缀 `f` 和 `l` 是有意义的，因为它们隐含着增加一个上面这样的强制算符，使用的是缺省舍入模式。

强制操作的语义保证，如果没有计算溢出并且程序中的缺省舍入模式没有改变，则 C 的浮点运算 $e_1 \text{ op } e_2$ 的浮点结果和逻辑表达式 `(float)($e_1 \text{ op } e_2$)` 有同样的实数值。但要注意，对于浮点数的相等比较并非都是这样。C 的浮点表达式 `-0.0 + -0.0` 等于浮点数 `-0.0`，`-0.0` 和 `0.0` 的相等比较的结果为假，因为 `0.0` 是逻辑表达式 `(float)(-0.0 + -0.0)` 的值。

最后，还需要提供一些谓词，用于检查它们的变元有限，无限还是非数。

```
predicate \is_finite(double x);           // 是有限的双精度数
predicate \is_plus_infinity(double x);    // 等于正无穷
predicate \is_minus_infinity(double x);   // 等于负无穷
predicate \is_infinity(double x);        // 等于正无穷或负无穷
predicate \is_NaN(double x);             // 是 NaN 双精度数。
```

在 IEEE754 的规则下，任何两个 NaN 的值的比较，其结果都为假。因此，若一个双精度的变量 *d* 是 NaN，那么 C 语言表达式 `d == d` 和逻辑表达式 `d == d` 都为假，逻辑表达式 `\real_of_double(d)` 没有定义。但是逻辑表达式 `\real_of_double(d) == \real_of_double(d)` 为真，因为该式被看成恒等公理 $x \equiv x$ 的一个特殊实例。

对于非 NaN 的浮点数，它的符号可以用逻辑函数 `\sign` 抽取。

```
/*@ type sign = \Positive | \Negtive;
logic sign \sign(float x);
logic sign \sign(double x); */
```

对 `real` 类型的变量的量化当然是在所有实数上的量化。对 `float` 和 `double` 类型的变量的量化也是允许的，但量化的范围是所有那些能分别代表单精度数和双精度数的实数，在这个范围，但它们不包括 NaN、 $+\infty$ 和 $-\infty$ 。

传统的数学运算，例如指数、正弦和余弦函数等都可作为 SCSL 的内建函数来使用，其中标注 `\pi` 指实数 π ，标注 `\e` 指自然对数的底，并且 `\log(\e) == 1`，`\exp(1) == \e`。

```
integer \min(integer x, integer y); integer \max(integer x, integer y);
real \min(real x, real y); real \max(real x, real y); integer \abs(integer x);
real \abs(real x);
real \sqrt(real x); real \pow(real x, real y);
integer \ceil(real x); integer \floor(real x);
real \e; real \exp(real x); real \log(real x); real \log10(real x);
real \pi; real \cos(real x); real \sin(real x); real \tan(real x);
real \cosh(real x); real \sinh(real x); real \tanh(real x);
real \acos(real x); real \asin(real x); real \atan(real x);
real \atan2(real x, real y); real \hypot(real x, real y);
```

因此，对于 C 语言数学函数库中的函数，都有函数协议，协议中的 `\result` 等于上述相应内建数学函数的应用。目前的协议给不出对这些内建数学函数应用的精度描述。

3.5.2 对浮点计算程序的验证

验证计算结果的精度，是浮点计算程序验证的重要内容。在目前不提供计算结果的精度验证情况下。可以从下面几方面学习浮点程序的验证。

1. 可以验证函数代码实现的算法是否与标注所描述的抽象算法一致。

例 3.6 用循环迭代算法对数组 `arr` 的元素求和，代码和标注见图 3.3。标注中归纳定义的逻辑函数 `sum`，是对变元数组的元素 `arr[0]`, `arr[1]`, ..., `arr[n-1]` 逐步累加求和。函数的循环代码，体现出的也是对形参数组 `arr` 的相同计算过程。函数后条件给出的结论是，代码完成的计算就是逻辑函数 `sum` 的计算。也就是说，断言中描述的浮点计算过程和结果表达式与程序运行的过程和结果表达式一致，这个函数是可以验证的。 □

这样的例子在本系统所提供的参考实例中还有，例如浮点类型的矩阵分块乘算法。

```
#include<limits.h>
/*@logic double sum(double* arr,int n) = n < 0 ? (double)0.0 : sum(arr, n-1) + arr[n];

/*@ requires 0 <= n <= INT_MAX && \length(arr) == n;
   ensures \result == sum(arr, n-1) && \length(arr) == n; */
double fsum(double* const arr, const int n){
    int i; double ret; ret = 0.0;
    //@loop invariant 0 <= i <= n <= INT_MAX && \length(arr) == n && ret == sum(arr, i-1);
    for(i = 0; i < n ; i++){
        ret = ret + arr[i];
    }
    return ret;
}
```

图 3.3 对 `arr[0] + ... + arr[n-1]` 求和的代码和标注

2. 代码中有浮点常数时，对计算结果的关注不同，能否验证则也可能不一样。下面的例 3.7 讨论的是能否按浮点计算的语义，验证代码的计算结果是正确的。

例 3.7 图 3.4 中两个函数的代码是一样的，都是通过累加 `0.1f+0.1f+ ... +0.1f` 的方式计算 `0.1f * n`。

本例两个函数的证明目标不同。函数 `clock_single1` 证明 `\result == clockSum(n)`，其中 `clockSum(n)` 是一个归纳定义的逻辑函数。该逻辑函数体现出计算结果是通过逐步累加 `0.1f`

得到的，与代码的实际计算过程是一致的。对于该函数的验证来说，最关键的是在循环出口证明验证条件（略去下式中的一些无关紧要的断言）：

$$t == (\text{float})(\text{clockSum}(i-1) + (\text{float})0.1) ==> t == \text{clockSum}(i)$$

基于 `clockSum(n)` 的定义可以得到这个证明。因此本例验证关注的仍然是计算过程。

函数 `clock_single2` 证明 `\result == (float)(0.1 * n)`，就是试图证明 n 个 0.1 累加的结果等于 $0.1 * n$ 。由于浮点计算存在误差，该函数的验证得不到这个结论。对于该函数的验证来说，最关键的是在循环出口证明验证条件（略去下式中的一些无关紧要的断言）：

$$t == (\text{float})(0.1 * (i-1)) + (\text{float})0.1 ==> t == (\text{float})(0.1 * i)$$

忽略类型强制的话，就是证明 $t == 0.1 * (i-1) + 0.1 ==> t == 0.1 * i$ 。real 类型上的定理 $x * (y-1) + x == x * y$ 在此不能使用，因为这里是浮点数的近似计算，不存在这个定理，因此证明不了。

从本例可以看出，代码中有浮点常数的浮点计算程序是可能验证的，关键是待证性质的描述要紧密地联系到获得该性质的计算过程。本例第二个函数缺少这样的联系，它试图用 real 类型上的性质来证明程序结果等于 $0.1 * n$ ，在浮点近似计算时就证明不了了。□

```
#include<limits.h>
//@logic float clockSum(int n)= n <= 0 ? (float)0.0 : (float)(clockSum(n-1) + (float)0.1);
/*@ requires 0 <= n <= INT_MAX;
    ensures \result == clockSum(n); */
float clock_single1(const int n){
    float t; int i;
    //@ loop invariant 0 <= i <= n <= INT_MAX && t == clockSum(i) && !\is_infinite(t);
    for (i = 0, t = 0.0f; i < n; i++){
        t = t + 0.1f;
    }
    return t;
}
/*@ requires 0 <= n <= INT_MAX;
    ensures \result == (float)(0.1 * n); */
float clock_single2(const int n){
    float t; int i;
    //@ loop invariant 0 <= i <= n <= INT_MAX && t == (float)(0.1 * i) && !\is_infinite(t);
    for (i = 0, t=0.0f; i < n; i++) {
        t = t + 0.1f;
    }
    return t;
}
}
```

图 3.4 同样方式计算 $0.1f * n$ ，但验证的目标不一样

3. 在目前不提供计算结果的精度验证情况下，由于机器误差的存在，一般来说，迭代计算的浮点程序的具体结果值是验证不了的。举一个典型的例子。

例 3.8 下面的迭代计算是 Muller 给出的[7]，代码见图 3.5。

$$u_1 = 2, u_2 = -4,$$

$$u_n = 111 - 1130/(u_{n-1}) + 3000/((u_{n-1}) * (u_{n-2})),$$

其中 n 的取值范围： $3 \leq n \leq 31$ 。

图 3.5 是针对这个迭代计算的代码，它应该收敛于 6。但是由于机器误差，它在任何系统和任何精度下均出错，均收敛于 100。当 $i \leq 16$ 时，迭代的结果向 6 收敛。当 $i > 16$ 后，继续迭代计算使得结果逐步向 100 收敛。

若把该程序的运算对象和运算都提升到 `real` 类型，则程序在 `real` 类型上的计算结果与验证结果一致，都收敛于 6。这个结果对验证来说没有用处，因为在机器上是按浮点类型进行计算，计算结果收敛于 100。收敛于 100 这个结果不是程序员想要的，它是因机器误差引起的。 □

```
#include <stdio.h>
int main(){
    double u = 2.0, v = -4.0, w;
    int max = 31, i;
    for(i=3; i<=max; i++){
        w = 111. - 1130./v + 3000./(v*u);
        u = v; v = w;
        printf("u%d = %1.17g\n", i, v);
    }
    return 0;
}
```

图 3.5 迭代计算未得正确结果的例子

3.6 数组和指针

取地址算符、数组访问、指针算术和指针脱引用运算类似于它们在 C 中对应的运算。由于逻辑语言中没有指针类型，因此指针在断言语言中的使用一定受到限制。在这样的限制下，指针被看成是整型的。

先说指向栈区或静态数据区的指针，注意数组名也是指针。取地址运算的使用必须小心，在逻辑表达式中取地址运算得到的是不依赖于 C 内存的一个抽象值而不是具体值。根据安全 C 语言的规定，取地址算符的使用范围大大缩小。对于指针 `p` 和 `q`，限定必须用 `p == q` 或 `p == q ± e` 这样的断言来表示 `p` 和 `q` 指向同一个数据块（其中 `e` 是整型表达式），同时表示它们的大小关系。否则 `p` 和 `q` 被认为指向不同的数据块或者指针关系断言不合要求。上面的限定同时体现出不允许对 `p` 和 `q` 进行加减乘除和比较等其他运算（虽然 `q - p == e` 本质上就是 `p == q - e`）。`p != q`（或 `!(p == q)`）表示两个指针不相等，它用来表示 `p` 和 `q` 指向不同的数据块。指向同一个数据块但并不相等的 `p` 和 `q` 用 `p == q ± e` 方式表示。同样，`!(p == q ± e)`（或 `p != q ± e`）也仅表示 `p` 和 `q` 指向不同的数据块。

指向不同数据块的指针进行相等与否比较和大小比较毫无意义，其它操作更免谈。因此在逻辑表达中，除了上面提到的“!”外，不允许指向不同数据块的两个指针进行任何操作。

可以用 `p == \null` 来表示指针 `p` 不指向任何数据。确实不关心 `p` 的具体指向时，在 `p` 不会是悬空指针的情况下，可以用 `p != \null` 来表示 `p` 指向某个数据块。

一些有两个指针形参 `p` 和 `q` 的函数，调用的两个实参指向不同数据块和指向同一个数据块都能得到各自所要的结果。这种情况是存在的，例如下一章的例 4.5。该函数的验证肯定分成两种情况。怎样用断言区别这两种情况？若指向同一数据块的要求用断言 `p == q` 表示，则指向不同数据块的要求用断言 `!(p == q)` 或 `p != q` 表示。若指向同一数据块的要求用断言 `p == q ± e` 表示，则指向不同数据块的要求用断言 `!(p == q ± e)`（或 `p != q ± e`）表示。

基于安全 C 的定义，指向动态分配的数据块的指针之间只能出现相等与否的断言，或者指针是否等于 `NULL` 的断言。

和 C 语言不同的是，SCSL 不会把 0 隐式转换成指针 `\null`，也不允许这样的强制转换。

和 C 不一样的是，在逻辑表达式中，包括在幽灵代码中，可以出现数组区间表达式和数组区间之间的关系断言。例如，数组区间表达式 `a[e1..e2]`（`a` 是数组名或数组元素的指针，`e1` 必须小于或等于 `e2`）和数组区间关系断言 `a[e1..e2] <= b[e3..e4]`（`e4 - e3` 必须等于 `e2 - e1`）。若 `e3 = e1 + n`，则后者表示 `\forall integer i: [e1..e2]. a[i] <= b[i + n]`。因此数组区间之间的关系断言实际上是对应全称量化断言的一种简洁表示。即使 `a` 和 `b` 是同类型的程序数组，也不允许用 `a == b` 来表示它们的对应元素相等，因为在 C 语言中数组名是指向数组第一个元素的指

针，若允许则断言 $a == b$ 可能有歧义。为避免歧义，表示程序数组 a 和 b 对应元素相等时，两边都必须用区间表达式，例如断言 $a[0..99] == b[0..99]$ ，其中 a 和 b 的长度都是 100。对于程序数组，断言 $a == b$ 表示两个指针相等。有关逻辑数组在第 7 章介绍。在幽灵代码中，数组还可以区间赋值。有关幽灵变量和语句在第 9 章介绍。

对于一般的数组和指针，有 2 个内建函数。

1. 长度函数 `integer \length(τ p)`: 这是一个多态函数， τ 可以是任意的数组类型或任意的指针类型。`\length(τ p)` 表示有效指针 p 所指向的一片 τ 类型元素构成的数据区间中 τ 类型元素的个数。注意，在 C 语言中，单独的数组名是指针类型。

2. 偏移函数 `integer \offset(τ p)`: 这也是一个多态函数。其结果是有效指针 p 所指向的元素在这片区间中的偏移（以一个元素所占区间大小为单位）， $0 \leq \text{\offset}(p) < \text{\length}(p)$ 。

各种情况下这两个属性的计算请见安全 C 语言使用手册 2.1 节。

例如，若 C 函数的形参 a 的声明是 `int a[10]`、`int a[]` 或 `int *a`，当包含 `\length(a)` 和 `\offset(a)` 的断言出现在函数前条件中时，则认为这些断言是对调用时实参 p 指向其中的数据区间的大小和在该数据区间中 p 所指向位置的一种约束。若函数前条件中无这样的两个断言（或缺少其中一个断言），则默认有断言 `\length(a) == 1` 和 `\offset(a) == 0`（或默认其中一个断言）。

例 3.9 图 3.6 是冒泡排序函数的 C 代码及其前后条件和循环不变式。

```

#define MAX_LEN 10000
/*@ requires \length(a) == m && 0 < m <= MAX_LEN && \offset(a) == 0;
    assigns a[0..m-1];
    ensures \length(a) == m && 0 < m <= MAX_LEN && \offset(a) == 0 &&
                                                \forall int i:[0..m-2]. a[i] <= a[i+1]; */
void bubbleSort(int *const a, int const m) {
    int low, up, j, k, temp;
    low = 0; up = m-1; k = up;
    /*@ loop invariant \length(a) == m && 0 < m <= MAX_LEN && \offset(a) == 0 &&
                                                low == 0 && up == m-1 &&
        (k == up && (\forall int i:[k+1..up-1]. a[i] <= a[i+1]) ||
         low <= k < up && (\forall int i:[low..k]. a[i] <= a[k+1]) &&
         \forall int i:[k+1..up-1]. a[i] <= a[i+1]); */
    while( k != low) {
        j = low;
        /*@ loop invariant \length(a) == m && 0 < m <= MAX_LEN && \offset(a) == 0 &&
                                                low == 0 && up == m-1 && k != low && low <= j <= k &&
        (k == up && (\forall int i:[low..j-1]. a[i] <= a[j]) &&
         (\forall int i:[k+1..up-1]. a[i] <= a[i+1]) ||
         low < k < up && (\forall int i:[low..j-1]. a[i] <= a[j]) &&
         (\forall int i:[low..k]. a[i] <= a[k+1]) && \forall int i:[k+1..up-1]. a[i] <= a[i+1]);
        */
        while( j != k ) {
            if (a[j] > a[j+1]) { temp = a[j]; a[j] = a[j+1]; a[j+1] = temp;
            }
            j = j + 1;
        }
        k = k - 1;
    }
}

```

图 3.6 冒泡排序函数

对于具有断言`\length(b) == 100 && \offset(b) == 0`的调用 `bubbleSort(b, 100)`来说, 调用点的断言蕴涵 `bubbleSort` 的前条件。把 `bubbleSort` 的前条件中的形参 `a` 和 `m` 分别代换成 `b` 和 `100`, 则调用后得到

`\forall` integer `i`: `[0..98]. b[i] <= b[i+1]`。 □

在逻辑表达式中可以使用指向函数的指针, 但仅能用于相等性比较。

例 3.10 图 3.7 是使用函数指针的例子。对于函数指针来说, 只能检查是否相等。 □

```
int f(int x);
int g(int x);
...
//@ requires p == &f || p == &g;
void h(int(*p)(int)) { ... }
```

图 3.7 使用函数指针的一个简单例子

3.7 结构体、共用体和数组

C 聚集类型的对象（结构体、共用体和数组）可以在逻辑表达式中作为项的值。它们可以作为参数传递给逻辑谓词和函数, 也可以从逻辑函数返回, 除了共用体外, 其余的也可以进行相等与否比较等。3.6 节已经提到, 数组名作为项时在逻辑表达式和幽灵代码中代表整个数组。

在逻辑表达式中, 对于一个共用体 `u` 来说, 在任何情况下, 不能出现有关 `u.a` 的断言和 `u.b` 的断言的合取, 因为共用体的不同成员 `a` 和 `b` 不会同时存在。若出现这样的情况, 则该合取式为 `\false`。

对于 C 代码中已定义的聚集类型, 可以声明这些类型的逻辑变量, 它们可用来表达函数出口聚集类型的对象与其在函数入口的值的变化的。

3.8 字符串

C 语言虽然没有字符串类型, 但有一些与字符串相关的内容。

- 有字符串常量。例如 “abc”和 “123”, 其中的字符不能是 ‘\0’。
- 字符指针指向的地址连续的字符序列若出现 ‘\0’, 则第一个 ‘\0’ 之前的部分构成由这个字符指针指向的字符串。该指针这时可以看成字符串指针。连同其存储特性一起考虑的字符串称为物理字符串。物理字符串的存储特性就是指它以 ‘\0’ 作为结束标记。
- 有一些适用于物理字符串的内建函数。

在规范语言的逻辑表达式中也可以使用字符串。在逻辑表达式中出现的字符串称为逻辑字符串, 其特点是不关注其物理存储是否有结束标记 ‘\0’。逻辑字符串常量的表示法与 C 的一致, 逻辑字符串常量中允许出现的字符也与 C 的一致。例如 “abc”和 “123” 都是逻辑字符串常量, 此时不关心它们的存储特性。

为方便下面介绍系统提供的涉及逻辑字符串的内建函数, 我们引入逻辑字符串类型 `STRING`。`STRING` 实际上是一种记录类型, 它需要两个域, 其一是长度足够的字符数组或指针, 另一个是大于等于 0 的整数, 后者用来指明存放在该数组中的字符串的长度。`STRING` 类型的逻辑字符串与存放在字符数组中的物理字符串的重要区别是, 没有以 ‘\0’ 作为结束标记这个概念, 因此它另需一个整数来表示逻辑字符串的长度。类型 `STRING` 相当于 `char* × integer` 的缩写, 其中 `char*` 类型的指针是指向该逻辑字符串第 1 个字符的指针, 不管该逻辑字符串是从相关数据块中什么偏移开始的。`\string(str, m)` 是 `STRING` 类型的最简单的非常量的项形式, 它表示从 `str` 指向位置开始的长度为 `m` 的逻辑字符串 (稍后有严格的描述)。

若在代码中出现赋值 `s = “abc”`, 则之后可得到断言 `\string(s, 3) == “abc”`。其中 `s` 指向的字符数据块的内容构成 “abc” (未考虑其后的 ‘\0’)。类似于赋值 `x = 3` 之后可得断言 `x == 3` 那样。

在规范语言中，不允许声明 `STRING` 类型的变量。若确实需要，可通过声明逻辑变量 `logic char* str, int m` 和使用 `\string(str, m)` 来达到所需的效果。

注意，若物理字符串被改变，例如对构成字符串的字符数组的元素赋值，则基于它能得到的逻辑字符串也改变。但是，若物理字符串的结束标记被修改而不复存在，例如当长度为 `m` 的物理字符串的结束标记被修改，其对应的长度为 `m` 的逻辑字符串仍然存在。正是由于 C 语言的这些特点，导致上述这种特定的面向 C 的 `STRING` 类型的设计。

逻辑字符串上所允许的运算是并置（用“+”号作为算符）和按字典序的大小比较（用通常的 6 种关系算符），其中字符串相等与否的比较也就是字符串相同与否的比较。下面是与字符串有关的内建函数，其中从第 6 个开始都是仅与逻辑字符串有关的函数。这些函数尽量都用第 7 章提供的逻辑定义的方式或接近它的方式进行描述。

1. 判断字符指针 `str` 指向的内容是否构成长度为 `n` 的逻辑字符串的谓词

```
predicate \is_string(char *str, integer n) =
    n >= 0 && n <= \length(str) - \offset(str) &&
    (\forall integer i: [\offset(str) .. \offset(str)+n-1]. ^str[i] > 0);
```

其中 `^str` 代表 `str` 所指向的数据块（若用第 7 章方式，“^”符号是不需要的），`\offset(str)` 表示 `str` 指针在 `^str` 数据块中的偏移，即 `str == ^str + \offset(str)`。由此看出，谓词应用 `\is_string(s, m)` 的值依赖于实变元的显式描述的一些属性，例子可见 4.3 节例 4.5。

若 `str` 是字符数组的名字，则上面谓词的定义体简化为：

```
n >= 0 && n <= \length(str) && (\forall integer i: [0 .. n-1]. str[i] > 0)
```

2. 判断字符指针 `str` 指向的内容是否构成长度为 `n` 的物理字符串的谓词

```
predicate \is_pstring(char *str, integer n) =
    \is_string(str, n) && \length(str) - \offset(str) >= n+1 && ^str[\offset(str)+n] == 0;
```

显然，`\is_pstring(str, n) ==> \is_string(str, n)`。

3. 求物理字符串长度的函数 `\strlen`

在断言中，不管是逻辑字符串还是物理字符串都要有长度信息。但是有些代码并不关注物理字符串的长度，它们是依据是否扫描到结束标记来编程的。为了便于把描述物理字符串的断言统一到都有长度变元的形式，引入函数 `\strlen`。

该函数的类型是 `char* → integer`，并且要求其变元必须指向一个物理字符串。

按上面所给定的类型，写不出该函数的归纳定义。该函数是一个未解释函数，其含义体现在如下的内建引理中。

lemma pstringProperty:

```
\forall char *str. \forall integer n.
    \is_string(str, n) && ^str[\offset(str)+n] == 0
    ==> \strlen(str) == n;
```

在不关心物理字符串的长度时，程序标注可以用 `\is_pstring(str, \strlen(str))` 来表示 `str` 指向一个物理字符串。有了这个函数后，在串长为 `m` 时，把 `\is_pstring(str, m)` 写成 `\is_pstring(str, \strlen(str)) && \strlen(str) == m` 也是一样的。

4. 取逻辑字符串的函数（使用条件为 `n >= 0`）

从字符数组取长度为 `n` 的逻辑字符串的函数，其类型是：`char* × integer → STRING`。

```
logic STRING \string(char *str, integer n) = { ^str[\offset(str) .. \offset(str)+n-1], n};
```

若 `n` 等于 0，则该逻辑函数的结果是空串。

即该函数的结果是由 `^str[\offset(str)]`, `^str[\offset(str)+1]`, ..., `^str[\offset(str)+n-1]` 构成的长度为 `n` 的字符串。为方便表达，这里采用第 7 章所提供的数组区间方式（即 `^str[\offset(str).. \offset(str)+n-1]`）来表示逻辑字符串的内容，并用 `{ ..., ... }` 方式来表示有两个域的记录的值得。

这个函数仅在 $(\text{\is_string}(\text{str}, m) \parallel \text{\is_pstring}(\text{str}, m)) \ \&\& \ m \geq n$ 情况下使用。先前介绍说 $\text{\string}(\text{str}, m)$ 是 **STRING** 类型的最简单的非常量的项形式，其实就是把 \string 函数应用的结果作为逻辑字符串的项。

为易理解起见，下面介绍关于字符串操作的内建函数时，所列谓词和函数都把字符指针 str 、 str1 和 str2 等都限定为是数组名的情况。

5. 逻辑字符串之间的关系运算 == 、 != 、 < 、 <= 、 > 和 >= （使用条件为 $m \geq 0$ 且 $n \geq 0$ ）

它们的类型都是 $\text{STRING} \times \text{STRING} \rightarrow \text{boolean}$ 。上面这些关系算符都是中缀算符，结果类型是布尔型，在下面介绍它们的定义时，把它们看成谓词的名字。串并置算符“+”也按这种方式描述。

(1) 相等与否运算 == 和 !=

predicate “ == ” ($\text{STRING} \ \text{\string}(\text{str1}, m)$, $\text{STRING} \ \text{\string}(\text{str2}, n)$) =
 $m == n \ \&\& \ (\text{\forall} \text{integer } i:[0..m-1]. \text{str1}[i] == \text{str2}[i]);$

这是字符串 $\text{\string}(\text{str1}, m)$ 和 $\text{\string}(\text{str2}, n)$ 的相等关系运算，不相等关系可类似地定义。

(2) 大小关系运算 < 、 <= 、 > 、 >=

小于关系的定义如下，其他关系的定义可类似地写出。

predicate boolean “ < ” ($\text{STRING} \ \text{\string}(\text{str1}, m)$, $\text{STRING} \ \text{\string}(\text{str2}, n)$) =
 $\text{\exists} \text{integer } k:[0..n-1]. \quad \quad \quad // \text{若 } n == 0, \text{ 则结果一定为 } \text{\false}$
 $(\text{\forall} \text{integer } i:[0..k-1]. \text{str1}[i] == \text{str2}[i]) \ \&\& \quad // \text{这两行针对 } m \geq 0, n > 0$
 $(k \geq m \parallel \text{str1}[k] < \text{str2}[k]); \quad // \text{若 } k \geq m, \text{ 则 } n > m \text{ 且两者前 } k-1 \text{ 个}$
 $\quad \quad \quad // \text{字符相同, 则结果一定为 } \text{\true}$

6. 逻辑字符串并置函数“+”（使用条件为 $m \geq 0$ 且 $n \geq 0$ ）

该函数的类型是: $\text{STRING} \times \text{STRING} \rightarrow \text{STRING}$ 。

logic STRING “+” ($\text{STRING} \ \text{\string}(\text{str1}, m)$, $\text{STRING} \ \text{\string}(\text{str2}, n)$) =
 $\text{\string}(\text{str3}, m+n)$,

其中 str3 是新的字符数组，它满足：

if ($m == 0 \ \&\& \ n == 0$) 结果 $\text{\string}(\text{str3}, m+n)$ 是空串；

else 结果 $\text{\string}(\text{str3}, m+n)$ 满足

$\text{str3}[0..m-1] == \text{str1}[0..m-1] \ \&\& \ \text{str3}[m..m+n-1] == \text{str2}[0..n-1]$ 。

7. 取逻辑字符串前缀的函数 \prefix （使用条件为 $m \geq 0$ ， $n \geq 0$ 且 $n \leq m$ ）

该函数的类型是: $\text{STRING} \times \text{integer} \rightarrow \text{STRING}$ 。

logic STRING $\text{\prefix}(\text{STRING} \ \text{\string}(\text{str}, m)$, integer n) = $\text{\string}(\text{str}, n)$;

该函数的结果是 str 数组中从第 1 字符到第 n 个字符构成的逻辑字符串。注意，若 $n == 0$ ，则结果是空串。

8. 取逻辑字符串后缀的函数 \suffix （使用条件为 $m \geq 0$ ， $n \geq 0$ 且 $n \leq m$ ）

该函数的类型是: $\text{STRING} \times \text{integer} \rightarrow \text{STRING}$

logic STRING $\text{\suffix}(\text{STRING} \ \text{\string}(\text{str}, m)$, integer n) =
 $\text{\string}(\text{str}+(m-n), n); \quad (= \text{str}[m-n..m-1])$

该函数的结果是 str 数组中从第 $m-n+1$ 字符开始到第 m 个字符，共 n 个字符构成的逻辑字符串。注意，若 n 等于 0，则结果是空串。

9. 一个字符是否属于某个逻辑字符串的谓词 \membership （使用条件为 $m \geq 0$ ）

该谓词的类型是: $\text{STRING} \times \text{char} \rightarrow \text{boolean}$ ，它只用于有长度信息的逻辑字符串。

inductive $\text{\membership}(\text{STRING} \ \text{\string}(\text{str}, m)$, char c) =
 $m == 0 ?$
 $\text{\false} :$

$m \geq 1 \ \&\& \text{str}[0] == c ?$

$\backslash\text{true}:$

$\backslash\text{membership}(\backslash\text{string}(\text{str}+1, m-1), c);$

10. 逻辑字符串 str1 是否包含 str2 的谓词 $\backslash\text{contains}$ (使用条件为 $m \geq 0$ 且 $n \geq 0$)

该归纳谓词的类型是: $\text{STRING} \times \text{STRING} \rightarrow \text{boolean}$

$\text{inductive } \backslash\text{contains}(\text{STRING } \backslash\text{string}(\text{str1}, m), \text{STRING } \backslash\text{string}(\text{str2}, n)) =$

$m < n ?$

$\backslash\text{false}:$

$\backslash\text{prefix}(\backslash\text{string}(\text{str1}, m), n) == \backslash\text{string}(\text{str2}, n) ?$

$\backslash\text{true}:$

$\backslash\text{contains}(\backslash\text{string}(\text{str1}+1, m-1), \backslash\text{string}(\text{str2}, n));$

11. 计算 str2 首次出现在 str1 中的索引的函数 $\backslash\text{index}$ (使用条件为 $m \geq 0$ 且 $n \geq 0$)

该函数的类型是: $\text{STRING} \times \text{STRING} \rightarrow \text{long}$

$\text{logic long } \backslash\text{index}(\text{STRING } \backslash\text{string}(\text{str1}, m), \text{STRING } \backslash\text{string}(\text{str2}, n)) =$

$m < n ?$

$n-1:$

$\backslash\text{prefix}(\backslash\text{string}(\text{str1}, m), n) == \backslash\text{string}(\text{str2}, n) ?$

$0:$

$1 + \backslash\text{index}(\backslash\text{string}(\text{str1}+1, m-1), \backslash\text{string}(\text{str2}, n))$

该函数的结果若在区间 $[0..m-n]$ 之间, 则是逻辑字符串 str2 首次出现在 str1 中的索引, 否则结果大于等于 m , 表示逻辑字符串 str2 并未出现在 str1 中。

12. 取逻辑字符串中第 n 个字符的函数 $\backslash\text{char}$ (使用条件为 $m > n > 0$)

该函数的类型是 $\text{STRING} \times \text{long} \rightarrow \text{char}$ 。

$\text{logic char } \backslash\text{char}(\text{STRING } \backslash\text{string}(\text{str}, m), \text{long } n) = \text{str}[n-1]$

对于上述这些内建函数, 在 C 语言的标准库中有功能类似的函数, 但是它们都是操作于物理字符串, 出现在程序中。这里的逻辑函数和谓词是出现在断言中。

例 3.11 图 3.8 是 C 字符串库函数 strcpy 函数和 strcat 函数的协议。早年的 strcpy 对形参 src 没有修饰符 const , 它可用于更多场合, 其函数协议见例 4.5。 □

```
// 库函数 strcpy(char* dest, const char* src)的验证, 这个协议适用于 dest 和 src 指向不同的数据块
//@ logic char *olddest, *oldsrc; logic integer m;
/*@ requires \is_pstring(src, m) && \length(dest) - \offset(dest) > m && olddest == dest &&
    oldsrc == src && \old(dest) == dest;
    assigns \old(dest)[0..m];
    ensures \result == olddest && \is_pstring(\result, m) && \is_pstring(oldsrc, m) &&
    \string(\result, m) == \string(oldsrc, m);
*/ char * strcpy(char * dest, const char * src);

//@ logic integer m, n; logic char *olddest, *oldsrc;
/*@ requires \is_pstring(dest, m) && \is_pstring(src, n) && m + n < \length(dest) - \offset(dest) &&
    olddest == dest && oldsrc == src && \old(dest) == dest;
    assigns \old(dest)[m..m+n];
    ensures \is_pstring(\result, m+n) && \is_pstring(oldsrc, n) && \result == olddest &&
    \string(\result, m+n) == \string(olddest, m) + \string(oldsrc, n);
*/
char * strcat(char* dest, const char* src);
```

图 3.8 库函数 strcpy 函数和 strcat 函数的协议

从例 3.11 可以看出，允许在逻辑表达式中使用字符串相等与否比较，可免去程序员用量化断言通过逐字符比较来判断两个字符串相等与否。但是，若代码修改字符串中的字符或结束标记‘\0’时，还是免不了要使用这种展开的量化断言。对于一般数组的相等与否比较也是这样。

例 3.12 图 3.9 代码中的字符指针 p、q 和 t 都指向字符串“abc”，这三个指针的值是否相等，取决于编译器在只读数据区安排 1 个还是 3 个字符串常量“abc”。因此在判断它们是否指向相同的字符串时，宜调用库函数 strcmp 来判断，而不宜直接比较 p 是否等于 q。对于图 3.9 的程序，验证器不认为 p、q 和 t 的值是相等的 □

```
char* p = "abc";
main() {
    char* q;
    char* t = "abc"; q = "abc";
    ... ..
}
```

图 3.9 字符串常量的一个例子

对于字符数组或字符指针，当用它来存放字符串或者作为字符串指针时，通常仍然要继续关心它作为数据块或数据块指针的特性，即长度 length(...) 和偏移 offset(...) 性质。这在例 3.11 有体现。

第 4 章 函数协议

函数协议 (*function contract*) 是指函数定义和函数使用之间的约定, 函数协议的语法见图 4.1。从语法上看, 函数协议允许出现在 C 声明可出现的地方, 但实际限定它们必须出现在相应函数的声明或定义之前。尤其是, 若函数 *f* 的应用和定义分处不同源文件时, *f* 的所有协议必须分别出现在 *f* 的各个声明之前, *f* 的所有声明应该在这些源文件都包含的头文件中。*requires* 子句和 *ensures* 子句分别称为函数的前条件和函数的后条件。

<i>C-declaration</i>	→ <i>/*@ function-contract */</i>
<i>function-contract</i>	→ <i>requires-clause* terminates-clause? decreases-clause? simple-clause* named-behavior* completeness-clause*</i>
<i>requires-clause</i>	→ <i>requires assert ;</i>
<i>terminates-clause</i>	→ <i>terminates assert ;</i> //用于当 <i>assert</i> 为真则终止的函数
<i>decreases-clause</i>	→ <i>decreases term (for id)?</i> //用于递归函数
<i>simple-clause</i>	→ <i>assigns-clause ensures-clause abrupt-clauses-fn</i>
<i>assigns-clause</i>	→ <i>static? assigns term (, term) ;</i>
<i>ensures-clause</i>	→ <i>ensures assert ;</i>
<i>abrupt-clauses-fn</i>	→ <i>exits assert ;</i>
<i>named-behavior</i>	→ <i>behavior id : behavior-body</i>
<i>behavior-body</i>	→ <i>assumes-clause* requires-clause* simple-clause*</i>
<i>assumes-clause</i>	→ <i>assumes assert ;</i>
<i>completeness-clause</i>	→ <i>complete behaviors (id (, id)*)?</i> <i>disjoint behaviors (id (, id)*)?</i>

图 4.1 函数协议的语法

4.1 内建构造 `\result` 和 `\old`

函数的后条件经常要引用函数的结果, 还有可能引用函数形参和所操作的全局变量在函数初始状态下的值。

1. `\result` 指称函数的返回值。`\result` 只能出现在返回类型不是 `void` 的函数的 `ensures` 子句中。

2. 内建构造 `\old(t)` 用来记住全局变量 *t* 或者函数形参 *t* 所对应的实参在函数入口点的值。例如, 在一般情况下, 函数入口有 `\old(x) == x`, 其中 *x* 是函数的形参。该断言用于验证与 *x* 对应的实参在函数调用前后的值的变化。内建构造 `\old(t)` 可理解为一个多态函数, 其中 *t* 是代表程序变量的左值的简单表达式。简单表达式是指没有赋值运算符, 也不以函数调用、条件表达式或逗号表达式作为其子表达式的表达式。例如变量 *m*、*a[0]*、*r.b* 和 *p->next* 等。`\old` 构造的变元还要注意下面几点。

- 带 `\old` 的断言 `\old(b+c) == b+c` 不被接受, 因为 `\old(b+c)` 中的 *b+c* 不是左值表达式。需要的话, 应该写成 `\old(b) + \old(c) == b + c`。类似地, 断言

`\old(\forall int i:[0..99]. a[i] == 0) == \forall int i:[0..99]. a[i] == 0`

也不被接受, 因为 `\old` 构造的变元是断言。需要的话, 应写成

`\forall int i:[0..99]. \old(a[i]) == a[i]`

- 若函数前条件的 `\old(t) == t` 的 *t* 中含有可以改变 *t* 的左值的变量, 例如 `\old(a[x])` 和 `\old(head->(next:m)->data)` 中的 *a*、*x*、*head*、*m* 和 *head->(next:m)* 的程序变量或幽灵变量, 则在该函数中不能对这些变量赋值, 因为赋值之后 *t* 的左值可能就不再等于函数入口处 *t* 的左值。

注意，若函数前条件中有断言`\forall int i:[0..99]. \old(a[i]) == a[i]`，`\old(a[i])`中虽有 `i`，但 `i` 是可以在 0 到 99 之间变化的该全称量化断言的约束变元。`i` 值的变化虽然改变 `a[i]` 的左值，但该左值总表示 `a[0]` 到 `a[99]` 这 100 个数组元素中某个数组元素的左值。因此这不是对 `i` 赋值，`i` 不属于上面所说的不能赋值的变量。类似的情况见例 7.3，例 7.3 的函数前条件中有

```
\forall int i:[0..m-1]. \old(head->(next:i)->data) == head->(next:i)->data
```

同样，其中 `i` 是全称量化断言的约束变元，`i` 的值变化，则 `head->(next:i)->data` 的左值就变化。只要 `i` 的值在 0 到 `m-1` 之间，`head->(next:i)->data` 的左值就是该链表上某个表元的 `data` 域的地址。

在`\old(t)`中，`t` 的内部无需也不允许存在外加`\old`的子项，也就是`\old`构造不必也不能嵌套使用。

3. 在一些特殊场合，会有`\old(dest[i+k]) == dest[i]`或者`\old(dest[i]) == dest[i+k]`这样的需求。例如，在例 4.5 中，在函数协议的一种命名行为中（命名行为的含义见 4.3 节），前条件和后条件中分别有

```
\forall integer i:[m-k..m]. olddest[i] == \old(olddest[i]) 和
```

```
\forall integer i:[0..m-k-1]. olddest[i] == \old(olddest[i+k])
```

对这种情况的限制是，

- 在函数前条件中，仍然必须是`\old(t) == t`；
- 而在循环不变式、程序点断言和函数后条件中，可以是`\old(t) == t ± e` 或者 `\old(t ± e) == t`。其中 `e` 只允许是简单表达式，并且 `e` 中没有程序可赋值的变量。

在例 4.5 中，`olddest[i] == \old(olddest[i+k])`中的 `k` 是逻辑变量。`i` 的值改变不会改变 `olddest[i]`和 `olddest[i+k]`两者在 `olddest` 数组中的相对位置。

4. 另外，`t` 可以是代表数组区间的项，例如`\old(id[0..al]) == id[0..al]`。对于这里的上下界表达式，也可以有符合第 3 点中要求的`± e`表达式。

在本手册中，有时为解释方便起见，在表达式 `E` 和断言 `A` 的具体形式还不清楚时，也使用`\old(E)`或`\old(A)`，其含义是 `E` 和 `A` 在函数入口状态下的值，`\old` 应用到其中的变量。在 SCSL 语言中禁止这么使用。

`\old(t)`构造在标注中的使用要注意下面几点。

- 若 `x` 是全局变量或形参，并且被 `const` 修饰，则没有必要用`\old(x)`来记住 `t` 在函数入口处的值，因为 `x` 在函数的执行过程中不会被改变，`x` 可直接出现在函数后条件中。
- 若 `x` 是指针类型的全局变量或形参，其指向的数据没有 `const` 修饰，情况与上一点就不一样了。若想验证函数执行前后，相应实参指向数据的变化，这时可以用`\old`构造来记住指针 `x` 在函数入口点所指向数据的值。例 4.5 是使用`\old`构造的一个例子，后面各章还有一些例子。

• 一般情况下 `t` 是左值表达式，唯一的例外是，`t` 可以是并非左值的 `STRING` 类型的最简单的非常量项`\string(t1, t2)`，其中 `t1` 和 `t2` 是不带函数调用的简单项。若要仔细推敲，`t1` 是`\string(t1, t2)`的左值。

• 函数前条件有某个`\old(...)`时，函数其它地方的标注才可以出现这个`\old(...)`，包括上面第 3 和第 4 点提到的略有区别的情况。

按照`\old`构造的定义，在函数入口点，断言`\old(t) == t`一定成立。在函数前条件中添加断言`\old(t) == t`，是在其他程序点验证 `t` 的当前值和 `t` 在函数入口点的值之间的关系的基础，尤其是 `t` 涉及到通过指针可访问一批数据时，用这种方式比较方便。例如，链表插入函数使得结果链表和参数链表之间的区别仅是前者在某个位置多了一个节点及其数据，见例 7.3。

后面还会介绍，存在其他方式可用来表达相同或类似的需求。

- 可用逻辑变量记住 `t` 在函数入口状态下的值，逻辑变量的介绍见 7.2 节。例子可见例

3.11, 还可见例 7.1 和例 7.2 等。采用逻辑变量的方式比采用 `\old` 构造的方式简单, 因此, `\old(t)` 一般用于 `t` 中有易变数据结构的节点指针类型的变量的情况。两种表示方式的区别与联系请见 7.2 节。

4.2 简化的函数协议

一个简单的函数协议只有简单子句, 没有命名行为, 其形式如下:

```
/*@ requires P1; ... requires Pm;  
   assigns V1; ... assigns Vn;  
   ensures Q1; ... ensures Qk;  
*/
```

该协议的语义如下:

1. 函数的调用者必须保证, 其调用状态使得性质 $P_1 \ \&\& \ \dots \ \&\& \ P_m$ 成立。 $P_1 \ \&\& \ \dots \ \&\& \ P_m$ 即常说的函数前条件 (在介绍带命名行为的协议后还会充实)。

就程序变量而言, 有形参和全局变量以及由它们开始的访问路径所达的变量可以出现在 $P_i (1 \leq i \leq m)$ 中。对于指针类型的形参和全局变量, 调用者必须保证指针型的实参和全局变量相互之间不会形成别名, 除非这种别名是从函数前条件可以推导的。

2. 被调用函数必须保证, 其返回状态使得性质 $Q_1 \ \&\& \ \dots \ \&\& \ Q_k$ 成立, $Q_1 \ \&\& \ \dots \ \&\& \ Q_k$ 即常说的函数后条件。

就程序变量而言, 局部变量和未带 `const` 修饰符的形参不能出现在 $Q_i (1 \leq i \leq k)$ 中。

函数前后条件中可以出现第 7 章介绍的逻辑变量, 也可以出现第 9 章介绍的幽灵形参和幽灵全局变量。

3. V_1, \dots, V_n 是指可能被函数赋值的非局部变量。它分成两种情况, 一种是属于可赋值的外部变量 (包括由可访问外部变量开始的访问路径所代表的变量, 因而有可能是堆变量), 另一种是属于通过指针型形参开始的访问路径所代表的可赋值变量, 下面分别介绍。对于堆指针指向的易变数据结构, 存在一些特殊性, 在下面 (1) 和 (2) 之后单独介绍。

(1) V_1, \dots, V_n 都是代表外部变量 (静态外部变量的情况稍后解释) 的访问路径 (访问路径中出现的变量必须都是外部变量)。在函数前条件成立的情况下, 这些访问路径所代表的外部变量 (包括以它们开始的访问路径所代表的变量, 整个这个变量集称为 S), 在该函数的执行过程中可能直接被赋值 (指在该函数中经某个赋值操作被赋值) 或间接被赋值 (指在该函数中经某个函数调用被赋值)。任何以外部变量开始的访问路径所代表的变量, 只要不属于集合 S , 则在该函数的执行过程中没有直接被赋值。言下之意, S 集合超出运行时直接赋值的外部变量集是允许的, 只要它不影响代码的验证。

一种特殊情况是, V_1, \dots, V_n 中可以出现外部数组名, 外部数组名相当于不能被赋值的变量。数组名本身的 S 集合包含该数组的所有元素, 也包含数组名。由于数组名是常量, 代码中不可能出现对它的赋值, 因此数组名可以出现在 S 集合中, 只要不影响验证。注意, 对于多维数组, 例如 $A[100][200]$, $A[i]$ ($0 \leq i < 100$) 都是一维数组名, 同样不可能被赋值。

另一种特殊情况是, V_1, \dots, V_n 中也可以出现指向静态区的外部指针名 p , `assigns p` 和 `assigns *p(或 p[e1], p[e1..e2])` 的 p 和 `*p(或 p[e1], p[e1..e2])` 分别表示对 p 赋值和对 p 在函数入口的初值指向的对象赋值。若按先前的约定, `assigns p` 表示由 p 开始的所有访问路径, 则没有方式可表示仅能对 p 赋值的情况。需要注意, 像 `assigns p, *p(或 p[e1], p[e1..e2])` 这样, p 和 `*p(或 p[e1], p[e1..e2])` 同时出现在一个函数的 `assigns` 子句的情况要避免, 应该写成 `assigns p, *\old(p)` (或 `\old(p)[e1], \old(p)[e1..e2]`), 若希望后者是对 p 在函数入口的初值指向的对象赋值的话。

若函数 f 的执行过程中修改了它的定义所在文件的静态外部变量 x , 则 x 可能出现在 f 的 `assigns` 子句中, 分成下列三种情况。

- 若 f 函数仅能被本文件的函数调用, f 修改的静态外部变量和外部变量列在同一个 assigns 子句中。

- 若 f 函数仅能被其它文件的函数 g 调用, 则 f 的协议必须出现在 f 所在源文件和 g 所在源文件共享的某个头文件中。但是, f 函数修改的静态外部变量不必列在 f 函数协议的 assigns 子句中。因为 g 函数协议上根本不会出现 f 函数所修改的静态外部变量。

- 若 f 函数能被本文件的函数调用, 也能被其它文件的函数 g 调用, 则 f 的协议也必须出现在 f 所在源文件和 g 所在源文件共享的某个头文件中。这时 f 函数的 assigns 子句需要区分修改的外部变量和静态外部变量, 其中静态外部变量列在 static assigns 之后。例如:

```
assigns v1, v2;           // v1 和 v2 是外部变量。  
static assigns x;        // x 是静态外部变量。
```

这么做的原因是避免在验证 g 中调用 f 的语句时, 因找不到这些静态外部变量的声明而报错。

对 V_1, \dots, V_n 还需进一步细化解释:

- 所说的访问路径都由变量标识符 (包括幽灵变量, 也包括逻辑变量) 和数组名开始。在此谈论访问路径时, 把访问路径 *p... 和 &b... 视同为 p*... 和 b&..., 即都视为由变量标识符开始的访问路径, 以简化描述, 同时也表明 *p... 和 &b... 不包括单独的 p 和 b。

所说的访问路径包括像 a[e] 和 a[e1..e2] 这样代表一个数组元素 a[e] 和数组元素集 {a[e1], ..., a[e2]} 的访问路径。若访问路径中的指针 a 会被赋值, 而仍希望 a[e] 等的 a 表示 a 在函数入口的初值的话, 则需要使用 \old(a) 描述, 例如 \old(a)[e]。这里的 e、e1 和 e2 只能是简单表达式, 其中若有变量 x, 则 x 应是外部变量。若使用的是 \old(x), 则 x 在本函数协议的 S 集合中, 否则 x 不在该 S 集合中。

(2) V_1, \dots, V_n 中有部分是由指针型形参开始的访问路径所代表的可赋值变量, 以其中的 V_i ($1 \leq i \leq n$) 为例来解释其特点。

- 一般情况下, V_i 一定不会是单纯的指针型形参, 因为对形参赋值不会改变对应的实参。当 V_i 是有效指针时, 它一定代表指针型形参指向的对象 (* V_i 或 $V_i \rightarrow next$ 等), 并且形参声明中是允许对指向对象赋值的。

特殊情况指 V_i 是堆指针, 并且 V_i 指向的对象会在代码中被 free(V_i) 释放, 导致对应实参变成悬空指针。把这种情况留到稍后专门解释堆指针时介绍。

- 对 V_i 赋值修改的是什么变量, 由调用的实参决定的。虽然实参可能不是可访问的外部变量, 但也肯定不是该函数的局部变量, 因此 V_i 也属于上面所讨论的 S 集合。函数前后条件中对这类形参的描述, 实际上就是对实参的描述。

- 若 V_i 是上面所提到的 a[e] 和 a[e1..e2] 这样的形式, 并且 a 是形参, 若可以对 a 赋值, 而仍希望 a[e] 等的 a 表示 a 在函数入口的初值的话, 则需要使用 \old(a) 描述, 例如 \old(a)[e], 以保证都是用 a 在函数入口的值来计算下标变量的地址。若 e、e1 和 e2 中有变量 x, 则一般使用的都是 \old(x), 否则 x 不在本函数协议的 S 集合中。

对 V_1, \dots, V_n 还有两点需要解释。

(1) 对于递归函数 f, 由于 f 可能被调用多次。若其 assigns 子句中出现外部变量或是指针型形参开始的访问路径所代表的可赋值变量, 例如 a[\old(x)..10000], 其中 x 是外部变量, 那么 f 每次被调用时, f 可修改的 a 数组元素必须落在由调用时 x 的值决定的区间 a[\old(x)..10000] 中, 即 f 每次被调用时, a 数组的可赋值区间可能不一样。

(2) 若程序标注中声明了幽灵数组 bitstream[], 且函数协议的 assigns 子句出现幽灵数组区间 bitstream[0..\infinity], 则表示在这个函数中可对任意的 bitstream[i] ($i \geq 0$) 赋值 (见第 9 章)。

对于堆指针, 由于它指向的易变数据结构是动态变化的, 因此对出现在 assigns 子句中

的堆指针相关访问路径有一些特殊考虑。

(1) 只有局部指针指向的易变数据结构不必出现在 `assigns` 子句中。

(2) 对于出现在 `assigns` 子句中的由全局指针变量开始的访问路径 `path`，函数不能修改 `path` 的真前缀中的任何指针（包括不能释放这些指针指向的对象）。`path` 或 `path` 是其前缀的访问路径都属于 `S`，不必关心这些访问路径在函数入口点所表示的变量和在赋值点所表示的变量是否相同。

(3) 对于出现在 `assigns` 子句中的由函数的指针形参开始的访问路径 `path`，若函数代码把 `path` 指向的节点释放而导致对应实参指针的值虽未变但性质变成悬空指针，则 `path` 必须单独出现在 `assigns` 子句中，以体现对应实参值的性质改变。不是这种特殊情况，则应是 `*path` 出现在 `assigns` 子句中。

(4) `assigns` 子句中的访问路径中可以出现折迭域，折迭域的重复次数表达式中的变量不能赋值。

(5) `assigns` 子句中的访问路径中不可以出现 `\old` 约束的指针路径。

要求在 `assigns` 子句中列出相关变量，是为了便于：

- 掌握函数调用之前程序点的哪些断言不受调用的影响，可以延续到调用点之后的程序点。

- 以文件为单位的分模块验证。

下面用几个简单的例子来说明怎样使用函数协议。

例 4.1 下面是计算平方根（近似到整数）的函数声明及协议。

```
/*@ requires x >= 0;
   ensures  \result >= 0;
   ensures  \result * \result <= x;
   ensures  x < (\result + 1) * (\result + 1);
*/
int isqrt(const int x);
```

该协议表示，`isqrt` 函数必须用一个非负实参调用，并且返回值满足对应的 3 个 `ensures` 子句的合取。由于在函数中不会对 `x` 赋值，形参 `x` 可以出现在函数后条件中。

对本例子，若形参 `x` 在函数中有可能被赋值，则 `x` 不能出现在后条件中，需把协议写成

```
/*@ requires x >= 0 && \old(x) == x;
   ensures  \result >= 0;
   ensures  \result * \result <= \old(x);
   ensures  \old(x) < (\result + 1) * (\result + 1);
*/
int isqrt(int x);
```

在 `ensures` 子句中用 `\old(x)` 来引用 `x` 的初值（即实参的值），同时在 `requires` 子句中有断言 `\old(x) == x`。实参的值在 `isqrt` 执行时保持不变，即实参在调用前后的状态中值相同。□

例 4.2 下面的函数协议指出，函数 `incrstar` 将实参指针所指向数据的值增 1。

```
#define MAX_VALUE 1000
/*@ requires p != \null && \old(*p) == *p && *p < MAX_VALUE;
   assigns *p;
   ensures *p == \old(*p) + 1; */
void incrstar( int * const p) {
    *p = *p + 1;
}
```

该协议说，调用者的实参必须是不等于 NULL 的指针（若 p 是堆指针，则实参不是悬空指针由安全 C 语言的形状系统保证）。ensures 子句指出，实参指针所指向数据的值增 1。

注意 C 中 int* const p 和 const int *p 的区别，前者指 p 不能被赋值，后者指 p 指向的整型单元不能被重新赋值。还有，若有类型定义 typedef long array[100]，函数声明 void f(const array a)和 void f(array const a)的效果一样，都等价于 void f(const long* a)，即不能对 a 指向的区间赋值，但可以对 a 赋值。

p 虽不是全局变量，但它是指针类型的形参，并且该函数修改对应实参所指向的数据，因此必须把*p 列入 assigns 子句。

本例的函数前后条件也可以改成：

```
#define MAX_VALUE 1000
/*@ logic int oldValue;
/*@ requires p != \null && oldValue == *p && *p < MAX_VALUE;
    assigns *p;
    ensures *p == oldValue + 1;
*/
void incrstar( int * const p) {
    *p = *p + 1;
}
```

两个协议的区别是，前一个用\old(*p)来记住*p 在函数入口状态下的值。后条件用断言 *p == \old(*p)+1 来表示 p 指向对象的值增 1。后一个用实参 p 指向的对象代换逻辑变量 oldValue，使得 oldValue == *p。后条件中用*p == oldValue+1 来表示 p 指向对象的值增 1。□

例 4.3 若有三个函数的一个源文件如下，其中的 assigns 子句都给出了相应函数直接和间接赋值的全局变量。

```
int a=0, b=0, c=0;
/*@ requires c == 0; assigns c; ensures c == 1; */
void f2() { c = c + 1; }
/*@ requires b == 0 && c == 0; assigns b, c; ensures b == 1 && c == 1; */
void f1() { f2(); b = b + 1; }
/*@ requires a == 0 && b == 0 && c == 0;
    assigns a, b, c;
    ensures a == 1 && b == 1 && c == 1; */
void test() { f1(); a = a + 1; } □
```

例 4.4 若 main、f1 和 f2 三个函数分处于不同的文件中，例如：

- 文件 d.h:

```
extern int b, c;
/*@ requires b == 0 && c == 0; assigns b, c; ensures b == 1 && c == 1; */
void f1();
```

- 文件 file1.c:

```
#include "d.h"
int a;
/*@ requires a == 0 && b == 0 && c == 0; assigns a, b, c;
    ensures a == 1 && b == 1 && c == 1; */
void main() { f1(); a = a + 1; }
```

- 文件 file2.c:

```

#include "d.h"
int b, c;
/*@ requires c == 0; assigns c; ensures c == 1; */
void f2() { c = c + 1; }
void f1() { f2(); b = b + 1; } // 函数 f1 的协议在头文件 d.h 中。

```

可以看出，main 函数虽然只对 a 赋值，但它调用 f1，f1 的协议必须出现在共享的头文件 d.h 中。没有这样的头文件，main 的协议没法写。即使写得出来，main 的验证也完成不了，因为找不到 f1 的协议，完成不了对 f1 调用语句的验证。 □

例 4.2、例 4.3 和例 4.4 用大篇幅介绍了 assigns 子句的使用要注意的事项，下面继续有关函数协议的其他事情并给出一些例子。

使用多个 requires、assigns 和 ensures 子句仅改进可读性，因为本节一开始介绍的函数协议等价于下面更简单的函数协议

```

/*@ requires P1 && ... && Pm;
    assigns V1, ..., Vn;
    ensures Q1 && ... && Qk;
*/

```

如果函数协议中无 requires 子句，意味着它缺省到 true。若无 ensures 子句，情况类似。若无 assigns 子句，则表示函数不修改任何以全局变量或指针型函数形参的名字开头的左值表达式所表示的变量。

需要注意的是，函数协议要把函数体中所有的出口（使用 return 的出口以及未用 return 的函数出口）都考虑在内。调用 exit 函数的异常终止，在 4.4 节讨论。

4.3 带命名行为的函数协议

函数协议的一般形式可以包含若干命名行为，这些行为的名称必须有区别。下面一般都用 2 个命名行为来解释。例如：

```

/*@ requires P; // 各命名行为的共享部分
    assigns V;
    ensures Q;
    exits S;
    behavior b1: // 命名行为 b1
        assumes A1;
        requires R1;
        assigns V1;
        exits S1;
        ensures Q1;
    behavior b2: // 命名行为 b2
        assumes A2;
        requires R2;
        assigns V2;
        exits S2;
    ensures Q2; */

```

其中 exits 子句的解释见 4.4 节。就协议中允许出现的程序变量而言，此处规定与 4.2 节一样。

在有命名行为的情况下，函数入口的 `\old(t) == t` 断言要写在命名行为共享部分的 requires 子句中，而不要分别写在各命名行为的 requires 子句中。

assumes 子句仅用于命名行为，并且在 requires 子句之前。assumes 子句在语法上有点限制。

(1) 断言 A_1 和 A_2 中会出现变量，例如形参 x 。

- 若 x 在函数中有可能被赋值，则必须写成 $\text{old}(x)$ ，以表示 assumes 子句用的是 x 在函数入口处的值，通常用的是断言 $\text{old}(x) == x$ 。

- 若代码中不允许函数对 x 赋值的修饰，则在 assumes 子句可直接写 x 。

(2) 断言 A_1 和 A_2 不含谓词、归纳谓词或逻辑函数的应用。

在多命名行为场合，assigns 子句在语法上也有限制。 V 是在各命名行为中都可能被修改的非局部变量， V_1 和 V_2 是仅可能分别会被命名行为 b_1 和 b_2 修改的非局部变量。若 V 中的变量集是 V_1 和 V_2 中变量集的真子集，则是一种错误的 assigns 子句描述。反之，则 V 中还有变量未出现在 V_1 和 V_2 中，则命名行为 b_1 和 b_2 的行为并不完备(完备的含义见下面)，即相应函数存在着执行路径，对 V_1 和 V_2 以外的非局部变量赋值。

通常，还可以把出现在本命名行为的 assigns 子句而未出现在对方 assigns 子句的变量，例如 x ，用 $\text{old}(x) == x$ 列在对方命名行为的前后条件中，以确保在不该修改的地方没有修改。这种方式并非必要，参见下面的例 4.6。

带命名行为的函数协议使用场合如下：

1. P 是任何调用都应满足的前条件。任何满足 P 的调用在调用点后能得到性质 Q 。

2. $A_1 \ \&\& \ R_1$ 和 $A_2 \ \&\& \ R_2$ 分别是某些额外性质。若调用满足 P 以外，还满足 $A_i \ \&\& \ R_i$ ($1 \leq i \leq 2$)，则在调用点后除了得到 Q 以外，还能得到 Q_1 和/或 Q_2 。

3. $A_i \ \&\& \ R_i$ ($1 \leq i \leq 2$) 还有这样的特点，若 A_i 为真则 R_i 为真，即 $A_i \implies R_i$ 。这是把 A_i 单独列在 assumes 子句的原因。

带命名行为协议有如下特点：

1. 函数的调用者必须保证，其调用状态使得性质 $P \ \&\& \ (A_1 \implies R_1) \ \&\& \ (A_2 \implies R_2)$ 成立。

2. 被调用函数必须保证，其返回状态使得性质 $Q \ \&\& \ (A_1 \implies Q_1) \ \&\& \ (A_2 \implies Q_2)$ 成立，也就是使得函数后条件成立。

3. 对于 $i = 1, 2$ ，若调用者的调用状态能使 A_i 成立，则列在 V_j ($j \neq i$) 而未列在 V_i 中的全局变量和以其名字开头的左值表达式所表示的变量，在调用之后的状态中保持不变。

4. 带命名行为的协议相当于把函数前后条件分成了若干种命名情况，改进可读性。

5. 命名行为中的 assumes 子句缺省时也默认为 true 。

若 V_1 和 V_2 都是空集，上述协议可以等价于下面的简化协议：

```
/*@ requires P && (A1 ==> R1) && (A2 ==> R2);
   assigns V;
   exits S || A1 && S1 || A2 && S2;
   ensures Q && (A1 ==> Q1) && (A2 ==> Q2); */
```

在 A_1 和 A_2 不同时为真也不同时为假并且 V_1 和 V_2 为空的情况下，上述带命名行为的协议等价于下面的简化协议：

```
/*@ requires P && (A1 && R1 || A2 && R2);
   assigns V;
   exits S || A1 && S1 || A2 && S2;
   ensures Q && (A1 && Q1 || A2 && Q2); */
```

简化协议 `/*@ requires P; assigns V; ensures Q; */`

等价于如下单个命名行为：

```
/*@ requires P;
   behavior <any name>:
```

```

    assumes \true;
    assigns V;
    ensures Q; */

```

下面介绍命名行为的完备性。在带命名行为的函数协议中，并不要求各 A_i 的析取为真，即不强求提供行为全集。若想确保所有行为构成全集，则需在协议中增加 `complete` 子句：

```

/*@ requires P;
...
complete behaviors b1, ..., bn; */

```

其含义是，行为 b_1, \dots, b_n 的集合是完备的，即

$$P \implies (A_1 \parallel \dots \parallel A_n)$$

成立，其中 P 是该协议全局的 `requires`。上述子句的简化版本如下：

```

/*@ requires P;
...
complete behaviors; */

```

它表示协议的所有行为都需要考虑在内。

若行为集是完备的，当调用点的断言蕴涵前条件时，则后条件中 $Q \ \&\& \ (\text{old}(A_1) \implies Q_1) \ \&\& \ \dots \ \&\& \ (\text{old}(A_n) \implies Q_n)$ 中，除了 Q 以外，至少还有一个 Q_i 在调用点之后成立。

类似地，对于带多个命名行为的协议，也没有要求两个不同的行为不相交。如果希望各行为两两不相交，则需要增加 `disjoint` 子句：

```

/*@ requires P;
...
disjoint behaviors b1, ..., bn; */

```

其含义是，对任意不相同的 i 和 j ($1 \leq i, j \leq n$)，

$$P \implies \neg(A_i \ \&\& \ A_j)$$

成立，其中 P 是该协议全局的 `requires`。该子句的简化版本是

```

/*@ ...
disjoint behaviors; */

```

同样，它表示协议的所有行为都需要考虑在内。

若行为集合中的行为两两不相交，当调用点的断言蕴涵前条件时，则后条件中 $Q \ \&\& \ (\text{old}(A_1) \implies Q_1) \ \&\& \ \dots \ \&\& \ (\text{old}(A_n) \implies Q_n)$ 中，除 Q 以外，至多一个 Q_i 在调用点之后成立。

若行为集完备且两两不相交（即正交），则后条件 $Q \ \&\& \ (\text{old}(A_1) \implies Q_1) \ \&\& \ \dots \ \&\& \ (\text{old}(A_n) \implies Q_n)$ 中，正好仅某个 Q_i 在调用点之后成立，调用点后的断言就可简化为 $Q \ \&\& \ Q_i$ 。

例 4.5 图 4.2 是早年串复制库函数 `strcpy(char* dest, char* src)` 的代码和标注。该协议的两个命名行为有特点： $(\neg(\text{oldsrc} == \text{olddest}+k) \parallel (\text{oldsrc} == \text{olddest}+k)) == \text{true}$ 并且 $(\neg(\text{oldsrc} == \text{olddest}+k) \ \&\& \ (\text{oldsrc} == \text{olddest}+k)) == \text{false}$ ，也就是这两个命名行为是正交的。它们分别适用于把 `src` 指向的串向 `dest` 指向的另一块空间复制和删去 `dest` 指向串的前 k 个字符。

本例两种命名行为 `non_overlap` 和 `overlap` 的 `assigns` 子句没有区别，但它们的实际含义不一样。前者仅对 `dest` 数组赋值，并不对 `src` 数组赋值；后者对 `dest` 数组赋值也就是对 `src` 数组赋值，所以都看成对 `dest` 数组的赋值。由于 `dest` 在函数中是可以赋值的，所以改用 `assigns \old(dest)[0..\length(\old(dest)) - 1]`，统一在公共的 `assigns` 子句中。本例中增加幽灵变量 n 的声明和对 n 赋值的幽灵语句。引入幽灵变量 n 有助于写出循环不变式。它们出现在标注中，不会影响程序的编译和运行。 □

```

//早期的串复制库函数 strcpy(char* dest, char* src)的验证
// 这个协议适用于 dest 和 src 指向不同的数据块和指向同一个数据块两种情况
//@ logic char *olddest, *oldsrc;
//@ logic int m, k;
/*@
requires \offset(dest)==0 && \length(dest) > 1 && olddest==dest && \old(dest) == dest && oldsrc==src &&
    \offset(src)>=0&& \length(src)>1 && \old(olddest[0..\length(olddest)-1])==olddest[0..\length(olddest)-1];
assigns \old(dest)[0..\length(\old(dest)) -1];
ensures \result == olddest;

behavior non_overlap:
    assumes !(oldsrc == olddest+k); //根据 SCSL 手册 3.7 节,可知 oldsrc 和 olddest 指向不同的数据块。
    requires \is_pstring(src, m) && \length(dest) > m ;
    ensures \is_pstring(\result, m) && \is_pstring(oldsrc, m) &&
        \string(\result, m) == \string(oldsrc, m);

behavior overlap:
    assumes oldsrc == olddest+k;
    requires \is_pstring(dest, m) && 0 <= k <= m && \is_pstring(src, m-k) && src == dest + k;
    ensures \is_pstring(olddest, m-k) && 0 <= k <= m &&
        (\forallall int i:[0..m-k].olddest[i] == \old(olddest[i+k])) &&
        olddest[m-k+1..\length(olddest)-1] == \old(olddest[m-k+1..\length(olddest)-1]);
complete behaviors;
disjoint behaviors; */
char* strcpyOld(char* dest, char* src){
    char* p;
    //@ ghost int n;
    p = dest;
    *dest = *src;
    //@ ghost n = 0;
    /*@ loop invariant
        !(oldsrc == olddest + k) && 0 <= n <= m && dest == olddest + n && src == oldsrc + n &&
            p == olddest && \old(dest) == olddest && \offset(olddest) == 0 &&
            \is_string(olddest, n) && \string(olddest, n) == \string(oldsrc, n) &&
            olddest[n+1..\length(olddest)-1] == \old(olddest[n+1..\length(olddest)-1]) &&
            \is_pstring(oldsrc, m) && \length(olddest) > m && *dest == *src ||
            src == dest + k && 0 <= k <= m && 0 <= n <= m-k && dest == olddest + n &&
            p == olddest && \is_pstring(src, m-k-n) && oldsrc == olddest + k && \old(dest)==olddest &&
            (\forallall int i:[0..n].olddest[i] == \old(olddest[i+k])) &&
            (\forallall int i:[n+1..\length(olddest)-1].olddest[i] == \old(olddest[i])) &&
            (*src != 0 && \is_pstring(olddest, m) && n < m-k ||
            *src == 0 && \is_pstring(olddest, m-k) && n == m - k) && \offset(olddest) == 0; */
    while(*src != 0){
        dest = dest + 1;
        src = src + 1;
        *dest = *src;
        //@ ghost n = n + 1;
    }
    return p;
}

```

图 4.2 早期的库函数 strcpy 的协议

例 4.6 图 4.3 的函数表明 `assigns` 子句分属不同命名行为的重要性。函数 `f` 修改指针型形参 `p` 和 `q` 指向的对象。由于函数前条件中没有 `p` 和 `q` 指向同一数据块的断言，因此根据形参 `n` 的值，`f` 的执行修改 `p` 或 `q` 指向的对象，但不会修改 `p` 和 `q` 共同指向的对象，因为不存在这种情况。两种命名行为有各自的 `assigns` 子句，精确指明各自修改的非局部变量。本例展示了不同的命名行为有自己 `assigns` 子句的重要性。 □

在同一个协议中，可以有多个 `complete` 和 `disjoint` 集合。例如，`P` 是 `1 <= a <= 2 && 1 <= b <= 2`，其中 `a` 和 `b` 都是 `const int` 类型的变量，4 个命名行为的 `assumes` 子句的断言 `A1`、`A2`、`A3` 和 `A4` 分别是 `a == 1`，`a == 2`，`b == 1` 和 `b == 2`，则 `A1` 和 `A2` 及 `A3` 和 `A4` 两个集合都是正交的。

```

/*@ behavior p_changed:
    assumes n > 0;
    requires p != \null;
    assigns *p;
    ensures *p == n;
behavior q_changed:
    assumes n <= 0;
    requires q != \null;
    assigns *q;
    ensures *q == n;
*/
void f(const int n, int* const p, int* const q) {
    if (n > 0) *p = n; else *q = n;
}

```

图 4.3 命名行为各带 `assigns` 子句

4.4 异常终止

函数协议的 `ensures` 子句并没有包括对函数异常终止状态（调用 `exit` 之后的状态）的约束。因此对于有 `exit` 调用的函数，函数协议中必须包括 `exits` 子句；若无 `exit` 调用，则无须包括该子句。若 `exits` 子句有堆指针断言，则这部分断言只能是 8.3.1 节提到的简单指针断言或者 `\dangling(p)`，其中 `p` 是堆指针。

例 4.7 在图 4.4 中，前一部分是库函数 `exit` 的规范。在 `ensures` 子句中，它什么也不保证。在 `exits` 子句中，

`\exit_status` 是内建的整型变量，该子句给出的异常标识就是调用 `exit` 函数的实参。

图 4.4 的其余部分是可能调用 `exit` 函数的 `mayExit` 函数及其规范，位于其前面的 `status` 是全局变量。`mayExit` 函数的异常终止 `exits` 子句告知出现异常的原因以及调用 `exit` 函数所用的参数等。

图 4.4 的函数 `mayExit` 的协议并不完备，因为该协议允许在不执行 `exit` 时也修改变量 `status`。使用命名行为，可以区分正常和异常情况，`mayExit` 函数的相应规范见图 4.5。 □

若一个协议的命名行为中有 `exits` 子句，当把该协议整理成简化协议时，各协议中的 `exits` 子句的析取作为简化协议的 `exits` 子句。

```

int status = 0;
/*@ ensures \false;
    exits \exit_status == status; */
void exit(const int status);
/*@ assigns status;
    exits cond == 0 && \exit_status == 1 && status == val;
*/
void mayExit(const int cond, const int val) {
    if (cond == 0) {status = val; exit(1);}
}

```

图 4.4 异常终止的一个例子

```

int status = 0;
/*@ behavior no_exit:
    assumes cond != 0;
    ensures \false;
behavior no_return:
    assumes cond == 0;
    assigns status;
    exits \exit_status == 1 && status == val;
    ensures \false;
*/
void mayExit(const int cond, const int val);

```

图 4.5 异常终止的另一个例子

4.5 多协议

C 的函数仅能定义一次，但可以声明多次。可以对函数的每个声明标注一个协议。这时把这个函数看成是有多个协议的函数。协议有多个命名行为的函数不能称为多协议函数，它们不是一回事。

在多协议的情况下，若调用点的状态满足其中部分协议，则调用之后的状态满足这一部分协议的后条件。

对于有完备且两两不相交性质的 n 个命名行为的函数协议，可以拆成 n 个函数协议，相互正交的性质分别属不同的协议。这时最多只有一个协议能得到验证。

例 4.8 图 4.6 是树堆 (*treap*) 中两个协议的右旋函数。树堆是一种有额外性质的 BST (二叉排序树), BST 在安全 C 语言使用手册第 2 章中已经介绍, 在本手册的后面也会用到。BST 的各个节点增加一个随机附加域使得它还满足堆 (见例 5.2) 的性质, 就构成这里定义的茶堆。

下面先介绍图 4.6 的代码用到的二叉树节点的数据类型。

```
typedef struct node { // 其中 priority 域就是随机附加域。
    int data; int priority; struct node* l; struct node* r;
}Node;
//@ shape l, r : binary_tree;
```

再给出定义 BST 和树堆所需要的归纳谓词。但验证中需要用到相关引理没有给出, 可从完整的树堆例中寻找。

```
inductive gt(integer x, Node* p) =
    p == \null || p != \null && x > p->data && gt(x, p->l) && gt(x, p->r);
inductive lt(integer x, Node* p) =
    p == \null || p != \null && x < p->data && lt(x, p->l) && lt(x, p->r);
inductive BST(Node* p) =
    p == \null ||
    p != \null && BST(p->l) && BST(p->r) &&
        gt(p->data, p->l) && lt(p->data, p->r);
inductive priority_le(integer x, Node* p) =
    p == \null ||
    p != \null && x <= p->priority && priority_le(x, p->l) && priority_le(x, p->r);
inductive heap_tree(Node* p) = // 带堆性质的一般二叉树,
    p == \null || // heap_tree(p) && BST(p)才构成这里所说的 treap
    p != \null && heap_tree(p->l) && heap_tree(p->r) &&
        priority_le(p->priority, p->l) && priority_le(p->priority, p->r);
// 上一行是随机附加域需要满足的性质
```

图 4.6 的右旋函数仅在两种不同场合下使用。其一是在树堆上插入节点, 若是在左子树插入, 则插入后有可能出现右旋操作。其二是在树堆上删除节点, 若是在左子树上删除, 则删除后可能出现右旋操作。了解插入和删除函数的协议之后, 才能理解该右旋函数的协议。

比较这两个协议, 可以看出, 两者之间的前条件和后条件都有很大差异。从前条件看, 它们分别都是针对二叉树的某种特殊情况进行的右旋, 这两种情况并不完备, 这是显然的。但是它们有可能相交, 因为它们有关 BST 的性质一致, 有关 heap_tree 的性质也并无矛盾。只是正好只有这两种特殊调用场合时, 它们不会同时成立。这时若想写成多命名行为会比较困难, 并且不易理解, 不如多协议来得清楚。 □

```

/*@ logic Node* oldt;
/*@ logic integer x, y, z;
/*@
requires    // 这个协议用于插入函数，在插入左子树后可能出现的右旋。
    oldt == t && BST(t) && gt(y, t) && lt(z, t) && x <= t->priority && heap_tree(t->l) &&
    heap_tree(t->r) && priority_le(t->priority, t->r) && t->l->priority < t->priority &&
    priority_le(t->priority, t->l->l) && priority_le(t->priority, t->l->r);
assigns *t;
ensures
    oldt == \result->r && BST(\result) && gt(y, \result) && lt(z, \result) &&
    heap_tree(\result) && priority_le(x, \result->l) && priority_le(x, \result->r);
*/
Node* rRotate(Node* t);
/*@
requires    // 这个协议用于删除函数，在删除左子树后可能出现的右旋。
    t != \null && t->l != \null && oldt == t && BST(t) && gt(y, t) && lt(z, t) && heap_tree(t->l) &&
    heap_tree(t->r) && priority_le(x, t->l) && priority_le(x, t->r) && t->l->priority < t->r->priority;
assigns *t;
ensures
    \result != \null && \result->r != \null && oldt == \result->r && BST(\result) && gt(y, \result) &&
    lt(z, \result) && priority_le(\result->priority, \result->l) && priority_le(\result->priority, \result->r->l) &&
    priority_le(\result->priority, \result->r->r) && x <= \result->priority && priority_le(x, \result->l) &&
    priority_le(x, \result->r->l) && priority_le(x, \result->r->r) && heap_tree(\result->l) &&
    heap_tree(\result->r->l) && heap_tree(\result->r->r);
*/
Node* rRotate(Node* t){
    // 对以*t为根的树作右旋处理，处理之后t指向新的树根节点，它是原来的左子树根节点
    Node* lc;
    lc = t->l;    t->l = lc->r;    lc->r = t;    t = lc;    return t;
}

```

图 4.6 面向树堆的有两个协议的右旋函数

4.6 缺省协议和函数指针变量的协议

先前已经提到，函数可以有缺少某些子句的协议。例如，`requires` 子句和 `ensures` 子句缺省时都看成 `\true`。也可以没有为函数书写协议，这表示该函数协议中的 `requires`、`assigns` 和 `ensures` 都使用默认情况，而不是该函数没有协议。

对于函数指针变量，在它们的声明处，无需用标注给出它们的协议，因为这时尚不知它们代表哪个函数。

4.7 main 函数的前条件

`main` 函数执行的初始状态，不仅包括了它所在 C 源文件确定的外部变量、静态外部变量和静态局部变量的初值，也包括了将与该文件一起连接的其它 C 源文件所确定的这三类变量的初值。这些文件中的这三类变量，若在声明时置初值的话，则都是在程序执行之前就静态确定的，它们出现在调用 `main` 函数时的程序状态中是合理的。

同其他函数一样，`main` 函数的前条件按理应由程序员提供，程序员必须把与验证 `main` 函数有关的初值断言列在 `main` 函数的前条件中，程序员还要关注 `main` 函数可能有的形参 `int`

argc 和 char* argv[]。

main 函数有形参 argc 和 argv 时，main 函数的前条件就自动有下面的断言：

```
argc > 0 && \length(argv) == argc &&  
(forall integer i:[0..argc-1]. \is_pstring(argv[i], \strlen(argv[i])) && \offset(argv[i]==0)
```

这是操作系统根据命令行参数得出的 argc 和 argv 满足的性质。程序员不必在 main 的前条件中写出上述断言，验证系统在验证 main 函数时会自动添加。

从验证的角度，操作系统调用 main 函数的调用点可理解为，与 argc 和 argv 相对应的实参具有上述断言描述的性质。若被验证程序关心 argc 和 argv 更多的性质，例如 argc 的值区间或具体值，某个字符串 argv[i] (1 ≤ i ≤ argc-1) 的长度或者其中某些位置应是某些具体字符等，这些性质不能写在函数前条件中，因为它们无法在程序运行前得到验证。程序员只有在 main 函数中，用代码来检查 argc 和 argv 是否具有这些性质，并且给出不满足时的合适处置代码，才能保证程序的验证是完整的。

main 函数协议的其它部分照常仍由程序员提供，例如 assigns 子句和 exits 子句等。

4.8 对形参或返回值有 void* 或 size_t 类型的函数协议的特别要求

若函数 f 有形参 void* x，则调用 f 时，对应形参 x 的实参必须是具体类型的指针而不是 void 类型的指针。为确保这一点，函数的前条件中必须有对*x 对应实参类型限定的断言。可参看图 4.7 中的函数 comp_int 和 comp_str 的协议。图 3.2 的断言语法中的断言 \typeof(C-expr) == C-type-name 是一个特殊的内建断言，\typeof(C-expr) 是获取 C 表达式 C-expr 的类型，C-type-name 是 C 程序中定义的类型名。

在函数 f 中，形参 x 必须先强制成具体类型的指针后才能使用，该具体类型必须和函数前条件中对*x 的类型限定一致。并且 x 的类型被强制后，不得再次把它强制成其它类型。

再看图 4.7 中对快速排序库函数 qsort 的调用。qsort 的函数原型如下：

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

其中有 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。两个调用 qsort(a, 10, sizeof(int), comp_int) 和 qsort(s, 5, sizeof(s[0]), comp_str) 都要执行。可以把 qsort 理解成是一个有多协议的函数，每个协议针对*p、*q 和*t 可取同一种具体类型。

针对第一个调用的协议如下：

```
/*@ //这是 qsort 第 4 个形参（函数指针）的协议（只列出与这里的介绍相关的断言）。  
是 qsort 协议前条件的一部分。
```

```
requires \typeof(*q) == int && \typeof(*t) == int;  
ensures \result < 0 ==> *q < *t || \result == 0 ==> *q == *t || \result > 0 ==> *q > *t; /*/  
/*@ //这是 qsort 协议的主要断言，前条件中有关第 4 个形参应有性质见上面的协议  
requires \typeof(*p) == int && \length(p) == len && size == sizeof(int);  
ensures forall integer j:[0..len-2]. p[j] < p[j+1]; /*/
```

针对第二个调用的协议如下：

```
/*@ //同样，这是 qsort 第 4 个形参（函数指针）的协议。
```

```
requires \typeof(*q) == char && \typeof(*t) == char &&  
        \is_pstring(q, \strlen(q)) && \is_pstring(t, \strlen(t));  
ensures \typeof(*q) == char && \typeof(*t) == char &&  
        \is_pstring(q, \strlen(q)) && \is_pstring(t, \strlen(t)) &&  
        (\string(q, \strlen(q)) < \string(t, \strlen(t)) && \result < 0 ||  
        \string(q, \strlen(q)) == \string(t, \strlen(t)) && \result == 0 ||  
        \string(q, \strlen(q)) > \string(t, \strlen(t)) && \result > 0); /*/
```

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
// 整数大小比较函数，作为 main 函数中第一个 qsort 调用的最后一个实参。
/*@ requires \typeof(*a) == int && \typeof(*b) == int &&
    0 <= \offset(a) < \length(a) && 0 <= \offset(b) < \length(b);
    ensures \typeof(*a) == int && \typeof(*b) == int &&
    0 <= \offset(a) < \length(a) && 0 <= \offset(b) < \length(b) &&
    (\result < 0 && *a < *b || \result == 0 && *a == *b || \result > 0 && *a > *b); */
int comp_int(const void* const a, const void* const b){
    int *pa, *pb;
    pa = (int*)a; pb = (int*)b; // 从函数的前条件知道，这两个强制是正确的。
    if (*pa < *pb){ return -1; } if (*pa > *pb){ return 1; } return 0;
}
// 字符串的词典序比较函数，作为 main 函数中第二个 qsort 调用的最后一个实参。
/*@ requires \typeof(*p1) == char && \typeof(*p2) == char &&
    \is_pstring(p1, \strlen(p1)) && \is_pstring(p2, \strlen(p2));
    ensures \typeof(*p1) == char && \typeof(*p2) == char &&
    \is_pstring(p1, \strlen(p1)) && \is_pstring(p2, \strlen(p2)) &&
    (\string(p1, \strlen(p1)) < \string(p2, \strlen(p2)) && \result < 0 ||
    \string(p1, \strlen(p1)) == \string(p2, \strlen(p2)) && \result == 0 ||
    \string(p1, \strlen(p1)) > \string(p2, \strlen(p2)) && \result > 0); */
int comp_str(const void* const p1, const void* const p2){
    char *src, *des;
    src = (char*)p1; des = (char*)p2;
    return strcmp(src, des);
}
/*@ requires \true; ensures \result == 0; */
int main(void){
    int i;
    int a[10] = {3,5,1,8,6,7,2,9,4,0};
    char s[5][6] = {"1235", "54321", "23451", "54322", "12344"};
    // 一维整型数组排序
    qsort(a, 10, sizeof(int), comp_int); // 从第 4 个实参 comp_int 函数的前条件可以知道，comp_int
    函数的实参实际是 int* 类型。这样，第 3 个实参 sizeof(int) 符合要求，实参 a 的类型也符合要求。
    printf("\n after sorting:\n");
    /*@ loop invariant 0 <= i <= 10 && (\forall int j:[0..8]. a[j] <= a[j+1]) &&
    (\forall int j:[0..4]. \is_pstring(s[j], \strlen(s[j]))) &&
    \string(s[0], 4) == "1235" && \string(s[1], 5) == "54321" && \string(s[2], 5) == "23451" &&
    \string(s[3], 5) == "54322" && \string(s[4], 5) == "12344"; */
    for(i=0; i<10; i++){ printf("%d\t", a[i]); }
    // 字符串数组排序
    qsort(s, 5, sizeof(s[0]), comp_str); // 类似上面的情况。
    printf("\n after sorting:\n");
    /*@ loop invariant 0 <= i <= 5 && (\forall int j:[0..4]. \is_pstring(s[j], \strlen(s[j]))) &&
    (\forall int j:[0..3]. \string(s[j], \strlen(s[j])) <= \string(s[j+1], \strlen(s[j+1])); */
    for(i=0; i<5; i++){ printf("%s\t", s[i]); } return 0;
}

```

图 4.7 快速排序程序

```

/*@ //同样，这是 qsort 协议的主要部分。有关第 4 个形参应有的性质见上面的协议。
requires \typeof(**p) == char && \length(p) == len && size == 6 * sizeof(char) &&
    (\forall integer j:[0..len-1]. \is_pstring(p[j],\strlen(p[j])));
ensures (\forall integer j:[0..len-1]. \is_pstring(p[j],\strlen(p[j])) &&
    \forall integer j:[0..len-2].\string(p[j], \strlen(p[j])) < \string(p[j+1], \strlen(p[j+1]));
*/

```

从这两个协议可看到，`qsort` 还要求，第一个形参 `p` 所指向对象的长度等于第二个形参 `len`。第三个形参 `size` 等于第一个形参所指向对象中一个元素所需的空间大小。

在 C 语言中，`void*` 作为形参的类型，主要是便于程序员编写一些对多种类型通用的函数，但把其中的类型安全留给程序员自己负责了。`SCSL` 采用增加类型标注的方式，在程序验证过程中来保证类型安全。

目前 `SCSL` 和验证系统对类型为 `void*` 的形参，以及随之可能有的 `size_t` 类型的形参或返回值的支特见下面两点。其中 `size_t` 类型的形参或返回值用于表示 `void*` 指针指向数据区的大小或数据区中偏移。

- 若函数 `f` 有 `void*` 类型的形参 `x`，则函数 `f` 的协议必须指明 `x` 对应实参的所有可能类型。

若函数 `f` 的形参 `void* x` 所对应实参不是唯一类型的，例如前条件中有 `(\typeof(*x) == char || \typeof(*x) == short || \typeof(*x) == long)`。则一般来说，这时函数 `f` 的代码需要根据 `*x` 的类型来分情况处理。由于 `\typeof(*x)` 是不能用到代码中来指示分情况处理的，因此函数 `f` 还需要有另一个形参 `y` 来帮助实现这一点，并且在函数前条件中把 `y` 和 `\typeof(*x)` 取值的对应用断言表达清楚。

- 同样，若函数 `g` 的形参中有作为形参的函数 `f形`，则 `f形` 的协议必须指明其 `void*` 类型形参对应实参的所有可能类型。并且对函数 `g` 采用多协议方式来描述，每种协议对应到一种类型。言下之意验证系统不支持程序员编写类似 `qsort` 这样的函数，因为 `qsort` 对作为形参的 `compare` 函数没有类型限定，只要能给出该类型两个元素的比较函数就可以。

注意，例 3.10 的函数 `h` 虽然也有函数指针作为形参，但该形参函数没有 `void*` 类型的参数。因此它在函数协议的前条件中用 `p == &f || p == &g` 来表示形参可以用函数 `f` 或 `g` 来代换，不需要用 `\typeof(...) == ...` 断言。

另外，在不涉及 `void*` 类型的形参时，对于类型 `size_t`，`SCSL` 和验证系统的支特如下。

- `size_t` 只能作为函数形参和返回值的类型，并且满足下面两点。
- 类型为 `size_t` 的形参，其对应实参的值正好是该函数某个形参的类型 `t` 的 `sizeof(t)` (也可以是 `t` 的元素的 `sizeof(t[0])`)。或者正好等于该函数某个指针型形参的类型 `p` 的 `\length(p)`。这两点约束必须以断言方式出现在函数的前条件中。
- 对于作为返回值类型的 `size_t`，该函数一定有形参，其类型和返回值的 `size_t` 类型相关，只要函数后条件正确写出了这个关系则可。例如：

```

/*@ logic int n;
/*@ requires \is_pstring(str, n) && n <= INT_MAX;
    ensures \is_pstring(str, \result) && \result <= INT_MAX;
*/

size_t strlen(const char * const str);

```

这样的例子在标准库函数中比较多。程序员也可以编写 `size_t` 类型的类似使用的代码，但禁止使用 `size_t*` 类型。

`size_t` 虽然是依赖于实现的类型，但 `size_t` 在程序中的上述使用方式保证了按这些方式使用的 `size_t` 不依赖于实现。

第 5 章 语句标注

C 的语句标注 (*statement annotation*) 有两种:

1. 程序点断言: 允许出现在任何语句之前或者程序块的结尾。
2. 循环标注: 有 *invariant* 和 *variant* 两种, 它们出现在 *while*、*for* 和 *do* 语句之前。

图 5.1 是 C 语句文法的扩展, 其中非终结符 *compound-statement*、*declaration* 和 *statement* 是 C 的非终结符, 非终结符 *assertion* 和 *loop-annot* 等是为语句标注而扩展的。

<i>compound-statement</i>	→ { <i>declaration</i> * <i>statement</i> * <i>assertion-check</i> ? }
<i>statement</i>	→ <i>assertion-check</i> ? <i>statement</i>
<i>assertion-check</i>	→ <i>assertion</i> <i>check</i>
<i>assertion</i>	→ /*@ assert <i>assert</i> ; */
<i>check</i>	→ /*@ check <i>assert</i> (, <i>assert</i>) * ; */
<i>statement</i>	→ /*@ loop-annot */ while (<i>expr</i>) <i>statement</i> /*@ loop-annot */ for (<i>expr</i> ; <i>expr</i> ; <i>expr</i>) <i>statement</i> /*@ loop-annot */ do <i>statement</i> while (<i>expr</i>) ;
<i>loop-annot</i>	→ <i>loop-invariant</i> * <i>loop-variant</i> ?
<i>loop-invariant</i>	→ loop invariant <i>assert</i> ;
<i>loop-variant</i>	→ loop variant <i>term</i> ; loop variant <i>term</i> for <i>id</i> ; //针对关系 id 的变式

图 5.1 语句标注的文法

5.1 程序点断言

程序点断言由关键字 *assert* 和 *check* 分成两种不同的用法, 因为它们出现的位置相同(但不能同时出现在同一个位置), 因此在本节中分成两小节来介绍。

5.1.1 assert断言

assert 断言的文法见图 5.1。从文法可以看出, *assert* 断言可以出现在任何语句之前和程序块末尾语句之后。但是, 程序点断言不能出现在循环语句之前的那个程序点, 因为这里有循环不变式, 可能还有循环变式。

assert Q 的含义是, *Q* 在当前状态下必须成立。当前状态是指程序正好执行到断言所在程序点的状态。并且 *Q* 是作为当前程序点的断言, 继续向下验证。

允许在任意程序点标注 *assert* 断言, 会引起验证条件的增加。但是适当地在某些程序点标注断言, 会给程序验证带来帮助:

1. 有助于程序员发现代码或断言中的错误。
2. 对于 *if* 和 *switch* 这类形成多个分支的语句, 在这些语句之后的程序点标注 *assert* 断言, 把这些分支的共同性质体现出来, 在有些场合可以简化随后代码中的验证条件。
3. 往回跳转的 *goto* 语句被看成构成一个循环, 程序员必须用 *assert* 断言的方式给出跳转所到达语句的前断言, 作为该循环的循环不变式。

需要注意的是, *assert Q* 必须包含在当前状态成立并且对随后代码的验证来说必不可少的所有性质。*assert Q* 不是作为程序员查看程序点是否具有某个性质的一种手段。

assert 断言暂时只能用于不操作易变数据结构的代码。

5.1.2 check断言

check 断言可以出现在 assert 断言可以出现的地方。若当前程序点的断言是 P, check Q 的含义是, 若 $P \implies Q$ 得证, 则把 $P \ \&\& \ Q$ 作为当前程序点断言, 继续向下验证。若 $P \implies Q$ 未得证, 则仍把 P 作为当前程序点断言, 继续向下验证。check Q_1, Q_2 的含义是, 先证 $P \implies Q_1$, 再证 $P \ \&\& \ Q_1 \implies Q_2$ 。若两步都得证, 则把 $P \ \&\& \ Q_1 \ \&\& \ Q_2$ 作为当前程序点断言并继续向下验证; 若仅前一步得证, 则把 $P \ \&\& \ Q_1$ 作为当前程序点断言并继续向下验证; 若两步都未得证, 则仍把 P 作为当前程序点断言并继续向下验证。

check 断言的一个重要用途是, 若 $P \implies Q$ 和 $P \ \&\& \ Q \implies R$ 都能证明, 但因所用定理证明器的能力问题, $P \implies R$ 就是证明不了, 并且 R 不是一个需要归纳证明的性质。这时可用 check 断言方式进行帮助, 免去显式引入引理 $P \implies Q$ 。

同样, check 断言暂时也只能用于不操作易变数据结构的代码。

5.2 循环标注

循环标注的语法见图 5.1。下面分别介绍循环不变式和循环变式。

5.2.1 循环不变式

一个简单的循环不变式标注如下:

```
//@ loop invariant I;  
... // 循环语句的代码
```

它表示下面的 2 个条件成立。

1. 粗略地说, 在整个循环开始执行之前, 断言 I 成立。对于循环语句的三种形式, 这个粗略的说法有细微的区别, 分别仔细说明。

- (1) 对于 while (c) s 循环, 断言 I 在该语句之前程序点的状态中成立。
- (2) 对于 for (init ; c ; step) s 循环, 断言 I 在执行初始化表达式 init 之后的状态中成立。
- (3) 对于 do s while (c) 循环, 断言 I 在第一次执行 s 之后的状态成立。

2. 断言 I 是归纳不变式。它的含义是, 若 I 和 c 在某个状态中都为真, 并且在这个状态下执行循环体能正常终止 (或通过 continue 终止) 在循环体的结束点, 则 I 在结果状态中仍为真。在不允许循环条件 c 有副作用的情况下, 可以再精确描述为:

- 对于 while (c) s 循环, 断言 I 必须由 s 保持。
- 对于 for (init ; c ; step) s 循环, 断言 I 必须由 s 和随后的 step 保持。
- 对于 do s while (c) 循环, 断言 I 必须由 s 保持。

再进一步, 若在循环体中没有 return、goto 和 break 等可以非正常离开循环的语句, 这三种循环语句可以用 Hoare 逻辑推理规则表示如下。

$$\frac{\{I \wedge c\} s \{I\}}{\{I\} \text{while}(c) s \{I \wedge \neg c\}} \quad \frac{\{P\} \text{init} \{I\} \quad \{I \wedge c\} s ; \text{step} \{I\}}{\{P\} \text{for}(\text{init} ; c ; \text{step}) s \{I \wedge \neg c\}}$$
$$\frac{\{P\} s \{I\} \quad \{I \wedge c\} s \{I\}}{\{P\} \text{do} s \text{while}(c) \{I \wedge \neg c\}}$$

循环语句若无 invariant 子句, 则被默认为 invariant \true。它没有描述任何循环不变性。

例 5.1 从上述 do s while(c) 语句的推理规则看, 循环不变式是在第 1 次执行循环体 s 后成立, 而不是在 do s while(c) 语句的入口点就成立。若把 do s while(c) 语句的循环不变式标注

在循环体 s 之后，不会造成误会。但带来的问题是，是否把 `for (init; c; step) s` 语句的循环不变式标注调整在 `init` 之后。为了方便程序员使用，还是让循环不变式标注都出现在循环语句的入口。程序员一定要按上面的三条推理规则来理解循环不变式。 □

```
#define Max 10000
/*@ requires 0 < n <= Max && \length(t) == n;
    ensures \result == \max(0, n-1, (\lambda integer k. t[k])) && \length(t) == n;
*/
//\max 是高阶逻辑构造，见 7.5 节
int max(int* const t, const int n) {
    int i = 0; int m, v;
    m = t[0];
    /*@ loop invariant 0 < i <= n <= Max && m == \max(0, i-1, (\lambda integer k. t[k])) && \length(t) == n; */
    do {
        v = t[i++]; m = v > m ? v : m;
    } while (i < n);
    return m;
}
```

图 5.2 查找一维数组中的最大值

例 5.2 二叉堆可以用一维数组来实现 [4]，其特点是，对任何大于 1 的 j ， $\text{elements}[j] \geq \text{elements}[j/2]$ (elements 是存放数据的一维数组)。 element 是全局数组， $\text{elements}[0]$ 等于 INT_MIN ，任何插入的值都不小于 $\text{elements}[0]$ 。 size 是全局变量，指示 elements 数组中当前有多少个元素 ($\text{elements}[0]$ 除外)。图 5.3 包括了全局声明及标注，插入函数 `insert` 的代码及其协议，以及其中循环代码的循环不变式标注。图 5.4 是删除最小元素 $\text{elements}[1]$ 的 `delete` 函数的代码、协议和循环不变式，其中与图 5.3 一致的全局声明及标注没有给出。

```
#include <limits.h>
#define CAPACITY 10000
int size = 0;
/*@ global invariant capacity : 0 <= size <= CAPACITY; // 全局不变式，见第 7 章
int elements[CAPACITY+1];
/*@ requires size >= 0 && size <= CAPACITY-1 && elements[0] == INT_MIN &&
    (\forall integer j:[1..size].elements[j] >= elements[j/2]) && x > INT_MIN;
    assigns elements, size;
    ensures size > 0 && size <= CAPACITY && (\forall integer j:[1..size].elements[j] >= elements[j/2]); */
void insert(int x) {
    int i, t;
    size = size + 1; i = size; t = i / 2;
    /*@ loop invariant
        size > 0 && size <= CAPACITY && i == size && t == i/2 && elements[0] == INT_MIN &&
        x > INT_MIN && (\forall integer j:[1..size-1].elements[j] >= elements[j/2]) ||
        size > 0 && size <= CAPACITY && i <= size/2 && i > 0 && t == i/2 && elements[i] > x &&
        elements[0] == INT_MIN && x > INT_MIN &&
        (\forall integer j:[1..size].elements[j] >= elements[j/2]); */
    while (elements[t] > x) {
        elements[i] = elements[t]; i = i / 2; t = i / 2;
    }
    elements[i] = x;
}
```

图 5.3 二叉堆的插入函数

二叉堆的程序看似简单，但循环不变式不像例 5.1 那么直观。例 5.2 的循环不变式分成两种情况。从循环之前的代码到达循环入口时，程序状态满足第一种情况。循环体每次迭代计算后都满足第二种情况。这两种情况不能合并，因为刚到循环语句入口时， x 尚未与 $elements$ 的任何元素比较，不能保证这时一定满足第二种情况不可少的的断言 $elements[i] > x$ 。删除函数更复杂，循环不变式分成三种情况，缺一不可。 □

```

/*@ requires size > 0 && size <= CAPACITY && (\forallall integer j:[1..size].elements[j] >= elements[j/2]) &&
    \old(elements[1]) == elements[1];
assigns elements, size;
ensures size >=0 && size <= CAPACITY-1 && (\forallall integer j:[1..size].elements[j] >= elements[j/2]) &&
    \result == \old(elements[1]); */
int delete(){ /* 删除最小值 elements[1] */
    int i, child, lastElement, minElement;
    minElement = elements[1]; lastElement = elements[size]; size = size -1;
    if (size > 0) {
        i = 1; child = i * 2;
        if (child < size && elements[child +1] < elements[child]) {
            child = child + 1;
        }
        /*@ loop invariant
        size > 0 && size <= CAPACITY-1 && i > 0 && i <= size && child == i*2 && child < size &&
            elements[child] <= elements[child+1] && minElement == \old(elements[1]) &&
            (\forallall integer j:[1..size].elements[j] >= elements[j/2]) && lastElement >= elements[i] ||
        size > 0 && size <= CAPACITY-1 && i > 0 && i <= size && child == i*2+1 && child <= size &&
            elements[child-1] > elements[child] && minElement == \old(elements[1]) &&
            (\forallall integer j:[1..size].elements[j] >= elements[j/2]) && lastElement >= elements[i] ||
        size > 0 && size <= CAPACITY-1 && i > 0 && i <= size && child == i*2 && child >= size &&
            minElement == \old(elements[1]) && lastElement >= elements[i] &&
            (\forallall integer j:[1..size].elements[j] >= elements[j/2]);
        */
        while (child <= size && lastElement > elements[child]) {
            elements[i] = elements[child]; i = child; child = i * 2;
            if (child < size && elements[child +1] < elements[child]) {
                child = child + 1;
            }
        }
        elements[i] = lastElement;
    }
    return minElement;
}

```

图 5.4 二叉堆的删除最小元函数

下面给出一个循环不变式有误的简单例子。

例 5.3 一段 while 循环代码如下：

```

int x = 0; int y = 10;
/*@ loop invariant 0 <= x < 11; */
while ( y > 0 ) {
    x++; y--;
}

```

虽然在上述代码的执行过程中能始终保持 $0 \leq x < 11$ 。但是仅把 $0 \leq x < 11$ 作为循环不变式是证明不了 $x < 11$ 的。因为 x 的值和 y 的值是关联的, y 的值在循环过程中也有变化, 但循环不变式没有关于 y 的部分。正确的循环不变式应是 $0 \leq x < 11 \ \&\& \ x + y == 10$ 。 □

5.2.2 循环变式

下面介绍循环变式。循环变式用于验证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。之所以不能正式称为循环终止, 是因为终止还依赖于该循环内的调用语句都会返回, 内循环都会终止。循环变式是可选的, 若不使用则表示不要求对循环进行这种验证。若使用则其在循环标注中的形式是

```
//@ loop variant e;
```

其中 e 是 `integer` 或它的某个子类型的项。

它表示, 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。注意, e 的值在循环出口处可以是负数, 这并不危及循环的终止。

例 5.4 图 5.5 是循环变式的一个简单例子。 □

用其他的序而不是常见的整数上的序来描述终止性是可行的, 在第六章进一步解释。

在循环嵌套的情况下, 循环变式有一点使用的限制, 就是内层循环的代码不得修改外层循环的循环变式中的变量。

```
//@ requires -2<= x <= 10;
void f(int x) {
    /*@
        loop invariant -2<= x <= 10;
        loop variant x+1; */
    while (x >= 0) {
        x -= 2;
    }
}
```

图 5.5 一个简单的循环变式

第 6 章 终止性

程序的终止性也是一个重要问题，它与循环语句以及递归函数调用都有关系。终止性是通过把每个循环和每个递归函数各联系到一个度量 (*measure*) 函数来保证的，其中有关循环的终止性在 5.2 节已经简单讨论过。若无特别说明，度量函数的表达式默认为整型，并且前后两次度量之间用整数上常规的序进行比较。也可以把度量函数定义到其他论域和/或使用不同的序关系。

6.1 整数度量

函数的整数度量采用语法为

```
//@ decreases e;
```

的标注，它出现在函数协议中，其中度量项 *e* 中的变量必须是函数的形参。循环的整型度量用语法上类似的

```
//@ loop variant e;
```

来标注，在第 5 章已经介绍过。在这两种情况下，逻辑表达式 *e* 的类型都是 *integer*。*e* 各次计算（在函数体和循环体的开始点）的值必须递减，才能满足整数上常规的序关系

$$R(x, y) \iff x > y \ \&\& \ x \geq 0$$

对前后两次值 *x* 和 *y* 的要求。换句话说，度量必须是一个递减的整数序列并且保持非负，只有序列的最后一个值可以例外。

例 6.1 考虑在安全 C 语言使用手册上讨论过的二分查找函数，见图 6.1。循环变式可以是 *j - i*。*j - i* 的值在每次迭代中降低并保持非负，只有最后一次迭代的值是负数。 □

```
/*@ lemma property1: \forall integer *b. \forall integer value. \forall integer 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 integer *b. \forall integer value. \forall integer 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 6.1 二分查找函数的代码及标注

6.2 一般度量

除了整数度量外，更一般的度量可以使用关键字 `for` 来建立在其他类型上。对于函数，语法形式为

```
//@ decreases e for R;
```

对于循环语句，语法形式为

```
//@ loop variant e for R;
```

在这两种情况下，表达式 `e` 属于某种类型 τ 并且 `R` 必须是 τ 上的二元关系，即是一个已经声明的二元谓词：

```
//@ predicate R( $\tau$  x,  $\tau$  y) ...
```

当然，为了保证终止，必须证明 `R` 是良基关系。目前仅允许 τ 是整型、元素都是整型的构造类型，或者含自引用的结构体类型，`loop variant e for R` 中的 `e` 只能是 τ 类型的变量。

例如，下面的谓词定义字典序。

```
predicate lexico(int p1[2], int p2[2]) =  
    p2[0] < p1[0] && 0 <= p2[0] && 0 <= p2[1] ||  
    p2[0] == p1[0] && p2[1] < p1[1] && 0 <= p2[0] && 0 <= p2[1];
```

例 6.2 图 6.2 是循环变式的一个例子，它使用一对整数，按上面所定义的字典序。由于 SCSL 没有元组形式的项，因此需要通过幽灵语句把这一对整数赋给长度为 2 的幽灵数组。该例涉及第 7 章的谓词定义概念和第 9 章的幽灵 (*ghost*) 变量和幽灵语句概念。

另外还要注意，本例有两个函数，这两个函数都有协议。 □

```
#include <limits.h>  
/*@ predicate lexico(int p1[2], int p2[2]) = p2[0] < p1[0] && 0 <= p2[0] && 0 <= p2[1] ||  
    p2[0] == p1[0] && 0 <= p2[0] && p2[1] < p1[1] && 0 <= p2[1];  
*/  
/*@ ensures INT_MAX >= \result >= 0;  
int dummy (); // 函数值大于或等于 0  
/*@ requires INT_MAX >= x >= 0 && INT_MAX >= y >= 0;  
void f(int x, int y) {  
    //@ ghost int p[2];  
    //@ ghost p[0] = x;  
    //@ ghost p[1] = y;  
    /*@ loop invariant INT_MAX >= x >= 0 && INT_MAX >= y >= 0 && p[0] == x && p[1] == y;  
    loop variant p for lexico;  
    */  
    while (x > 0 && y > 0) {  
        if (dummy ()) {  
            x --; y = dummy();  
            //@ ghost p[0] = x;  
            //@ ghost p[1] = y;  
        } else {  
            y --;  
            //@ ghost p[1] = y;  
        }  
    }  
}
```

图 6.2 使用字典序判断循环的终止

6.3 递归函数调用

函数调用图上强连通的相互递归函数称为一个函数簇。为便于以文件为单位的分模块验证，一个函数簇中的函数必须定义在同一个 C 文件中。在一个函数簇中，各个函数必须标注基于同一个关系 R （从语法上看是同一个关系）的 *decreases* 子句。在该簇中的任意函数 f 的函数体中，任何对递归函数 g 的调用所在状态对 g 的度量（即 g 的度量项在该调用点的值），小于 f 函数开始状态中对 f 的度量（即 f 的度量项在 f 开始点中的值）。这个原则也适用调用自身的直接递归场合。

例 6.3 图 6.3 是经典的阶乘函数和斐波那契函数。

在斐波那契函数的两个递归调用点，验证函数入口时的度量与调用点的度量之间关系的验证条件如下（其中 $\text{old}(n)$ 表示 n 在函数入口点的值）：

$$n \geq 0 \ \&\& \ \text{old}(n) == n \ \&\& \ n \neq 0 \ \&\& \ n \neq 1 \\ \implies \text{old}(n) > n-1 \ \&\& \ n-1 \geq 0, \ \text{和}$$
$$n \geq 0 \ \&\& \ \text{old}(n) == n \ \&\& \ n \neq 0 \ \&\& \ n \neq 1 \\ \implies \text{old}(n) > n-2 \wedge n-2 \geq 0$$

如果该函数体缺少 $n == 0$ 或者 $n == 1$ 的分支，则这两个验证条件至少有一个得不到证明。 □

例 6.4 图 6.4 是相互递归函数的例子。 □

需要注意的是，无论是循环语句还是递归函数，若终止性的证明失败，并不代表它们一定不终止，有可能是度量表达式的不恰当。

类似于循环语句，一簇递归函数经过对度量的验证后，表示它们不会陷入无限次的递归调用（粗略地称为终止）。单独说某个循环不会陷入无限次迭代执行，某个函数簇不会陷入无限次递归的调用都不是严格意义上的终止。严格意义下的程序终止，是指构成一个程序的所有源文件的所有循环和所有递归函数簇，都不会陷入无限次迭代和无限次递归调用，则该程序是严格意义下的终止。这个概念也可用于一个函数。

```
/*@logic int factorial(int n) = n > 0 ?
    factorial(n - 1) * n : 1; */
/*@ requires 0 <= n < 10; decreases n;
    ensures \result == factorial(n); */
int fact(const int n) {
    if (n == 0) return 1;
    return n * fact(n-1);
}
/*@logic int fibonacci (int n) = n == 0 ? 0 :
    (n == 1 ? 1 :
        fibonacci(n - 1) + fibonacci(n - 2));*/
/*@ requires 20 >= n >= 0; decreases n;
    ensures \result == fibonacci(n); */
int fib(const int n) {
    if (n == 0) { return 0;
    } else if (n == 1) { return 1;
    } else { return fib(n-1) + fib(n-2);
    }
}
```

图 6.3 两个简单的直接递归函数

```
int odd(int x);
/*@ requires n >= 0; decreases n; */
int even(int n) {
    if (n == 0) return 1;
    return odd(n-1);
}
/*@ requires x >= 0; decreases x; */
int odd(int x) {
    if (x == 0) return 0;
    return even(x-1);
}
```

图 6.4 相互递归的函数

6.4 不终止的函数

在某些场合，函数就是要设计成不终止的。例如，反应式程序（*reactive program*）的 *main* 函数可能是 `while(1)`，它始终等待着下一个要处理的事件。更一般的情况是，函数可以在某些前条件得到满足时终止。在这种情况下，*terminates* 子句需要加到函数的协议中，其形式如下：

```
//@ terminates p;
```

以帮助验证函数在其所属函数的某些前条件得到满足时是否达到严格意义下的终止。在这里断言 p 所描述的是函数入口处的状态，若 p 中有变量在函数内被修改，则须用 old 定义。

terminates p 子句的语义是，如果 p 为真，那么所属函数被保证终止（更准确地说系统

必须证明它终止)。如果不存在这样的子句(尤其是若所属函数根本没有函数协议),它默认为 `terminates \true`, 也就是所属函数假设为一定终止, 这是大多数函数被期望的行为。

注意, 若 `p` 不为真, 相当于什么也没有说, 尤其是 `terminates \false` 并不暗示所在函数陷入无限循环。例如下面的函数

```
/*@ ensures \false;
   terminates \false;
*/
void f() { while(1); }
```

就是不会终止的一种可能情况。

`terminates p` 子句表达对函数终止的期望。在函数中有循环语句或函数本身是递归函数的情况下, 要让这种期望得到验证。这时, 函数的各循环语句的标注中必须给出 `variant` 子句, 若是递归函数则协议中必须给出 `decreases` 子句, 否则难以证明, 函数协议中的 `terminates` 子句也就没有什么意义。另一方面, 在有 `decreases` 子句和 `variant` 子句的情况下, 若 `terminates` 子句缺省, 则代表的是 `terminates \true`, 这是对函数总是终止的期望。

在函数各循环都有 `variant` 子句, 递归函数协议中有 `decreases` 子句的情况下, `terminates` 子句的语义是, 如果 `p` 在函数开始状态成立, 则函数终止。若 `p` 在函数开始状态不成立, 则等于什么也没有说, 即不知道该函数的执行是否终止。

例 6.5 图 6.5 是一个并非一定终止的函数的例子。但它在 $m \leq 0$ 时一定终止。本例略去了数据可能溢出的证明。

□

```
/*@ requires m == \old(m) && m <= 0;
   *@ terminates \old(m) <= 0;
void inc(int m) {
   /*@ loop invariant m <= 0 && m >= \old(m);
   /*@ loop variant -m;
   while(m != 0) {
       m = m + 1;
   }
}
```

图 6.5 并非一定终止的函数

注意, `terminates \false` 并不意味着相应函数陷入无穷循环。

第 7 章 全局逻辑定义和声明

通过声明新的逻辑常量、逻辑变量、逻辑函数和谓词，可以扩展使用在标注中的逻辑表达式语言。这些声明的文法见图 7.1。

<i>C-global-decl</i>	→	<i>/*@ logic-def* */</i>	
<i>logic-def</i>	→	<i>logic-const-def</i> <i>logic-variable-decl</i> <i>logic-function-def</i> <i>predicate-def</i> <i>lemma-decl</i> <i>inductive-def</i> <i>macro-def</i> <i>import-clause</i> <i>using-clause</i>	
<i>logic-const-def</i>	→	<i>logic type-expr id = term ;</i>	//逻辑常量定义
<i>logic-variable-decl</i>	→	<i>logic type-expr (*)[?] id (, (*)[?] id)* ;</i>	//逻辑变量声明
<i>logic-function-def</i>	→	<i>logic type-expr id parameters (= term)[?] ;</i>	//逻辑函数定义
<i>predicate-def</i>	→	<i>predicate id parameters = assert ;</i>	//谓词定义
<i>inductive-def</i>	→	<i>inductive id parameters (= assert)[?] ;</i>	//归纳谓词定义
<i>lemma-decl</i>	→	<i>lemma id ((both))[?] : assert ;</i> <i>lemma id ((both))[?] relies on : id (, id)*) : assert ;</i>	//引理声明
<i>parameters</i>	→	<i>(parameter (, parameter)*)</i>	
<i>parameter</i>	→	<i>type-expr (*)[?] id</i> <i>type-expr id [] ([integer])[*]</i> <i>type-expr id [integer] ([integer])[*]</i>	
<i>macro-def</i>	→	<i>define id : assert ;</i> <i>define id mac-params : assert ;</i>	//无参数的宏定义 //带参数的宏定义
<i>mac-params</i>	→	<i>(mac-param (, mac-param)*)</i>	
<i>mac-param</i>	→	<i>id</i>	//在词法上同于 C 标识符
<i>import-clause</i>	→	<i>import id (<id (id : id (, id : id)*>)?);</i>	//各 id 的含义见 7.7 节
<i>using-clause</i>	→	<i>using id (<id >)? (, id (<id >)?)* ;</i>	//各 id 的含义见 7.7 节

图 7.1 全局逻辑定义和声明的文法

7.1 逻辑定义和声明的作用域

逻辑定义和声明若出现在 C 函数的外面，则它们的作用域是从定义或声明点到所在文件的结尾。对于需要被多个文件共享的部分，可以定义或声明在被它们包含的头文件中。与其它定义和声明不同的是，引理声明若出现在 C 函数的内部，则引理声明的作用域只从声明点到所在函数的结尾。对逻辑变量声明的一点额外限制见 7.2 节。

逻辑定义和声明出现的名字（有点像关键字的 *logic*, *predicate*, *inductive*, *lemma*, *relies* 和 *on* 除外），按作用域原则，不能与 C 代码部分的名字有冲突。

此外还需注意先定义后引用原则：

- 逻辑部分和 C 代码部分分别都遵循这个原则。C 代码定义的名字可以在标注中引用，但必须先出现 C 的定义，然后才可以在逻辑定义和声明中引用 C 定义的名字。

- 对于逻辑定义和声明中出现的归纳谓词和归纳函数来说，由于可能出现间接归纳定义，因此允许先声明谓词和函数的原型，就像 C 的函数声明那样。

7.2 逻辑常量和逻辑变量

就像 C 的常量定义那样，逻辑常量定义也比较简单，被定义的常量在之后的标注中可以引用。对于逻辑常量声明 $\text{logic type-expr id} = \text{term}$ 中 term 的语法约束是：只能出现直接量和先前已经声明的逻辑常量，以及可以操作于它们的算符。

SCSL 中的逻辑变量可以用来表达程序对上下文的不变性质，这些性质用程序变量不足以表达。在 Hoare 逻辑中，程序断言表达程序变量在某个程序状态下的性质，难以描述程序变量在不同状态下的值之间的联系。若要用断言表达这种联系，使用逻辑变量是一种简单的解决办法。举个简单的例子，若不考虑机器能表示的最大整数有限制，若要表达无论 m 当前的值是多少，经过赋值 $m = m + 2$ 后， m 的值总是增 2。那么，若使用逻辑变量 oldm 表示 m 的当前值，则可写出 Hoare 三元式 $\{\text{oldm} == m\} m = m + 2 \{\text{oldm} == m - 2\}$ 。 oldm 的作用就是代表 m 原先的值。

在 SCSL 中，程序员声明的全局逻辑变量主要用来表达函数调用前后两个程序点的性质变化，因此一个逻辑变量同时出现在函数的前后条件中时才有实际意义。一个逻辑变量可以出现在多个函数的前后条件中，用于各自的目的。

逻辑变量的使用限制如下：

- 逻辑变量仅可以出现在断言中。
- 一个逻辑变量只有出现在函数的前条件中，它才可以出现在函数体中的循环不变式、程序点断言和函数的后条件中。这意味着不允许在函数体中声明局部的逻辑变量。
- 在函数前条件中，任意一个逻辑变量，只有其本身单独作为谓词应用的变元、逻辑函数应用的变元、量化断言区间的上下界或关系运算的一个运算对象（在此 $0 \leq x < 100$ 理解成一个或两个关系运算式都可以）后，才能出现在前条件的其他地方。可以理解为前者是逻辑变量的定义性出现（若有 $t \rightarrow \text{level} == m$ 和 $t \rightarrow l \rightarrow \text{level} == m$ ，其中 m 是逻辑变量，则择其一为定义性出现），其他地方的是引用性出现。一个谓词应用、逻辑函数应用、量化断言区间和关系运算最多只能有一个逻辑变量的定义性出现。
- 在描述逻辑变量定义性出现的关系运算中，关系运算的其他运算对象可以包括程序变量甚至量化断言。若是后者，则把整个量化断言理解成一个虚拟程序变量。在前条件中，这些程序变量尽量只用于逻辑变量的定义性出现的基本断言中，少出现在前条件的其他地方，以方便逻辑变量的匹配。

逻辑变量的一种典型应用是表达函数调用的指针型实参在函数返回点具有的性质。由于 C 语言允许函数体对形参赋值，因此，在一般情况下，SCSL 禁止不带 `const` 修饰符的形参出现在函数后条件中。这是因为不带 `const` 修饰符的形参可能被重新赋值，形参在函数返回点具有的性质不能作为实参在调用之后的性质。这时，逻辑变量可用来表达实参在函数调用之后具有的性质，或者表达其在调用前后的性质变化。

- 在函数前条件中，若指针型逻辑变量出现在等式断言中，则等式的另一端只能是程序指针变量，不能是程序指针加偏移的表达式。

例 7.1 图 7.2 是复制有序单向链表的函数框架。在有序单向链表的归纳谓词定义 `sortedList` 中，第 1 种情况是 $p == \text{\null}$ ，第 2 种情况是 $p != \text{\null} \ \&\& \ p \rightarrow \text{next} == \text{\null}$ ，第 3 种情况是 $p != \text{\null} \ \&\& \ p \rightarrow \text{next} != \text{\null} \ \&\& \ \dots$ 。在安全 C 语言使用手册中已经介绍过，在程序员给出的断言中，在一个合取子句中出现一条访问路径 s 时，与之合取的 s 的各个前缀不等于 `NULL` 的断言可以略去不写。图 7.2 的 `sortedList` 归纳定义是用这种缺省形式描述的。

在函数定义中，形参 p 指向被复制的链表，函数返回值指向新生成的链表。调用结束时， p 仍然指向原来的有序链表。在函数协议中，断言 $\text{oldp} == p$ 用来表示每次调用时，逻辑变量 oldp 将始终等于实参。在本例中就是实参仍然指向原来的有序链表。若某次调用的实参

是 t ，它具有性质 $\text{sortedList}(t)$ 。根据函数调用的语义，函数前条件中的形参 p 代换成实参 t 。从程序验证的角度，因为函数前断言中有断言 $\text{oldp} == p$ ，函数前后条件中的逻辑变量 oldp 也应代换成 t 。这样，调用后就有断言 $\text{sortedList}(\backslash\text{result}) \ \&\& \ \text{sortedList}(t)$ ，也就是实参 t 在调用之后仍然指向原来的有序链表。

```

/*@ inductive sortedList(Node* p) = p == \null || p->next == \null ||
                                     p->data <= p->next->data && sortedList(p->next);

    ...

    logic Node* oldp;
*/
/*@ requires sortedList(p) && oldp == p;
    ensures sortedList (\result) && sortedList(oldp);
*/
Node* listCopy(Node* p){
    Node* q;
    ...;
    return q;
}

```

图 7.2 解释逻辑变量的使用

就本例而言，若 p 在函数体中未被赋值，则可写成带 const 修饰符的形参 p 。这时函数协议可写成下面的形式，即形参 p 可以出现在 ensures 子句中，从而不需要用逻辑变量 oldp 来表示实参。

```

/*@ requires sortedList(p);
    ensures sortedList (\result) && sortedList(p);
*/
Node* listCopy(Node* const p){ ... }

```

安全 C 语言使用手册的例 2.10 和图 2.5 是有序单向链表的插入函数。代表实参的逻辑变量 oldhead 和返回值（插入新节点后的链表的头节点指针）之间有三种可能的关系：

```

oldhead == \result, 或 oldhead == \result->next, 或 oldhead == \null

```

分别代表插入节点不作为首节点、作为首节点和原来的链表是空表三种不同情况。若不用逻辑变量 oldhead ，则很难在后断言中把三种情况区分清楚。 □

类似地，逻辑变量也可用来表达全局变量在函数调用前后的性质变化。

逻辑变量的另一种典型应用是表达函数对其调用环境的不变性质，这种性质不是用逻辑变量与形参或可能被赋值的全局变量之间的关系断言可表达的。例如，对于二叉排序树的插入函数 $\text{Node* BSTInsert(Node* t, int data)}$ 来说，在调用点程序状态中的任何变量 x ，若 x 大于二叉排序树 t 上所有节点的数据域 d ，并且 x 大于待插入的数据 data ，则 x 大于结果二叉排序树上的所有数据。

例 7.2 图 7.3 是二叉排序树的插入函数及其标注，节点数据类型的声明及标注被略去。若该函数的前后条件都仅是 $\text{BST}(p)$ 是不够的。因为若如此，则递归调用 $\text{BSTInsert}(p->l, \text{data})$ 之后仅得到 $\text{BST}(p->l)$ 而得不到 $\text{Gt}(p->\text{data}, p->l)$ ，从而不足以推导出 $\text{BST}(p)$ 。引入整型逻辑变量 succ 和 pred ，把函数前后条件分别增强，则可以解决这个问题。该函数第一个协议的前后条件分别是：

```

BST(p) && succ > data && gt(succ, p) && pred < data && lt(pred, p) && oldp == p

```

和

```

\result != \null && BST(\result) && gt(succ, \result) && lt(pred, \result) &&
(oldp == \null || oldp == \result)

```

前后条件中 $\text{succ} > \text{data} \ \&\& \ \text{Gt}(\text{succ}, p)$ 和 $\text{Gt}(\text{succ}, \text{result})$ 表示的意思是：对于任何值，只要在调用前它大于被插入数据 data 并且大于 p 指向二叉树上的所有数据，则它也大于插入 data 后的结果二叉树上的所有数据。显然，在递归调用 $\text{insert}(p->l, \text{data})$ 前有 $p->\text{data} > \text{data} \ \&\& \ \text{Gt}(p->\text{data}, p->l)$ ，因此调用后有 $\text{Gt}(p->\text{data}, p->l)$ 。

逻辑变量 pred 的用意类似。

与 insert 函数内部递归调用 $\text{insert}(p->l, \text{data})$ 和 $\text{insert}(p->r, \text{data})$ 时的情况不同，从其他函数调用 $\text{insert}(p, \text{data})$ 时，当前程序点可能不存在某个数据表达式 DATA 满足 $\text{DATA} > \text{data} \ \&\& \ \text{gt}(\text{DATA}, p)$ ，这时可以使用该函数的第二个协议，其前后条件分别是：

$\text{BST}(p) \ \&\& \ \text{oldp} == p$

和

$\text{BST}(\text{result}) \ \&\& \ (\text{oldp} == \text{null} \ \|\ \text{oldp} == \text{result})$

其实，此时仍用第一个协议也是可以的。在此调用场景下，协议前条件中的逻辑变量 succ 和 pred 若没有任何的匹配（理解为只能分别匹配到 succ 和 pred 本身）。没有代换成程序数据表达式的匹配对继续向下的验证是没有用处的，系统会把从调用所得结果中略去。这时用第一个协议的效果和用第二个协议的一致。 □

图 7.2 和 7.3 的逻辑变量声明 logic Node* oldp 中的 oldp 是堆指针，对于不是指向堆区的逻辑指针，其使用方式和目的是一样的。图 7.3 采取仅用第一个协议的方法。

```
#include <stdlib.h>
typedef struct node { int data;  struct node* l;  struct node* r;
}Node;
/*@ shape l, r : tree;
  inductive Gt(int succ, Node* p) =  p == \null || succ > p->data && Gt(succ, p->l) && Gt(succ, p->r);
  inductive Lt(int pred, Node* p) =  p == \null || pred < p->data && Lt(pred, p->l) && Lt(pred, p->r);
  inductive BST(Node* p) = p == \null ||
      BST(p->l) && BST(p->r) && Gt(p->data, p->l) && Lt(p->data, p->r);
  lemma p1:\forall int pred1, int pred2, Node* p. pred1 < pred2 && Lt(pred2, p) ==> Lt(pred1, p);
  lemma p2:\forall int succ1, int succ2, Node* p. succ1 > succ2 && Gt(succ2, p) ==> Gt(succ1, p);
  */
/*@ logic int succ, pred;
  logic Node* oldp;
  requires BST(p) && succ > data && Gt(succ, p) && pred < data && Lt(pred, p) && oldp == p;
  assigns *oldp;
  exits \exit_status == 1;
  ensures BST(result) && Gt(succ, result) && Lt(pred, result) && (oldp == \null || oldp == result); */
Node* BSTInsert(Node* p, const int data) {
  if ( p == NULL) {
    p = (Node *)malloc(sizeof(Node));
    if(p == NULL) exit(1);
    p->l = NULL; p->r = NULL; p->data = data;
  } else if (p->data > data) {
    p->l = BSTInsert(p->l, data);
  } else if (p->data < data) {
    p->r = BSTInsert(p->r, data);
  }
  return p;
}
}
```

图 7.3 二叉排序树的插入函数

逻辑数组是用来描述被函数修改的全局数组在函数执行前后的变化。逻辑数组的声明有两种方式：指定大小和未指定大小的数组（例如 `logic int olda[100], oldb[]`），后者可与元素类型相同的任意大小的数组匹配。逻辑数组的名字不被看成逻辑数组首元素的指针。区间相等断言可用来记住一个相容类型的全局数组或实参数组在函数入口的值。一个逻辑数组可以有若干个区间相等断言，在这些断言中，逻辑数组的区间不能出现重叠。

具体例子可见例 9.5。在该例中，函数前条件中有断言 `olda[0..n-1] == a[0..n-1]`（其中 `n` 是数组的大小），这是用逻辑数组 `olda` 来记住全局程序数组 `a` 在函数入口的值。若函数体中有语句修改 `a[50]` 的值，则随后两个数组之间的关系就是 `\forall i in [0..49, 51..n-1]. olda[i] == a[i]`，也可以写成 `olda[0..49] == a[0..49] && olda[51..n-1] == a[51..n-1]`。

现在对逻辑变量做一个小结。逻辑变量按其使用特点和受到的限制，可分成代换逻辑变量和匹配逻辑变量两类。

(1) 代换逻辑变量用于记住指针型实参和被调用函数可能赋值的全局变量在函数入口点的值，以便在函数出口点描述函数的执行给相应实参和全局变量带来的变化。这些逻辑变量在函数入口点的断言限定为等于某个形参或全局变量的等式断言。称为代换逻辑变量的原因是，通过直接对被调用函数前条件中含代换逻辑变量的断言的代换，得到调用语句之后程序点的对应断言。

(2) 匹配逻辑变量用于表达函数对其调用环境的不变性质。称为匹配逻辑变量的原因是，需在调用点的断言中寻找合适的匹配，然后再进行代换。见例 7.2。对含这类逻辑变量的断言的限制是：

- 若函数前条件是析取范式 $P_1 \parallel \dots \parallel P_m$ 的形式，则在这些子式 P_i ($1 \leq i \leq m$) 中，含这类逻辑变量的基本断言集必须相同。

- 若函数协议由多个命名行为组成，则每个命名行为要符合上述限制。

例 7.3 图 7.4 是有序单向链表的插入函数，用以展示 `\old` 构造难以用指针型逻辑变量取代。函数前后断言不仅描述结果链表保持有序性，还进一步表示结果链表的节点及数据与原链表的相比，仅多了一个节点及数据，其余节点的数据仍能按序与原链表的一一对应。这是本例与安全 C 语言使用手册中例 2.10 的区别，本例中有关形状 (*shape*) 的描述见第 8 章，有关链表长度函数 `\length` 也在那儿解释。

注意，其中访问路径 `head->(next:k)->data` 表示 `head->next->next ...->next->data`，其中“`->next`”重复出现 `k` 次，在安全 C 语言用户手册中已介绍过。这样的访问路径称为带折迭域的访问路径。

本例不可能通过引入指针型逻辑变量 `oldhead`，并在函数前条件中增加断言 `oldhead == head` 来免去使用 `\old` 构造。首先，由于形参 `head` 的值在函数中有可能被修改，那么在函数出口点，以指针型逻辑变量 `oldhead` 开始的访问路径是在出口而不是入口状态下取值，因而取到的值可能与入口状态下的不一样。其次，由于链表上节点个数不确定，也不可能通过增加若干个整型逻辑变量，用它们记住函数入口状态下链表各节点的数据就能达到目的。

另一方面，`requires` 子句的断言

```
\forall integer i:[0..m-1]. \old(head->(next:i)->data) == head->(next:i)->data
```

在函数入口点一定为真，因为相等关系右边的访问路径也是在函数入口状态下计算，该断言并不多余。它是要求从一开始就跟踪变化中的链表各节点的数据与初始链表各节点数据之间的联系。 □

逻辑变量的声明可以出现在头文件中。因为若调用语句和被调用函数分处于不同的源文件，则函数协议中涉及的逻辑变量必须声明在两个源文件都包含的头文件中。两个源文件对这些逻辑变量的使用可以有各自的安排。

```

#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 &&
  (\forallall int i:[1..m-1].head->(next:i-1)->data <= head->(next:i)->data) &&
  \forallall int i:[0..m-1].\old(head->(next:i)->data) == head->(next:i)->data;
  assigns oldhead;
  exits \exit_status == 1;
  ensures \list(result) && \length(result, next) == m+1 && oldhead == \result && 0 < m <= 1000 &&
    (\forallall int i:[1..m].\result->(next:i-1)->data <= \result->(next:i)->data) &&
    (\exists int j:[0..m]. \result->(next:j)->data == data &&
      (\forallall int i:[0..j-1]. \old(head->(next:i)->data) == \result->(next:i)->data) &&
      (\forallall int i:[j..m-1]. \old(head->(next:i)->data) == \result->(next:i+1)->data)) ||
  \list(\result) && \length(\result, next) == m+1 && oldhead == \null && m == 0 &&
    (\forallall int i:[1..m].\result->(next:i-1)->data <= \result->(next:i)->data) ||
  \list(\result) && \length(\result, next) == m+1 && oldhead == \result->next && 0 < m <= 1000 &&
    (\forallall int i:[1..m].\result->(next:i-1)->data <= \result->(next:i)->data) && \result->data == data &&
    (\forallall int i:[1..m-1]. \old(head->(next:i)->data) == \result->(next:i+1)->data);  */
Node* listInsert(Node* head, const int data) {
  Node *ptr, *ptr1, *p; 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; }
  else { ptr1 = head; ptr = head->next; j = 1;
    /*@ loop invariant 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 &&
      ( 1 <= j < m && ptr==ptr1->next && \list(ptr) && ptr != \null && \length(ptr, next) == m-j &&
        (\forallall int i:[1..j-1].head->(next:i-1)->data <= head->(next:i)->data) &&
        ptr1->data <= ptr->data &&
        (\forallall int i:[j+1..m-1].head->(next:i-1)->data <= head->(next:i)->data) &&
        (\forallall int i:[0..j-1].\old(head->(next:i)->data) == head->(next:i)->data) &&
        (\forallall int i:[j..m-1].\old(head->(next:i)->data) == head->(next:i)->data) ||
        j == m && ptr1->next == \null && ptr == \null &&
        (\forallall int i:[1..m-1].head->(next:i-1)->data <= head->(next:i)->data) &&
        (\forallall int i:[0..m-1].\old(head->(next:i)->data) == head->(next:i)->data)); */
    while ((ptr != NULL) && (ptr->data < p->data)) {
      ptr1 = ptr; ptr = ptr->next;
      j = j + 1;
    }
    p->next = ptr1->next; ptr1->next = p;
  }
  return head;
}

```

图 7.4 有序单向链表的插入函数

下面总结\old 构造和逻辑变量的联系和区别。

- \old 构造在函数前条件中通过\old(t) == t 的方式，用来记住函数的形参所对应的实参在函数入口点的值，也可用来记住会被函数修改的全局变量在函数入口点的值。在函数的前条件中，也可用逻辑变量等于全局变量或等于形参的断言来达到这个效果。

- 逻辑变量还具有\old 构造取代不了的其他功能。这些逻辑变量出现在函数前条件的断言中，当它们用调用状态中适当的表达式代换时，能够使得调用状态满足函数前条件。见例 7.2。

- \old 也可用于逻辑变量难以表达的地方，例如，例 7.3 用\old 构造来记住链表各节点的数据在函数入口点的值。

7.3 谓词定义和函数定义

逻辑函数和谓词分别由表达式和断言显式地定义，出现在等号之后。表达式和断言中出现的变量都必须是相应定义参数表中的变元。若出现间接递归的函数和间接归纳的谓词，则按照先定义后引用的原则，需适当地先给出其中某些函数和谓词的原型。图 7.1 中逻辑函数和谓词的文法给出了描述原型的可能。

在 3.4 节已经提到，逻辑函数和谓词的变元宜根据需求声明为 C 的合适类型，一般情况下不要使用逻辑类型 integer 和 real。

当数组和指针类型作为变元类型时，一些补充规定如下。

1. 数组类型可作为逻辑函数、谓词（包括归纳谓词，下同）和引理的变元类型，并且允许完整的数组描述和缺少数组大小的数组描述两种形式。

例如，若有 typedef int array2[10][20], int array1[20], 则

- 在逻辑定义 logic int f(array2 a)... 和 logic int f(int a[10][20])... 中，变元 a 的类型都有完整的描述，两个定义中 a 的类型都是 array2。这与 C 函数对形参的规定有区别，在 C 中，形参 a 的这两处声明都表示 a 是指针类型，是长度为 20 的一维整型数组的指针。这两个逻辑定义的变元是真正的数组，内建函数\offset(a)恒为 0，\length(a)等于数组的长度。应用这个逻辑定义的变元必须是同类型的数组。

- 在逻辑定义 predicate p(int a[][20], int len) = \length(a) == len && len > 0 && ... 中，变元 a 的类型是缺少数组大小的二维数组类型，\offset(a)属性仍然恒为 0，这时必须还有另一个作为数组长度的变元，并且在定义体中有断言来描述数组长度受到的约束，例如\length(a) == len && len > 0。允许用这种方式定义数组变元，是因为不能定义数组变元 int a[len][20]，其中 len 是变量。这种方式可用于允许数组 a 的长度任意的场合。应用这个逻辑定义的第一个实变元必须是数组，其元素的类型与该逻辑定义中第一个变元的元素类型相同。

允许数组类型作为逻辑函数等的变元类型，是为了让逻辑函数等的定义或声明不出现指针，容易理解。

- 若需要写通用一些的逻辑定义，允许让逻辑定义中数组变元的元素类型为 integer，而该逻辑定义的应用中，相应的数组元素可以是 ingeger 类型的任意子类型。

2. 指针类型也可以作为逻辑函数、谓词和引理的变元类型，但受到较多的限制

逻辑定义中实际上虽无指针类型的变元，但 C 语言有指针类型。为方便程序员为 C 代码写标注，允许逻辑定义在下面两种情况下使用指针类型的变元。从逻辑定义的观点，这样的指针变元实际代表它指向的对象。

- (1) 构造易变数据结构的自引用结构体或共用体类型 S 的指针。从逻辑定义的角度，S 类型的变量可以理解为，有一个域的类型就是 S 类型，而不是 S 类型的指针。

- (2) 在其他情况下，也允许使用 type* a 描述的指针变元，但这时的 a 实际上被看成数组，哪怕只有一个数组元素，还是因为逻辑定义中没有指针概念。这种方式与上述缺少长度

信息的数组类型基本一样，唯一的区别是，`\offset(a)`不一定是 0，因此需用第三个变元表示。第 3 个变元的类型与该逻辑定义中第一个变元指向对象的类型相同。

例如，逻辑定义 `predicate p(int* a[20], int len, int off) = \length(a) == len && len > 0 && \offset(a) == off && 0 <= off < len && ...`，适用于 `a` 可指在所指向区间的任意元素（20 个元素的 `int` 数组）上。

- 若变元不是指针类型，但其分量是指针，这个分量同样被看成数组。
- 在逻辑函数和谓词的定义式和引理的声明中，不能出现指针运算符“*”和取地址运算符“&”。作为指针类型的变元，它的声明中出现“*”则是另外一回事，因为这个“*”不是算符。这表明逻辑定义把变元声明 `int a[10][20]`和 `int a[][20]`（包括上述 `int* a[20]`）中的变元 `a` 真正看成数组，而不像 C 那样把 `a` 看成数组元素的指针。需要强调一下，这是对逻辑定义的限制。指针运算符“*”和取地址运算符“&”是可以出现在函数协议、循环不变式和程序点断言等程序员给出的断言中的。
- 在内建函数中已多次用到 `type* a` 这样的指针变元。例如 `predicate \is_pstring(char* str, int m)`。

把有数组变元的谓词定义和函数定义分成上述完整的数组、缺少数组大小的数组和指针三种形式，有助于程序员根据自己程序的特点选择合适的形式，使得所定义的逻辑函数和谓词清晰。

3.8 节有关字符串的内建函数和谓词为适用于各种场合，主要是用于设计 C 的字符串库函数的协议，没有区分成三种形式，并且允许字符指针变元对应的实变元是指针加或减偏移表达式的形式。

和字符串内建谓词一样，字符串指针作为自定义谓词 `pred` 的变元时，该变元也被认为是指向字符串中的第一个字符，不管是逻辑字符串还是物理字符串。也就是谓词 `pred` 应用中对应该变元的实变元 `str1`（或 `str1 ± e`）必须指针，即真正把 `str1` 看成是字符串指针。无论 `str1` 是否指向实际数据块的起始地址，只要标注的上下文中有相关的信息，系统都能妥善处理。

指向堆区指针在使用中也有一些特殊规定，见 8.3 节。

例 7.4 下面给出一个谓词定义和一个函数定义。

```
/*@ predicate is_positive(long x) = x > 0;
   logic int getSign(double x) = x > 0.0 ? 1 : (x < 0.0 ? -1 : 0);
*/
```

□

谓词也可以归纳地定义。一般而言，归纳谓词的定义有如下形式：

```
/*@ inductive P(x1, ..., xn) = p1 || ... || pk; */
```

其中各 `pi` 都是断言，部分 `pi` 中有谓词 `P` 的应用，即归纳定义。至少有一个 `pi` 不含 `P` 的应用。各 `pi` 中出现的 `P` 应用都体现出是基于同一个变元 `xj` 进行归纳的，不允许出现基于多个变元进行归纳的情况。

该定义的语义是，`P` 是这些情况的最小不动点，即 `P` 是满足 `P(x1, ..., xn) = p1 || ... || pk` 的最小谓词，也就是满足该式并且为假情况最多的谓词。一般而言，最小不动点并不一定存在，因此工具可能需要对归纳谓词定义的形式强加某些语法约束。

注意：要求谓词定义的右部写成析取范式的形式，即使某个子式中有该谓词的直接应用，仍然认为是析取范式。

例 7.5 例 7.1 和 7.2 都有归纳谓词。下面定义归纳谓词 `isGcd(x, y, d)`，`d` 是 `x` 和 `y` 的最大公约数。

```
/*@ inductive isGcd(long a, long b, long d) =
   b == 0 && a == d ||
```

```

        b != 0 && isGcd(b, a%b, d);
    */

```

逻辑函数也可以递归地定义，还可以相互地递归定义。相互递归的逻辑函数的定义必须出现在同一群逻辑声明中，即出现在同一个注释/*@ ... */中。

例 7.6 下面的标注定义一个逻辑函数，它返回 $i+1$ ，其中 i 是使得 $t[i]$ 等于 0 的最大下标 i ($0 \leq i \leq n-1$)。

```

/*@ logic int maxIndex(int t[], int n) =
    (n == 0) ? 0:
    (t[n-1] == 0) ? n : maxIndex(t, n - 1); */

```

例 7.7 下面是取非空二叉排序树上最小数据的逻辑函数。

```

/*@ logic int smallest(Node* p) =
    (p->l == \null) ? p->data : smallest(p->l); */

```

与归纳谓词一样，SCSL 对这样的递归函数定义没有语法约束（例如限制到原始递归）。同样，工具可能需要对递归函数定义的形式强加某些语法约束。

谓词定义的断言中和逻辑函数定义的项中，不能出现除了它们的变元以外的程序变量、逻辑变量和幽灵变量。谓词定义和逻辑函数定义正确与否由程序员负责。逻辑函数的使用环境正确与否也由程序员负责，例如，应用 `maxIndex` 函数时两个变元应该满足 $\text{length}(t) > n$ ，应用 `smallest` 函数时变元应该指向非空二叉排序树。好在，若函数应用的变元不在函数定义规定的范围内，则函数应用得不到结果。

描述链表的表段和二叉树的树段的性质时，相关的谓词定义和逻辑函数定义涉及该数据结构的两个指针。例如，安全 C 语言使用手册的例 3.6 给出了线索二叉树的定义，验证操作线索二叉树的函数时，还需要用到二叉树的树段概念，线索二叉树的树段定义如下：

```

inductive tTreeSeg(Node* p, Node* pred1, Node* succ1, Node* q,
                  Node* pred2, Node* succ2) =
    p == q && q != \null && pred1 == pred2 && succ1 == succ2 ||
    tTreeSeg(p->l, pred1, p, q, pred2, succ2) && (p->r_tag == Link && tTree(p->r, p, succ1) ||
    p->r_tag == Thread && p->r == succ1) ||
    tTreeSeg(p->r, p, succ1, q, pred2, succ2) && (p->l_tag == Link && tTree(p->l, pred1, p) ||
    p->l_tag == Thread && p->l == pred1);

```

这是从 p 指向节点到 q 指向节点的非空线索二叉树段， pred1 与 succ1 和 pred2 与 succ2 分别是 p 和 q 指向的线索二叉树段的前驱与后继。

在定义表段和树段的谓词和逻辑函数时，若描述从 p 指向节点到 q 指向节点的表段或树段，则在变元列表中， p 必须出现在 q 之前。

7.4 引理声明

引理是程序员给出的断言，用以支持自动定理证明器来证明其他引理和/或验证条件的有效性。引理声明中引理名之后的括号中若有 `both`，则表示该引理既用于其他引理的证明，也用于验证条件的证明。若括号中有 `relies on`，则表示该引理本身的证明依赖于若干引理。`both` 和 `relies on` 可同时出现。若引理名之后的括号中没有 `both`，则表示该引理仅一种用途：若发现该引理被依赖，则它仅用于引理证明，否则仅用于验证条件的证明。下面是引理声明的一个简单例子。

```

/*@ lemma meanProperty: \forallall long x.\forallall long y. x <= y ==> x <= (x+y)/2 <= y;

```

例 7.8 从例 7.2 的归纳定义，经结构归纳证明，可以得到下面的引理。

```

/*@ 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); */□
```

注意，与谓词定义和函数定义不同的是，引理声明中没有参数表。并且声明中出现的变元并非一定要像例 7.8 那样受量词约束。声明中的自由变元按作用域规则联系到 C 代码中同样名字的声明。

SCSL 规范的完备验证必须完成对每个引理的证明。若某个引理 A 的证明依赖于引理 B 和 C，则引理 A 的声明宜写成：

```
lemma A (relies on : B, C) :
```

```
\forall ... ... ; //本行是代表引理 A 的断言。
```

若 B 是在标注中声明的引理，则按作用域原则，B 的声明在此处必须是可见的。若 B 是内建引理，则无须在标注中出现其声明。这句话也适用于 C。

把一个引理的证明需要用到的引理列出来，可以减少系统在证明引理方面的时间开销。例 7.8 两个引理的证明并不依赖于其它引理，因此它所依赖的引理表为空。

为便于系统自动证明引理，对引理的描述有下面几点限制或建议。

1. 引理中的整型约束变元，若其取值范围是闭区间的话，尽量写成显式的区间约束，例如 `\forall int i:[0..99].P(i)`，不必写成 `\forall int i. 0<=i<=99 ==> P(i)` 的形式。

2. 引理中不同位置出现的常数，若它们在引理中的含义是相关联的，则必须用 C 的同一个宏定义或其他方式来表达它们的相关性。例如，

```
#define N 100
typedef int Dis[N];
logic int count(Dis f, int m, int n) = n < m ? 0 : (count(f, m, n-1) + f[n-1]);
lemma counting: \forall Dis f. \forall Dis oldf. \forall int u.
    (\forall int i:[0..u-1].oldf[i] == f[i]) && oldf[u] == 0 && f[u] == 1 &&
    (\forall int i:[u+1..N-1].oldf[i] == f[i]) && count(oldf,1,N) == i
    ==> count(f,1,N) == i+1;
```

而不要把引理写成：

```
lemma counting: \forall Dis f. \forall Dis oldf. \forall int u.
    (\forall int i:[0..u-1].oldf[i] == f[i]) && oldf[u] == 0 && f[u] == 1 &&
    (\forall int i:[u+1..99].oldf[i] == f[i]) && count(oldf,1,100) == i
    ==> count(f,1,100) == i+1;
```

后者有可能导致引理自动证明的失败。

3. 所提供的引理能满足所在源文件或函数的验证即可，不必提供更一般的引理。例如，若一个引理仅用于某个 C 函数中某个或某些验证条件的证明，则把它声明在该函数的内部，放在声明语句之后的某个地方。

这样做的的好处是，可以把引理中用到的该函数的形参和局部变量作为引理的自由变元，免去对它们加全称量化约束，便于对该引理的自动证明和在验证过程中使用该引理，尤其当它们是作为字符串的数组类型时。

例 7.9 图 7.5 是双向链表倒置函数，它从后向前遍历链表，互换每个节点的 l 域和 r 域的值（即互换指向的节点）。在该图中有 4 个谓词定义 `inc_sorted_seg`、`inc_sorted`、`almost_inc_sorted`、`dec_sorted_seg` 和 `dec_sorted`，它们分别表示递增表段、递增链表、接近递增链表、递减表段和递减链表。在倒置过程中，呈现出看似方向相反的两个表段，但它们的 forward 方向一致。根据 8.3.2 节有关 forward 的描述规定，在本例函数的形参声明之后增加了 r 作为节点 forward 域的标注。

用一个指针 p 遍历链表，一般都是从左向右地推进的（按我们画示意图的习惯），但链表和表段的定义一般都是从右向左归纳的。图 7.5 中关于递增链表的表段的归纳定义 inc_sorted_seg ，是从 $inc_sorted_seg(p \rightarrow r, q)$ 加上一些适当的断言后得到 $inc_sorted_seg(p, q)$ 。

从第一个循环可以看到，指针 p 从左向右推进时，在已扫描表段的右边每次增加一个节点，形成右边拼加一个节点的表段，那就需要证明“表段+节点”得到“多一个节点的表段”。这个结论是显然的，但由于方向不一致，基于表段的定义很难证明它，导致需要内建引理。

是否可以让表段的定义也从左向右地归纳，使得定义和遍历的方向一致，以达到不必使用引理的目的。从概念上是可以的，但写表段的定义有难度。这时需要写成 $inc_sorted_seg(p, q)$ 的前驱节点 + q 指向的节点，得到 $inc_sorted_seg(p, q)$ 。不使用存在量词，表达不了 q 的前驱节点。而增加存在量词就有可能对引理的证明和使用带来影响。

在本例第二个循环的迭代计算中，头指针为 $list$ ，尾指针为 ptr 的递增链段在缩短，而头指针为 $last$ ，尾指针为 ptr 的递减表段在增长。这两个表段都是在尾部进行删或增，同样要用到内建引理。 □

```
// 采用自定义谓词和自定义引理的情况
/*@ inductive inc_sorted_seg(ListNode * p, ListNode * q) =
    p == q && p != \null ||
    p != q && p->r->l == p && p->data <= p->r->data && inc_sorted_seg(p->r, q);
predicate inc_sorted(ListNode * p) =
    p == \null || p->l == \null && p->r == \null ||
    p->l == \null && \exists ListNode* q.inc_sorted_seg(p, q) && q->r == \null;
predicate almost_inc_sorted(ListNode * p) =
    \exists ListNode* q. inc_sorted_seg(p, q) && q->r == \null;
inductive dec_sorted_seg(ListNode * p, ListNode * q) =
    p == q && p != \null ||
    p != q && p->r->l == p && p->data >= p->r->data && dec_sorted_seg(p->r, q);
predicate dec_sorted(ListNode * p) =
    p == \null || p->l == \null && p->r == \null ||
    p->l == \null && \exists ListNode* q.dec_sorted_seg(p, q) && q->r == \null;
predicate almost_dec_sorted(ListNode * p) =
    \exists ListNode* q. dec_sorted_seg(p, q) && q->r == \null;

lemma inc_sorted_seg_push_back_lemma: \forallall ListNode *p, ListNode *q.
    inc_sorted_seg(p, q) && q->r->l == q && q->data <= q->r->data ==> inc_sorted_seg(p, q->r);
lemma dec_sorted_seg_push_back_lemma: \forallall ListNode *p, ListNode *q.
    dec_sorted_seg(p, q) && q->r->l == q && q->data >= q->r->data ==> dec_sorted_seg(p, q->r);
lemma inc_sorted_seg_pop_back_lemma: \forallall ListNode *p, int n.
    n >= 0 && inc_sorted_seg(p, p->(r:n + 1)) ==> inc_sorted_seg(p, p->(r:n));
lemma inc_sorted_seg_back_lemma: \forallall ListNode *p, int n.
    inc_sorted_seg(p, p->(r:n + 1)) && n >= 0 ==> p->(r:n)->data <= p->(r:n + 1)->data;
lemma dec_sorted_lemma: \forallall ListNode *p, ListNode *q.
    dec_sorted_seg(p, q) && p->l == \null && almost_dec_sorted(q) ==> dec_sorted(p);
*/
```

图 7.5-1 双向链表倒置函数-使用自定义谓词和引理

```

#include <stdlib.h>
typedef struct listnode { int data; struct listnode*r;  struct listnode* l;}ListNode;
//@ shape l, r: dlist;

/*
 自定义谓词和引理的声明与定义（见图 7.5-1）
*/
//@ logic integer n;
/*@ requires \length(list, r) == n && inc_sorted(list);
    assigns *list;
    ensures \length(\result, r) == n && dec_sorted(\result) &&
           (n > 0 && list == \result->(r:n-1) || n == 0 && list == \null);  */
ListNode * invert(ListNode * const list) /*@ shape ListNode forward: r;*/ {
  ListNode *last, *ptr, *tmp;  //@ ghost int j;
  last = list;
  if (last != NULL) {          //@ ghost j = 0;
    /*@ loop invariant 0 <= j < n && \dangling(ptr) && \dangling(tmp) &&
        list->l == \null && inc_sorted_seg(list, last) && last == list->(r:j) &&
        almost_inc_sorted(last) && \length(last, r) == n - j; */
    while (last->r != NULL) { last = last->r;  //@ ghost j = j + 1;  }
    ptr = last;    tmp = NULL;    //@ ghost j = 0;
    /*@ loop invariant
        n > 0 && j == 0 && list->l == \null && inc_sorted_seg(list, ptr) &&
        ptr == list->(r:n-1) && ptr == last && last->r == \null && tmp == \null ||
        0 < j < n && list->l == \null && inc_sorted_seg(list, ptr) && ptr == list->(r:n-j-1) &&
        tmp == ptr->r && tmp->r == ptr && ptr->data <= tmp->data &&
        dec_sorted_seg(last, tmp) && tmp == last->(r:j-1) && last->l == \null ||
        n > 0 && j == n && list->r == \null && dec_sorted_seg(last, tmp) &&
        tmp == last->(r:n-1) && tmp == list && last->l == \null && ptr == \null; */
    while (ptr != NULL) {
      tmp = ptr->r; ptr->r = ptr->l; ptr->l = tmp; tmp = ptr; ptr = ptr->r;  //@ ghost j = j + 1;
    }
  }
  return last;
}

```

图 7.5-2 双向链表倒置（使用自定义谓词与引理）

```

#include <stdlib.h>
typedef struct listnode { int data; struct listnode*r; struct listnode* l;}ListNode; // @ shape l, r: dlist;
// @import \dist_module< ListNode( r:r, l:l )>
/* @
    using \inc_sorted_dlist_seg_push_back_lemma<ListNode>;
    using \dec_sorted_dlist_seg_push_back_lemma<ListNode>;
    using \inc_sorted_dlist_seg_pop_back_lemma<ListNode>;
    using \inc_sorted_dlist_seg_back_lemma<ListNode>;
    using \dec_sorted_dlist_lemma<ListNode>; */
// @ logic integer n;
/* @
    requires \length(list, r) == n && \inc_sorted_dlist(list);
    assigns *list;
    ensures \length(\result, r) == n && \dec_sorted_dlist(\result)
           && (n > 0 && list == \result->(r:n-1) || n == 0 && list == \null);
*/
ListNode *invert(ListNode * const list) /* @ shape ListNode forward: r; */ {
    ListNode *last, *ptr, *tmp; // @ ghost int j;
    last = list;
    if (last != NULL) {
        // @ ghost j = 0;
        /* @ loop invariant 0 <= j < n && \dangling(ptr) && \dangling(tmp)
           && list->l == \null && \inc_sorted_dlist_seg(list, last) && last == list->(r:j)
           && \almost_inc_sorted_dlist(last) && \length(last, r) == n - j; */
        while (last->r != NULL) {
            last = last->r; // @ ghost j = j + 1;
        }
        ptr = last; tmp = NULL; // @ ghost j = 0;
        /* @ loop invariant
           n > 0 && j == 0 && list->l == \null && \inc_sorted_dlist_seg(list, ptr)
           && ptr == list->(r:n-1) && ptr == last && last->r == \null && tmp == \null
           || 0 < j < n && list->l == \null && \inc_sorted_dlist_seg(list, ptr)
           && ptr == list->(r:n-j-1) && tmp == ptr->r && tmp->r == ptr
           && ptr->data <= tmp->data && \dec_sorted_dlist_seg(last, tmp)
           && tmp == last->(r:j-1) && last->l == \null
           || n > 0 && j == n && list->r == \null && \dec_sorted_dlist_seg(last, tmp)
           && tmp == last->(r:n-1) && tmp == list && last->l == \null && ptr == \null;
        */
        while (ptr != NULL) {
            tmp = ptr->r; ptr->r = ptr->l; ptr->l = tmp; tmp = ptr; ptr = ptr->r; // @ ghost j = j + 1;
        }
    }
    return last;
}

```

图 7.5 双向链表倒置（使用内置谓词与引理）

7.5 高阶逻辑构造

项 $\lambda x_1 \dots x_n. t$ 称为抽象，它代表 n 元的逻辑函数，其参数是 x_1, \dots, x_n ，它们的类型分别是 τ_1, \dots, τ_n ，函数体是 t 。 λ 的优先级同 \forall 和 \exists 的一样。在 SCSL 中， λ 抽象作为函数类型的表达式，它目前仅用作下面所列扩展范域词的第 3 个变元。

\forall 、 \exists 和 λ 统称为范域词 (*quantifier*)，其中全称范域词 \forall 和存在范域词 \exists 习惯称为全称量词 (*universal quantifier*) 和存在量词 (*exist quantifier*)， λ 可称为抽象范域词。在形式为 $\text{quant}(t_1, t_2, t_3)$ 的项中， quant 是扩展范域词，目前只有 \max 、 \min 、 sum 、 product 和 numof 五个扩展范域词。变元 t_1 和 t_2 必须是 integer 类型或它子类型， t_3 必须是变元为 integer 类型或它子类型的一元函数。在 numof 中，一元函数 t_3 的结果是布尔值，其余各函数的一元函数 t_3 的结果是 integer 或 real 类型或它们的子类型。函数 t_3 的类型不是显式声明的，它是由对 t_3 表达式本身进行推导得到的。它们的含义如下：

$$\begin{aligned} \backslash\max(i, j, f) &= \max\{f(i), f(i+1), \dots, f(j)\} \\ \backslash\min(i, j, f) &= \min\{f(i), f(i+1), \dots, f(j)\} \\ \backslash\text{sum}(i, j, f) &= f(i) + f(i+1) + \dots + f(j) \\ \backslash\text{product}(i, j, f) &= f(i) \times f(i+1) \times \dots \times f(j) \\ \backslash\text{numof}(i, j, f) &= \#\{k \mid i \leq k \leq j \ \&\& \ f(k)\} = \\ &\quad \backslash\text{sum}(i, j, \lambda \text{int } k. f(k) ? 1 : 0) \end{aligned}$$

如果 $i > j$ ，那么 $\backslash\text{sum}(i, j, f)$ 和 $\backslash\text{numof}(i, j, f)$ 等于 0， $\backslash\text{product}(i, j, f)$ 等于 1， $\backslash\max(i, j, f)$ 和 $\backslash\min(i, j, f)$ 的值没有规定。

例 7.10 图 7.6 是对双精度数组的元素求和的例子。 □

```
#define Max 10000
/*@ requires Max >= n >= 0 && \length(t) == n;
    ensures \result == \sum(0, n-1, \lambda int k. t[k]) && \length(t) == n;
*/
double array_sum(double* const t, int const n) {
    int i; double s = 0.0;
    /*@ loop invariant 0 <= i <= n <= Max && s == \sum(0, i-1, \lambda int k. t[k]) && \length(t) == n;
        loop variant n - i;
    */
    for(i = 0; i < n; i++) s = s + t[i];
    return s;
}
```

图 7.6 一维数组求和

例 7.11 图 7.7 是矩阵相乘函数，使用了谓词定义和高阶逻辑构造。

```

const int m = 100; const int k = 200; const int n = 300;
typedef int AB[100][200];
typedef int BC[200][300];
typedef int AC[100][300];
AC Z = {0};
/*@ predicate element(AB X, BC Y, AC Z, int i, int j, int l) =
    (Z[i][j]==\sum(0, l-1, \lambda int l1.X[i][l1]*Y[l1][j]))&&-3000*3000*1 <= Z[i][j] <= 3000*3000*1;
    // 元素 Z[i][j]的值的计算已经累加了 X[i][l-1] * Y[l-1][j]
predicate column(AB X, BC Y, AC Z, int i, int j) =
    \forall int j1: [0..j-1]. element(X, Y, Z, i, j1, \dimension(X, 2));
    // 在第 i 行中, 元素 Z[i][0], ..., Z[i][j-1]的值都已算好
predicate row(AB X, BC Y, AC Z, int i) =
    \forall int i1: [0..i-1]. column(X, Y, Z, i1, \dimension(Y, 2));
    // 数组 Z 从 0 到 i-1 各行的元素的值都已算好
predicate matrixMultiply(AB X, BC Y, AC Z) = row(X, Y, Z, \dimension(X, 1));
    // 数组 Z 的所有行都已经算好 */
/*@ requires m == 100 && k == 200 && n == 300 && \length(X) == m && \length(Y) == k &&
    (\forall int i:[0..99]. \forall int j:[0..199]. -3000 <= X[i][j] <= 3000) &&
    (\forall int i:[0..199]. \forall int j:[0..299]. -3000 <= Y[i][j] <= 3000);
    assigns Z;
    ensures matrixMultiply(X, Y, Z) && \length(X) == m && \length(Y) == k; */
void MatrixMultiply1(int (*const X)[200], int(*const Y)[300]) {
    int i, j, l;
    /*@ loop invariant 0 <= i && i <= m && row(X, Y, Z, i) && m == 100 && k == 200 && n == 300 &&
        \length(X) == m && \length(Y) == k &&
        (\forall int i:[0..99]. \forall int j:[0..199]. -3000 <= X[i][j] <= 3000) &&
        (\forall int i:[0..199]. \forall int j:[0..299]. -3000 <= Y[i][j] <= 3000);
    loop variant m - i; */
    for (i = 0; i < m; i = i + 1) {
        /*@ loop invariant 0 <= i && i < m && row(X, Y, Z, i) && 0 <= j && j <= n &&
            column(X, Y, Z, i, j) && m == 100 && k == 200 && n == 300 && \length(X) == m &&
            \length(Y) == k &&
            (\forall int i:[0..99]. \forall int j:[0..199]. -3000 <= X[i][j] <= 3000) &&
            (\forall int i:[0..199]. \forall int j:[0..299]. -3000 <= Y[i][j] <= 3000);
        loop variant n - j; */
        for (j = 0; j < n; j = j + 1) {
            Z[i][j] = 0;
            /*@ loop invariant 0 <= i && i < m && row(X, Y, Z, i) && 0 <= j && j < n &&
                column(X, Y, Z, i, j) && 0 <= l && l <= k && element(X, Y, Z, i, j, l) &&
                m == 100 && k == 200 && n == 300 && \length(X) == m && \length(Y) == k &&
                (\forall int i:[0..99]. \forall int j:[0..199]. -3000 <= X[i][j] <= 3000) &&
                (\forall int i:[0..199]. \forall int j:[0..299]. -3000 <= Y[i][j] <= 3000); */
            for (l = 0; l < k; l = l + 1) { Z[i][j] = Z[i][j] + X[i][l] * Y[l][j]; }
        }
    }
    return;
}

```

图 7.7 矩阵相乘函数及协议

7.6 宏定义

宏定义和宏应用类似 C 语言中的宏定义和宏调用，也分成无参数和有参数两种情况，在语法上和 C 的略有区别（见图 7.1）。

1. 无参数的宏定义

无参数的宏定义相当于用一个无参谓词名来代表一个较长的断言，它在多层嵌套循环中比较有用。

对于多层嵌套循环，若外层循环有较多的不变性不受内层循环的影响，则它们都要作为内层循环的循环不变式的一部分，否则外循环的循环体出口处的验证条件难以得到证明。在循环嵌套层次较多时，若没有一种简写的方式，则会使得内层循环的循环不变式越来越长，书写不方便且影响可读性。若不受当前循环影响的不变性用一个宏名代替，则当前循环不变式中的其余部分都是与当前循环有关的不变性质，也便于自己检查循环不变式正确与否。

一个典型的例子是矩阵分块乘法的验证，见本系统所提供的参考实例。

2. 带参数的宏定义

这里宏定义的语法和 C 的略有区别，其原由是出于带参数的情况，避免宏替换 Macro (a, b) 把宏定义

```
define Macro (x, y): assertion1 && x == y ==> assertion2
```

当成一个无参的宏定义（因为 Macro 和 (x, y) 之间有空格），把 Macro 替换成

```
(x, y): assertion1 && x == y ==> assertion2 (a, b)
```

而不是把 Macro (a, b) 替换成

```
assertion1 && a == b ==> assertion2
```

7.7 常用逻辑定义和引理组成的逻辑模块及其使用

对于在程序验证中常用的一些逻辑定义和引理，按照 C 语言的类型，或类型的运算，或基于 C 的类型构造的数据结构，分门别类地组织成各种逻辑模块，以方便写标注时引用它们。一个逻辑模块中除了有引理外，还可能有这些引理中用到的逻辑定义，一个逻辑模块中也可能只有逻辑定义而无引理。

导入声明的关键字 `import` 之后跟随内建模块名，即图 7.7 文法中第 1 个 `id`，可能有的第 2 个 `id` 是该程序中自引用结构体的类型名，可能有的随后的 `id: id` 是指模块中自引用结构体和程序中自引用的结构体类型之间域名的对应。这样将 `id: id` 中的对应顺序规定下来，即第一个 `id` 必须是模块中的域名，第二个 `id` 必须是程序中自引用结构体的域名。导入声明表示把这些模块的逻辑定义和引理依次展开在这个导入声明处，取代导入声明。并依据声明中的域名对应表，进行模块中自引用的结构体类型中的域名到程序中自引用结构体域名的代换。展开后的逻辑定义和引理的作用域遵循 7.1 节的规定。

使用声明的关键字 `using` 之后跟随内建引理名以及可能有的该程序中自引用结构体的类型名。与导入声明不一样的是，一个 `using` 之后可以跟随若干个内建引理。另一个不同是这里不需要给出域名之间的对应，因为在导入声明时已经给出。使用声明表示代码中相关位置验证条件的证明很可能要用到这些内建引理。若使用声明出现在函数内部，则表示从这个位置到函数末尾的验证条件，系统都会加上这个声明中的内建引理来证明这些验证条件。若使用声明出现在函数外部，则表示从这个位置到文件末尾的验证条件，系统都会加上这个声明中的内建引理来证明这些验证条件。

在图 3.1 项文法和图 3.2 断言文法中出现的 `\result`、`\old`、`\dimension`、`\pointer`、`\list`、`\c_list`、`\is_string` 和 `\prefix` 等逻辑定义，都是系统内建的定义，它们无须导入就能使用。

可供程序员导入并使用的逻辑模块请见附录一。这些模块中的引理是已经证明过的，导

入时无需证明。在使用导入声明时,请确认拟导入模块中的逻辑定义和引理是否为自己所需。若略有差异,例如变元的类型或表达式中的算符不同,则不要导入该模块。

第 8 章 数据不变式和形状标注

程序员若用标注指明某类型的所有变量在其生命周期中始终存在的性质,或者并非始终存在但在关键程序点存在的性质,这会给程序验证带来方便。这样的标注称为类型不变式。程序员也可以标注一个类型的某个变量所具备的特殊性质,这样的标注称为变量不变式。类型不变式和变量不变式统称为数据不变式。

对于用以构造易变数据结构节点的数据类型,程序员需要标注这些数据类型用以构造的易变数据结构的形状等信息。这种标注称为形状标注。

8.1 类型不变式

类型不变式可以标注到全局定义的类型 τ 上, τ 类型的任何外部变量和静态外部变量(也包括外部变量和静态外部变量的域或数组元素等属于 τ 类型的情况)都遵守类型不变式的规定。类型不变式有强弱之分。

1. 强类型不变式必须在程序执行的任何时候(更精确地说,是在 C 标准定义的任何顺序点(sequence point))都有效。若该函数的返回值属于 τ 类型,则返回值在函数出口必须满足 τ 的强类型不变式。

2. 弱类型不变式必须在函数的边界(函数的入口和出口)有效,但不要求在这两点之间保持有效。若该函数的返回值属于 τ 类型,则返回值在函数出口必须满足 τ 的弱类型不变式。

上述 τ 类型的变量必须静态置初值。另外,若是函数形参(包括其域或数组元素等属于 τ 类型的情况),则形参必须满足对 τ 的弱类型不变式的要求。

分配在栈区的 τ 类型的变量,不被认为具有 τ 的类型不变式所描述的性质,因为在声明该变量的函数入口,该变量尚未出现。因此 τ 的类型不变式对这样的变量来说,相当于什么也没有说。

对于一个有类型不变式的构造类型,它的域可以属于有类型不变式的类型,这两个类型不变式的强弱必须一致。另一方面,任何构造类型(包括构造类型中仅一个域的情况),都不允许同时受到多个类型不变式的约束。

类型不变式的文法在图 8.1 中,它正好出现在相应的类型定义之后。强度修饰符缺省则默认为弱不变式。

<i>C-declaration</i>	→	<i>/*@ data-inv-decl */</i>
<i>data-inv-decl</i>	→	<i>data-invariant type-invariant</i>
<i>data-invariant</i>	→	<i>int-strength[?] global invariant id : assert ;</i>
<i>type-invariant</i>	→	<i>int-strength[?] type invariant id (C-type-name id) = assert ;</i> <i> int-strength[?] type invariant id (C-type-name *id) = assert ;</i>
<i>int-strength</i>	→	<i>weak strong</i>

图 8.1 数据不变式的文法

例 8.1 下面是类型不变式的 4 个例子。

(1) 类型为 `temperature` 的变量的强类型不变式如下:

```
typedef double temperature;  
/*@ strong type invariant temp_in_celsius (temperature t) = t >= -273.15;
```

(2) 类型为 `struct S` 的变量的弱类型不变式如下:

```
struct S {  
    int f;  
};
```

```
//@ type invariant S_f_is_positive(struct S s) = s.f >= 0;
```

(3) 若类型为 `struct S` 的变量的类型不变式修改如下：

```
struct S {  
    temperature f;  
};  
//@ strong type invariant S_f_is_positive(struct S s) = s.f >= 0.0;
```

若上述 (1) 和 (3) 两个类型及类型不变式声明同时出现时，则是一个错误。因为类型 `temperature` 有两个类型不变式，即使这两个不变式没有矛盾也不接受。

(4) 对于指针类型 τ_1 ，其类型不变式主要描述 τ_1 指针所指向数据区的长度和指针的偏移，而 τ_1 指针所指向对象类型 τ_2 的不变式，应该描述在 τ_2 类型处，因为这是另一个类型。

```
#define N 100  
typedef enum {BLUE, WHITE, RED} color;  
//@ type invariant isColor(color c) = c == BLUE || c == WHITE || c == RED;  
typedef struct flag{int n; color* colors;}flag;  
/*@ strong type invariant is_colored(flag f) =  
    0 < f.n <= N && \length(f.colors) == f.n && \offset(f.colors) == 0 &&  
    \forall integer k. 0 <= k <= f.n <= N ==> isColor(f.colors[k]);  
*/
```

注意，上述最后一行的 `isColor(f.colors[k])` 是 `color` 类型的类型不变式 `isColor(color c)` 的应用，而不是定义指针 `color* colors` 的类型不变式。并且 `flag` 类型有 `colors` 指针域，其指向对象类型 `color` 是有类型不变式的，因此类型不变式中必须出现 `color` 类型不变式的应用。这里不是 (3) 所说的某个类型存在两个类型不变式。

由此可知，对于自引用的结构体类型 τ ，若有类型不变式，则应该定义在 τ 类型处。□
对类型不变式的语法限制如下。

1. 构造类型 `A` 的分量（包括构造类型和指针类型的分量）若有不变式，则这些不变式都是 `A` 的类型不变式的一部分。若类型 `A` 还是某个类型的某个分量的类型，则 `A` 的类型不变式就是该类型的不变式的一部分。

2. 若 `A` 类型的分量 `B` 是指针类型，指向的对象类型为 `C`，则需要把 `C` 类型的不变式作用到 `B` 指向的对象之后的公式作为 `A` 类型的不变式的一部分。除非 `C` 类型没有不变式。见例 8.1 的 (4)。

3. 若 `A` 类型的分量 `B` 属于构造类型 `C`，则需要把 `C` 类型的不变式作用到 `B` 之后的公式作为 `A` 类型的不变式的一部分。除非 `C` 类型没有不变式。见例 8.1 的 (3)。

在函数协议的前后条件中，程序员不必给出相关变量的类型不变性，系统会自动检查。对于强类型不变性，在函数体中的循环不变式和程序点断言中，程序员也不必给出相关变量的强类型不变性。

8.2 变量不变式

变量不变式可以标注到外部变量、静态外部变量和静态局部变量，它们都是静态分配的。变量不变式表达在其生命周期中始终存在的性质。变量不变式也有强弱之分。

1. 强变量不变式表示被标注的变量在程序执行的任何时刻都有效。因此强变量不变式表示这些变量在执行过程中的任何一步都必须保持的性质。

2. 弱变量不变式表示被标注的变量在函数的边界有效，但不要求其在这两点之间保持有效。因此弱变量不变式表示这些变量在任何函数的入口和出口的性质。

无论是带上述哪种变量不变式，变量都必须在声明时置初值。一个例外是在多源文件共

享时，变量不变式定义在共享头文件中该变量的 `extern` 声明处，变量的初值则在变量定义的源文件中给出，如例 8.2 中的例 2。

3. 和类型一样，一个变量也不可以有多个变量不变式，还不允许同时受类型不变式和变量不变式的约束。

4. 变量不变式是否需要出现在函数协议和语句标注中，与类型不变式的情况一致。

变量不变式的文法也在图 8.1 中，一般来说，它也是正好出现在相应变量的声明之后。特殊情况是，变量 `V` 的不变式中出现以 `V` 为实变元的某个逻辑定义的应用，而该逻辑定义以 `V` 的类型为其变元的类型，这时该逻辑定义可以出现在变量 `V` 的声明和它的不变式之间。强度修饰符缺省则默认为弱不变式。

例 8.2 下面是变量不变式的两个例子。

1. 全局弱变量不变式 `a_is_positive`，它表示全局变量 `a` 必须维持为正的，但是 `a` 在函数调用和返回的区间可以违反这个性质。

```
int a = 1;
/*@ global invariant a_is_positive: a > 0;
```

2. 全局强变量不变式 `m_eq_100`，它表示全局变量始终等于 100，任何情况下不会改变。稍许复杂一些，它涉及 3 个 C 源文件 `const.h`、`const1.c` 和 `const2.c`，它们有关的内容分别如下：

```
const.h:
extern int m;
/*@ strong global invariant m_eq_100: m == 100; // 标注出现在头文件中了，
... .. // 包含该文件的其他文件就不
const1.c: // 用再为 m 定义全局不变式，
#include "const.h" // 定义了反而会报错。
int m == 100; // 在本文件中定义 m，但无需标注。
... ..
const2.c:
#include "const.h" // 该文件中引用 m，也无需标注。
... ..
```

例 8.3 图 8.2 是变量不变式用于静态局部变量的例子。

```
void out_char(char c) {
    static int col = 0;
    /*@ global invariant I : 0 <= col <= 79;
    col ++;
    if (col >= 80) col = 0;
}
```

图 8.2 一个静态局部变量的不变性

第 9 章还有一些例子用到数据不变式。

8.3 易变数据结构的形状标注

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的形状特征。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，控制住易变数据结构的复杂性，提高指针操作程序的合法性门槛，便于排除没有构造出（或没有操作在）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用于构造的易变数据结构的形状。易变数据结构的形状声明可类比变量的类型声明，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。

在程序验证中，形状声明有助于程序员在函数前后断言、循环不变式和程序点断言中描述单态和多态命名基本形状中各节点的指针特性，还有助于程序验证系统发现程序中的指针错误。

形状系统在安全 C 语言使用手册的第 3 章已经介绍，本节介绍怎样在易变数据结构的节点的类型声明和相关的指针变量声明之后进行形状标注。形状标注的文法见图 8.4。

8.3.1 描述形状性质的指针断言的种类及基本限制

对于程序员给出的、描述形状性质的指针断言，包括对出现在节点数据性质断言中的指针类型的前缀（以下都简称指针断言），存在一些限制。

首先对作为指针的堆指针访问路径作一些限制。

<i>C-declaration</i>	→	<i>/*@ shape_desc */</i>
<i>C-direct-declarator</i>	→	<i>direct-declarator (parameter-type-list²)</i> <i>/*@ shape_desc (shape_desc)* */</i> // 形参、\result 和所用全局变量的标注
<i>shape_desc</i>	→	<i>shape (C-field-name (, C-field-name)[*])[?] : shape_decl (, feature)[?];</i> <i>shape C-variable-name (, C-variable-name)[*] : data_block;</i> <i>shape C-field-name : tag;</i> <i>shape C-type-name forward : C-field-name;</i> <i>shape C-variable-name (, C-variable-name)[*] : additional_list;</i>
<i>shape_decl</i>	→	<i>list c_list dlist c_dlist tree data_block no_name</i> <i>polymorphic_data_block polymorphic_no_name</i>
<i>feature</i>	→	<i>primary shared_data secondary to_shared additional predicate invariant</i>

图 8.4 形状标注的文法

(1) 在程序员用断言整体描述（见 8.3.3 节）双向链表和带环链表的形状时，访问路径不能出现环。例如，双向链表上， $p \rightarrow r \rightarrow l \rightarrow r$ 和 $p \rightarrow r \rightarrow r \rightarrow l \rightarrow l \rightarrow r$ 都有环，因为它们都是语义上与 $p \rightarrow r$ 一样的访问路径。

有一种情况不受这个限制。在定义谓词、归纳谓词和逻辑函数以及声明引理时，允许出现 $p \rightarrow r \rightarrow l$ 这样的访问路径。以例 7.6 双向链表倒置的图 7.5 为例，其中的归纳谓词定义和引理声明中都出现断言 $p \rightarrow r \rightarrow l \implies p$ ，其中构成小环的访问路径 $p \rightarrow r \rightarrow l$ 是无法用其它无环访问路径替代的。好在这是在定义或声明中用于表达构成小环这种性质，不是在程序点的断言中。

(2) 若访问路径带折迭域，例如， $p \rightarrow (\text{next:e}) \rightarrow \text{data}$ ，其中 e 是非负线性整型表达式并且其中不含对堆指针指向对象的访问，则称该访问路径为含折迭域的访问路径。一条堆指针类型的访问路径上最多出现一次折迭域，并且该指针域是这条访问路径的最后一个指针域。

对于作为节点数据域访问路径前缀的堆指针访问路径同样受上述两点限制。

对于程序员在标注中给出的堆指针访问路径，包括其后还有访问其指向节点数据域的情况，再增加下面两点约定。

(1) 在程序员给出的标注中，若从 p 指向节点到 q 指向节点有一条路径，则称 q 依赖于 p 。

(2) 在谓词定义、性质引理和逻辑函数的定义中，以及在函数前后条件、循环不变式

和程序点断言中，若在一组合取的断言中出现 $p \rightarrow next$ 或 $p \rightarrow data$ 这样的访问路径，则默认在这组合取的断言中有合取子句 $p \neq \text{null}$ ，不管该断言是否显式给出。经常需要提醒，这里的 p 并非一定是指针型的声明变量，它可以是复杂一些的指针型访问路径。

指针断言最终体现出的是表达指针相等与否的断言（包括是否等于 null ，是否为悬空指针，是否为单节点指针），从语法特点上说，可以概括为以下几种形式。

(1) 简单指针断言

直接是指针相等与否断言，等号或不等号的两边是指针型简单访问路径。简单访问路径是指访问路径中没有程序函数和幽灵函数的应用，没有逻辑函数应用，也不出现折迭域。

(2) 含折迭域的指针断言

在简单指针断言中出现含折迭域的访问路径。对于简单指针断言和含折迭域的简单指针断言，建议把相对简单的访问路径（尤其是指针型声明变量时）放在等号或不等号的左边。

(3) 含逻辑函数应用的指针断言

在指针相等与否的断言中，若等号或不等号两边的访问路径表达式含结果是指针类型的逻辑函数应用，则称它为含逻辑函数应用的指针断言。例如 $p == \text{rightMost}(\text{succ} \rightarrow l)$ ，其中 rightMost 的定义是：

```
logic Node* rightMost(Node* p) = p->r_tag == Thread ? p : rightMost(p->r);
```

(4) 用谓词（包括归纳谓词）表达的指针断言

最简单的是谓词 $\text{dangling}(p)$ 和 $\text{singleton}(p)$ 。复杂一些的情况是谓词中有指针相等与否断言。例如，从根节点 p 开始的非空线索二叉树的定义如下：

```
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));
```

其中， pred 和 succ 分别是 p 指向线索二叉树段的前驱和后继。注意，不是表达指针相等与否的断言可以出现在这样的谓词中。

几点限制如下。

- 内建形状谓词（例如 $\text{dangling}(p)$ 和 $\text{singleton}(p)$ ）不能出现在程序员定义的谓词和归纳谓词中。
- 对于程序员定义的谓词，若有 p 和 q 两个指针变元，并且 q 依赖于 p ，则在谓词的变元表中， p 必须先于 q 。
- 在谓词定义中，不能出现没有依赖关系的指针进行比较。没有依赖关系意味着它们指向不同的易变数据结构。

(5) 量化指针断言

对于 $\text{forall integer } i: [0..99]. (\dots)$ 这样的全称量化断言（其中 (\dots) 是该全称量化的辖域）， (\dots) 要符合上面的约定，也要符合 7.3 节对归纳谓词的限定。例如图 7.4 的 $(\text{forall int } i: [1..m-1]. \text{head} \rightarrow (\text{next}: i-1) \rightarrow \text{data} \leq \text{head} \rightarrow (\text{next}: i) \rightarrow \text{data})$ 是可以接受的，因为它符合上面的约定。还有，矩阵分块乘法中的 $\text{forall int } i1: [0..s1-1]. \text{matrixColumn}(X, Y, Z, b, i1, \text{COLUMN2}/b)$ （见本系统所提供的参考实例），辖域中有归纳谓词的应用，这个归纳谓词的归纳符合 7.3 节对归纳谓词的限定，它是基于 matrixColumn 的第 6 个变元进行归纳的。

从演绎推理的角度，对于程序员给出的指针断言，还有下面两点约定。

- (1) 在含 p 的断言中，若没有并且也推导不出断言 $p == \text{null}$ ，则认为 p 是有效指针，除非存在用谓词表达的断言 $\text{dangling}(p)$ 。
- (2) 未在易变数据结构的谓词定义中用断言表示为相等的两个指针，若用演绎推理也

不能证明它们相等，则它们一定不相等。

以二叉树的谓词定义

```
tree(Node *p) = p == \null || tree(p->left) && tree(p->right)
```

来说明为什么要强调这一点。从这个定义是不可能推出 $p \rightarrow \text{left} == p \rightarrow \text{right}$ 的，也不可能推出 $p \rightarrow \text{left} == p \rightarrow \text{left} \rightarrow \dots$ 和 $p \rightarrow \text{left} == p \rightarrow \text{right} \rightarrow \dots$ 等，因此不能认为该定义包括某些节点有不止一个父节点（即无环有向图）的模型，例如 $p \rightarrow \text{left}$ 等于 $p \rightarrow \text{right}$ 的模型。因为 $p \rightarrow \text{left} == p \rightarrow \text{right}$ 这个等式是不能从这个定义推导出来的。

8.3.2 各种易变数据结构的形状标注

基本形状分成单态和多态两类。图 8.4 文法中的非终结符 *shape_decl* 那一行列出了系统所允许的各种基本形状。单态基本类型分成单态命名基本形状和单态无名基本形状两种。

1. 单态命名基本形状

单态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*) 共 6 种。它们已经在安全 C 语言使用手册中已经介绍过，包括前 5 种形状中的内建谓词 `list(Node *p)` 等。对于这些内建谓词的应用，在验证过程中，验证系统会根据内建谓词应用的实变元的类型，调整安全 C 语言手册中相应内建谓词的定义式，主要是类型和指针域。

在这里需要补充的是双向链表和循环双向链表的 *forward* 标注。双向链表和循环双向链表的形状声明中，应该指明哪个指针域作为 *forward* 指针域。*forward* 指针域主要用在两处。

- 在循环或递归代码中按 *forward* 指针域的前进方向遍历节点。
- 在用量化断言或归纳谓词来逐点描述相邻节点的数据关系（例如递增链表）时，*forward* 指针域出现在这些节点的访问路径上。

一般来说，这两种情况下用的 *forward* 指针域确实是同一个。下面再说如何给出这样的标注。

• 对于操作双向链表或/和循环双向链表的每个函数，需要通过形状标注指明本函数是按哪个指针域来遍历易变数据结构的节点的。该指针域也是标注中访问节点数据所使用的访问路径上允许出现的指针域。

对于例 7.9 双向链表倒置的函数 *invert*，从该函数的代码可知，指针域 *r* 的前进方向就是 *forward* 方向。所以 *forward* 标注应该如下（见图 8.4 的文法）：

```
ListNode* invert(ListNode* const list) /*@ shape ListNode forward : r; */{ ... }
```

在该例的有序链表和表段的定义中，相邻节点的数据大小比较，也是以 *r* 为前进方向的。

若仅是函数原型，这样的形状标注仍然出现在参数表之后。

• 使用这种节点数据类型的不同函数，可以确定各自的 *forward* 指针域。若需要在一个函数中操作使用这种节点类型构造两个链表（例如两个双向链表），并且它们的 *forward* 指针域不同，则需要用 *typedef* 方式为类型 *ListNode* 定义另一个名字，例如 *Node1*，让其中一个链表的节点使用 *Node1* 类型。这是标注 `/*@ shape ListNode forward : r; */` 中需要出现节点类型名的原因。

• 在 *r* 为 *forward* 域的情况下，在程序员的标注中， $p \rightarrow r \rightarrow \text{data} == 10$ 和 $p \rightarrow \text{data} < p \rightarrow r \rightarrow \text{data}$ 都是允许的， $p \rightarrow r \rightarrow l \rightarrow r \rightarrow \text{data} == 10$ 和 $p \rightarrow \text{data} < p \rightarrow r \rightarrow l \rightarrow r \rightarrow \text{data}$ 都是不允许的。虽然两者与前两者在语义上是一样的。

在仅是基本形状而没有基本形状嵌套的情况下，只需要给出基本形状的描述，无需给出有关形状嵌套特征（文法中的 *feature*）的主次（文法中的 *primary* 和 *secondary*）描述。也无需给出域指针关系特征的描述，因为单态命名基本形状的这些关系是安全 C 语言预定义的，已列在安全 C 语言使用手册 3.1.2 节。

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。

在验证带节点数据断言的操作单态命名基本类型的函数时,定义节点数据断言的谓词通常蕴涵这里的内建谓词。例如,二叉排序树的定义是

```
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 $BST(p) \implies \text{tree}(p)$ 。因此在书写循环不变式和程序点断言时,有了断言 $BST(p)$,就不必再写上 $\text{tree}(p)$ 。有了断言 $BST_seg_r(p, q)$ 时,就不必再写上 $\text{tree_seg_r}(p, q)$ 了。

2. 单态无名基本形状

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谓词或归纳谓词提供在标注中。对这样的形状没有公认的形状名称。

在安全 C 语言使用手册的第 3 章介绍了线索二叉树的例子。现在再介绍一例,每个节点有 3 个指针的单态无名形状的类型定义及随后的形状标注如下。

```
typedef struct node{int data; struct node* first, second, third;} Node;  
/*@ shape first, second, third: no_name, invariant;  
  inductive ternary_tree(Node *p) = p == \null ||  
    ternary_tree(p->first) && ternary_tree(p->second) && ternary_tree(p->third);  
  type invariant tTree(Node *p) = ternary_tree(Node *p); */
```

这种形状标注中虽然没有主次特征描述,但特征描述的 `invariant` 表示用类型不变式来描述该形状是三叉树。

三叉树的类型定义及其形状标注也可以如下。

```
typedef struct node{int data; struct node* ptr[3];} Node;  
/*@ shape ptr: no_name, invariant;  
  inductive ternary_tree(Node *p) = ... ..  
  type invariant tTree(Node *p) = ternary_tree(Node *p); */
```

`shape` 之后的数组名 `ptr` 表示该数组的所有元素都是无名基本形状的指针。

在定义单态无名形状的语法限制之前,先区分形状上的有向边。从根节点向下看,

- 指向子节点的边为正向边。
- 指向祖先节点的边为逆向边。
- 互不以对方为祖先的两个节点之间的边为交叉边,

单态无名基本形状是一种多叉的树形结构,最下面节点的指针域若不等于 `NULL`,则一定在形状图上形成逆向边(不允许形成交叉边)。即若 `p` 指向节点的逆向指针 `p->left` 等于 `q`,则存在一条从 `q` 指向节点出发并沿着正向边的路径可以到达 `p` 指向的节点。

线索二叉树和上述三叉树都符合这个定义,双向链表也符合这个定义,但循环双向链表不符合这个定义。这个定义禁止了把无环有向图作为一种形状。实际代码中,无环有向图通常用邻接矩阵表示。

由弱类型不变式的语义知道,在函数的入口和出口,该类型的全局变量、形参和返回值都满足该不变式。安全 C 语言使用手册的例 3.7 和例 3.8 以及后面的例 8.5 也都是用弱类型不变式表达形状特征。

若把形状描述中的 `invariant` 改成 `predicate`,则没有规定在函数入口和出口一定有这种不变性,相关的例子可见安全 C 语言使用手册例 3.6 和本手册的例 8.7。

用于构造单态基本形状的节点的数据类型,除了作为静态区的指针以外,其指向本类型的指针数必须和相应形状的规定一致,并且不得有多余指针,即所有堆指针都必须有形状标

注。这句话对下面引入的各种形状都适用。

3. 多态基本形状

多态基本形状用于称呼那些各节点的类型相同,所含指针的个数可以不同但都指向节点自身类型的数据结构。节点的多态性依靠节点类型中的共用体成员或灵活数组域来体现。多态基本形状也分成两个子类:多态命名基本形状和多态无名基本形状。

多态命名基本形状只有数据块 1 种,它与单态的类似,区别是同一结构体类型的数据块可以有不同的数据域,因此它的形状描述用 `polymorphic_data_block`。

多态无名基本形状的各节点所含指针个数并非都相同。例如,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。这类多态基本形状没有公认的形状名称,抽象语法树也只是特定领域内的称呼。多态无名基本形状的形状描述用 `polymorphic_no_name`。安全 C 语言使用手册中的例 3.7 是抽象语法树的描述。

和单态无名基本形状一样,多态无名基本形状的标注中同样也没有主次特征描述。指针之间关系特征的描述方式与单态无名基本形状的一样,即 `predicate` 或 `invariant`。

单态和多态的基本形状合称基本形状。多态无名基本形状的节点类型还有强类型不变性,即下面的第(2)点和第(3)点规定。

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,并且它们含指向外层结构体类型的指针,否则就不是多态。若是用共用体,则共用体的各个成员代表多态节点的不同形态;若是用灵活数组域,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。安全 C 语言使用手册中的例 3.7 是一个使用共用体的例子,其中 `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。安全 C 语言使用手册中的例 3.8 是一个使用灵活数组的例子。

多态无名基本类型是子节点个数有多种可能的多叉树结构,并且最下指针只允许等于 `NULL`,不允许出现逆向边(逆向指针)的情况。

4. 嵌套形状

从这里的嵌套形状一直到第 7 点相同形状实例的序列都是描述怎样从基本形状构造复杂形状。

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。

用于构造嵌套形状中主形状的指针的特征(图 8.4 文法中的非终结符 `feature`)是 `primary`,而指向嵌套在其中的次形状的指针的特征是 `secondary`。若次形状节点的类型中也有主形状和次形状指针之分,则形成 3 层的嵌套形状。按此方式可以构造多层嵌套的形状。

若指向嵌套在其中的次形状的指针的特征是 `to_shared`,则主形状必须是单态命名基本形状。`to_shared` 表示该形状的各节点共享一个数据块,即它们都通过这个 `to_shared` 指针,直接或通过其下若干层数据块的 `secondary` 指针,指向同一个数据块,这个数据块的特征描述是 `shared_data`,表示被共享。

用于构造易变数据结构类型中的各指针都必须有形状标注。其中主形状指针数必须符合相应形状的规定。次形状指针数可以有多个,但主形状的节点对每个次形状正好有一个指针。次形状指针的类型必须区别于主形状指针的类型。

数据块也可以有内嵌形状实例的指针，这时可以称它为过渡块。数据块所含内嵌形状指针的指向类型必须有别于数据块所属类型。数据块的 `secondary` 指针有可能直接指向 `shared_data`，但是它的特征仍用 `secondary`，而不用 `to_shared`。因为数据块这种形状只有一个节点，没有必要出现本形状的所有节点指向同一个数据块的概念；另一方面，它作为共享现象的过渡块是可能的，所以有可能出现由它直接指向共享数据块。

嵌套形状声明的例子可见安全 C 语言使用手册的例 3.9 和例 3.10。

5. 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命名和无名基本形状，但不包括数据块，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。

(1) 确定的附加指针

若附加指针与基本指针（构造基本形状的指针）之间的相等关系可以用简单指针断言来描述，则称其为确定的附加指针。例如，含父节点指针的二叉树（见安全 C 语言使用手册的例 3.11）和含两个逆向指针的左孩子右兄弟树等。

对于确定的附加指针来说，相关谓词和类型不变式应该描述附加指针与基本指针之间的关系，或者虽是描述附加指针 `a1` 和 `a2` 之间关系的谓词，但可推导出它们与基本指针之间的关系。不应仅出现描述附加指针 `a1` 和 `a2` 之间关系的谓词，但无法推断出它们与基本指针之间的关系。

例 8.5 对于含两个逆向指针的左孩子右兄弟树，其基本形状是二叉树，附加指针用于联系基本形状上的节点，其指向可明确描述。若结构体变量中构造骨架的两个域指针分别是 `child`（左子树指针，其含义是第一个孩子节点指针）和 `sib`（右子树指针，其含义是右兄弟节点指针），两个附加指针分别是 `parent`（父节点指针）和 `sib_prev`（左兄弟节点），两个附加指针都指向基本形状的节点，因此它们的形状描述也是 `tree`，它们的特征由弱类型不变式 `mcfTree` 定义，`mcfTree` 的定义要用到归纳谓词 `mcf_tree`，见图 8.5 □

```
typedef struct node{struct node *child, *sib, *parent, *sib_prev;} Node;
/*@ shape child, sib : tree, primary;
    shape parent, sib_prev: tree, invariant;
    inductive mcf_tree(s) =
        (s->sib!= \null ==> s->sib->sib_prev== s && s->sib->parent == s->parent && mcf_tree(s->sib)) &&
        (s->child != \null ==> s->child->parent==s && s->child->sib_prev == \null && mcf_tree(s->child));
    type invariant mcfTree(Node* s) = s == \null || s->parent == \null && mcf_tree(s); */
```

图 8.5 含两个逆向指针的左孩子右兄弟树的节点的类型定义和标注

例 8.6 图 8.6 是 B-树的类型定义和标注（B-树的定义参考文献[6]）。在第一行标注中，`ptr: no_name` 之后是 `primary`，表示 `ptr` 数组的数组元素都是基本指针。`parent` 是附加指针。`ptr` 虽然没有 `invariant` 或 `predicate` 的提示，由于它是无名形状，隐含着它也还是要用不变式或谓词来描述它的特性的。 □

(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来描述，则称其为部分确定的附加指针。例如两层跳表，其节点的类型定义和标注见安全 C 语言使用手册例 3.12。在这种情况下的附加指针的形状描述和确定附加指针的情况没有什么区别，只是描述指针性质的等式所受限制少一些。

和确定附加指针的场合一样，相关谓词和类型不变式应该描述附加指针与基本指针之间的关系，或者虽是描述附加指针 `a1` 和 `a2` 之间关系的谓词，但可推导出它们与基本指针之间的关系。

(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。

例如, GNU C Library 中采用请求队列实现异步 I/O 操作,就绪队列由请求队列上的若干满足一定条件的成员组成(构成一个附加单向链表),就绪队列的成员在请求队列中的位置不是能静态确定的,因而难以用指针断言来描述。见安全 C 语言使用手册例 3.13。

```
#define M 5                // B-树的阶
typedef struct BTreeNode {
    int keynum;            // 节点中关键字个数,即节点的大小
    struct BTreeNode* parent; // 指向双亲节点
    int key[M+1];          // 关键字向量, key[0]单元未用
    struct BTreeNode* ptr[M+1]; // 子树指针向量
    //@ ghost int hight;    // 树的高度
}BTreeNode, *BTree;      // B-树节点和 B-树的类型
/*@ shape ptr : no_name, primary;
    shape parent : no_name, invariant;
    inductive Lt(integer key, Btree t) = t == \null || k < t->key[1] && Lt(k, t->ptr[0]);
    inductive Gt(integer key, Btree t) = t == \null || k > t->key[t->keynum] && Gt(k, t->ptr[t->keynum]);
    inductive B_Tree(Btree t) = t == \null ||
        (t->parent == \null ==> 2 <= t->keynum <= M-1) &&
            (t->parent != \null ==> (M+1)/2 <= t->keynum <= M-1) &&
            (t->hight == 1 ==> (\forallall integer i:[0..keynum]. t->ptr[i] == \null)) &&
            (t->hight > 1 ==> (\forallall integer i:[0..keynum]. t->ptr[i] != \null &&
                t->ptr[i]->hight == t->hight-1 && t->ptr[i]->parent == t)) &&
            (\forallall integer i:[keynum+1..M]. t->ptr[i] == \null) &&
            (\forallall integer i:[1..keynum-1].t->key[i] < t->key[i+1]) &&
            (\forallall integer i:[1..keynum].Gt(t->key[i], t->ptr[i-1]) && Lt(t->key[i], t->ptr[i]));
    type invariant B_Trees(BTree t) = t == \null || t->parent == \null && B_tree(BTree t); */
```

图 8.6 B-树的类型和标注

与上两种情况的区别是,节点的不确定附加指针的描述是 list 和 additional, list 表示用附加指针构成的是单向链表,特征 additional 的含义是内部定义的,不需要用类型不变式或谓词来定义附加单向链表指针的含义。

不确定的附加指针还有一种属于自己特有的情况:基本形状的指针和附加单向链表的指针虽然类型相同,但它们不能混淆使用。因此附加单向链表的指针变量(包括作为形参和返回值的情况)都需要额外的标注,可见安全 C 语言使用手册例 3.13。

6. 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简单指针断言或量化指针断言来定义。不能使用不确定的外来附加指针。见安全 C 语言使用手册例 3.14 的队列,其队列的类型定义和标注见图 8.9。

在这种情况下,每个外来的附加指针 q 指向基本形状上某个特殊节点,使得 q 与基本形状的指针 p 之间的关系可以用含量词的指针等式来描述。

```

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

```

图 8.9 队列的类型定义和标注

7. 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。

这种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，标准形状概念见安全 C 语言使用手册 3.3.1 节。

8. 单个节点的使用

形状系统的形状检查（见安全 C 语言使用手册 3.3.2 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`（一种特殊的形状标注）来表示形参的这个性质。

谓词 `\singleton(p)` 表示，不管 `p` 是什么易变数据结构的指针，`p` 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。例如，若 `p` 指向节点有 `left`、`right` 和 `second` 三个域指针，则 `\singleton(p)` 表示 `p->left`、`p->right` 和 `p->second` 都是悬空指针，`\singleton(p) && \list(p->second)` 表示仅 `p->left` 和 `p->right` 是悬空指针，而 `p->second` 的性质由谓词 `\list(p->second)` 表达。系统提供的内建谓词 `\list`、`\c_list`、`\tree`、`\dlist`、`\c_dlist` 和 `\data_block` 等也可用于单个节点的域指针。

`\singleton(p)` 断言可以出现在函数的前条件和后条件中，用来表示参数和返回值的特性，也可以用来表示全局指针在函数入口和出口点的特性，还可以在函数中表示刚通过动态存储分配得到的节点。例如，安全 C 语言使用手册的例 3.15 单向链表插入函数的函数原型如下：

```

Node* listInsert(Node *head, Node *const p);

```

其中 `head` 是单向链表的指针，而 `p` 是待插入节点的指针，`p` 的 `next` 指针域可能并没有赋值，等待插入后赋值。也就是 `p` 指向的单个节点并不能看成单向链表。这时需要在函数前条件中为 `p` 加上断言 `\singleton(p)`，其含义是，`p` 并不像通常的那样指向单向链表，而是指向一个独立的节点。

8.3.3 对函数协议和循环不变式等的指针断言和节点数据断言的限制

在函数前后条件、循环不变式和程序点断言中，需要用断言把形状特征描述清楚，就是把相应形状图上的堆指针的有效性和其中有效指针之间的相等性描述清楚。对描述所使用的断言，在 8.3.1 节的基础上，还有进一步的限制。

本节主要以单向链表为例来进行解释，见图 8.10。若指针 head 指向长度大于 1 的链表的第一个节点，还有指针 p 和 q 依次指向该链表中某处的相邻节点，从 head 指向节点到 p 指向节点一共有 m ($m > 1$) 个节点，q 指向的节点到表尾一共有 n ($n \geq 0$) 个节点。该表一共有 $m + n$ 个节点。

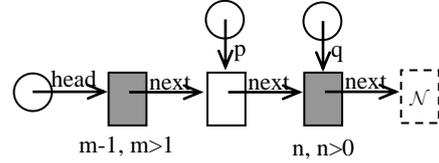


图 8.10 单向链表的形状图的一个例子

1. 形状的描述方式

有分段描述和整体描述两种方式。

(1) 分段描述 对于图 8.10 由 head 指向节点开始的单向链表，指针 p 和 q 把整个链表分成 3 段。分段描述的断言是：

$$\begin{aligned} &\backslash\text{list_seg}(\text{head}, p) \ \&\& \ p == \text{head} \rightarrow (\text{next}:\text{m}-1) \ \&\& \ q == p \rightarrow \text{next} \ \&\& \\ &\backslash\text{list}(q) \ \&\& \ \backslash\text{length}(q, \text{next}) == n \end{aligned}$$

其中 $\backslash\text{list_seg}(p, q)$ 和 $\backslash\text{list}(q)$ 分别描述了前后两部分的形状特征， $p == \text{head} \rightarrow (\text{next}:\text{m}-1)$ 和 $\backslash\text{length}(q, \text{next}) == n$ 分别描述前后两部分的长度。 $q == p \rightarrow \text{next}$ 同时表达了中间部分的形状特征和长度，它也可以看成长度为 2 的表段。注意，与 $q == p \rightarrow \text{next}$ 不一样，断言 $q == p \rightarrow (\text{next}:i)$ ($i > 1$) 只含长度信息，因为当 $i > 1$ 时，该断言不保证没有其它外来指针指向该段的某个中间节点上。

简单说，分段描述就是从左到右，逐段描述两个相邻外来指针所指向部分的性质（若整个形状图的最右节点是 NULL 节点，可能没有外来指针指向它）。

(2) 整体描述 对于图 8.10 的单向链表，整体描述的断言是：

$$\begin{aligned} &\backslash\text{list}(\text{head}) \ \&\& \ \backslash\text{length}(\text{head}, \text{next}) == \text{m}+\text{n} \ \&\& \\ &p == \text{head} \rightarrow (\text{next}:\text{m}-1) \ \&\& \ q == p \rightarrow (\text{next}:1) \end{aligned}$$

其中由 $\backslash\text{list}(\text{head})$ 和 $\backslash\text{length}(\text{head}, \text{next}) == \text{m}+\text{n}$ 给出了该单向链表的宏观描述。然后用 $p == \text{head} \rightarrow (\text{next}:\text{m}-1)$ 和 $q == p \rightarrow (\text{next}:1)$ 分别描述前面两段的长度，最后一段的长度可以推导出来，因而其描述可以省略。长度描述也可以不用分段方式，换成都以 head 为基准来描述 p 和 q 到 head 的距离，即

$$\begin{aligned} &\backslash\text{list}(\text{head}) \ \&\& \ \backslash\text{length}(\text{head}, \text{next}) == \text{m}+\text{n} \ \&\& \\ &p == \text{head} \rightarrow (\text{next}:\text{m}-1) \ \text{和} \ q == \text{head} \rightarrow (\text{next}:\text{m}) \end{aligned}$$

对于分段描述和整体描述这两种方式，要注意下面两点。

- 若合取的所有断言不引用该链表任何节点的其他域（非构造链表的指针域），则该链表可以分段描述也可以整体描述。否则必须分段描述，原因见下面描述形状所使用的各类断言的第 (6) 点。

- 在整体描述一个形状图时，对于描述外来指针之间距离的分段和以 head 为基准这两种方式，只能选用一种，不要混用。另外，在分段描述一个形状图时，不允许使用以 head 为基准的方式。

2. 描述形状所使用的各类断言

(1) 指针关系断言

8.3.1 节已经给出了对一般的堆指针访问路径和指针关系断言的约束，这里有进一步的限制。

- 含折迭域的指针断言 只能是等式断言, 并且含折迭域的访问路径只能出现在等号的一侧, 例如 $p == \text{head} \rightarrow (\text{next}:m-1)$ 和 $q \rightarrow \text{next} == \text{head} \rightarrow (\text{next}:m-1)$ 分别表示从 head 指向节点到 p 指向节点和从 $q \rightarrow \text{next}$ 指向的节点都经过 $m-1$ 个 next 指针域。不要认为后者可以简化成 $q == \text{head} \rightarrow (\text{next}:m-2)$, 因为 $q \rightarrow \text{next}$ 可以是经过一个不在 $\text{head} \rightarrow (\text{next}:m-1)$ 这条路径上的一个节点再指向 $\text{head} \rightarrow (\text{next}:m-1)$ 的。

(2) 内建谓词

内建谓词大部分定义在安全 C 语言使用手册的第 3 章, 例如 $\backslash\text{list}(p)$ 、 $\backslash\text{tree}(p)$ 等。本小节所用例子已经使用了内建谓词。

(3) 自定义谓词

对于包含了数据断言的有序链表 $\text{sorted}(p)$ (8.3.4 节) 和二叉排序树 $\text{BST}(p)$ (7.2 节), 它们的定义分别包括了单向链表和二叉树的指针链接关系。因此当它们作为断言出现时, 不必再重复给出断言 $\backslash\text{list}(p)$ 和 $\backslash\text{tree}(p)$ 。

当描述节点数据性质的自定义谓词或其他方式的断言, 未能蕴涵相应形状的指针连接关系时, 这时仍要用内建谓词和指针关系断言来描述形状。例如, 安全 C 语言使用手册的图 2.5 中描述单向链表有序性的断言

$$(\text{forall integer } i:[1..m-1]. \text{head} \rightarrow (\text{next}:i-1) \rightarrow \text{data} \leq \text{head} \rightarrow (\text{next}:i) \rightarrow \text{data})$$

其中出现的最长指针型访问路径是 $\text{head} \rightarrow (\text{next}:m-1)$ (约束变元 i 取上界 $m-1$)。这里是用量化断言描述节点的数据特性, 它也能体现节点连接的部分形状特性。但是这并不意味着 head 是指向 m 个节点链接的表段, 一个问题是, 这个量化断言能表示出 head 指向链表的第 1 个节点, 但是哪个指针指向表段最后一个节点是不清楚的。另外一个问题是, 可能有外来指针指在某个内部节点上。因此, 描述节点数据特性的量化断言不被认为同时描述了节点链接的形状特征。

不管是用内建谓词还是程序员定义的谓词, 谓词应用的堆指针变元一般最多只含构成该形状的结构体指针域一次的访问路径, 必要时也可以出现多次的访问路径, 但不允许使用折迭域。例如 $\backslash\text{tree}(p)$ 、 $\backslash\text{tree}(p \rightarrow \text{left})$ 、 $\backslash\text{c_list_seg}(p, q)$ 、 $\backslash\text{c_list_seg}(p \rightarrow \text{next}, q)$ 、 $\backslash\text{c_list_seg}(p, q \rightarrow \text{next})$ 和 $\backslash\text{c_list_seg}(p \rightarrow \text{next}, q \rightarrow \text{next})$ 等。

(4) 描述形状的其他断言

还有一些描述形状的简单断言, 例如 $p == \backslash\text{null}$ 、 $p != \backslash\text{null}$ 、 $\backslash\text{dangling}(p)$ 和 $\backslash\text{singleton}(p)$ 等。内建谓词 $\backslash\text{dangling}$ 和 $\backslash\text{singleton}$ 应用的变元 p 可以是含构成该形状的结构体指针域名的访问路径。

- 若是指向并不构成形状的单个节点的指针, 则用断言 $\backslash\text{singleton}(p)$ 表示。也许还有进一步描述 p 指向节点的域指针的描述。 $\backslash\text{singleton}(p)$ 的变元 p 不允许有折迭域。

- 若指针 q 是悬空指针, 则用断言 $\backslash\text{dangling}(q)$ 表示, 同样, 也不能有 q 等于某个指针的等式断言。

也有在描述形状时不得使用的断言形式。

- 不得使用含逻辑函数应用的指针断言。
- 不得使用量化的指针关系断言。

(5) 形状长度描述断言

- 对于整体描述的形状, 用一个长度断言给出链表的表长, 例如从

$$\backslash\text{list}(\text{head}) \ \&\& \ p == \text{head} \rightarrow (\text{next}:m-1) \ \&\& \ q == p \rightarrow (\text{next}:1) \ \&\& \\ \backslash\text{length}(\text{head}, \text{next}) == m+n$$

可知表长为 $m+n$ 。

- 对于分段描述的形状, 用各分段长度之和作为整个形状的长度。例如, 从

$$\backslash\text{list_seg}(\text{head}, p) \ \&\& \ q == p \rightarrow \text{next} \ \&\& \ \backslash\text{list}(q) \ \&\&$$

`p == head->(next:m-1) && \length(q, next) == n`

可知表长为 $m+n$ 。也就是这个分段描述默认了表长为 $m+n$ 。

- 在分段描述方式中，各分段的描述和各分段的长度描述必须一致，即两者对形状的分段是一致的。例如 `\list_seg(p, q)` 的描述和长度描述 `q == p->(next:n)` 是一致的。

(6) 关于节点的数据断言

因为节点数据断言不描述形状，那么节点数据断言的描述与形状的描述必须一致，即两者对形状的分段也是一致的。它们或者像图 8.11 那样，合并既描述节点之间指针关系又描述节点间数据关系的谓词中，或者像安全 C 语言使用手册的图 2.5 那样，链表被外来指针分成几段，节点间数据关系的描述相应地也分成这么几段。

这么要求的目的是保证没有外来指针指向谓词应用和逻辑函数应用所涉及（影响到，touch）的易变数据结构的任何节点上，除非它和该应用的某个指针参数相等或者等于该应用第一个指针变元所指向的第 n 个节点（ n 是大于 1 的常数，例如 n 等于 5，外来指针 q 的断言是 `q == p->(next:4)`）。并且若外来指针等于表段或树段的末尾指针，则随后的代码不修改末尾指针所指向节点的域（构成形状的指针域除外），也不顺着表段或树段逆向访问它们的节点。同样，外来指针也不会指向量化断言所涉及的易变数据结构的任何节点上，除非它是量化断言所涉及节点中的起始节点或末尾节点。

下面考虑函数前后条件、循环不变式和程序点断言各自可能有的特殊情况。首先，对于这几种情况，断言所描述的形状都必须符合形状检查（安全 C 语言使用手册 3.3 节）的要求。

先说函数前条件。对函数前条件的一点额外要求是：若没有修饰符限定作为形参的 p 在函数中不能被修改，则前条件中必须有且仅有一个逻辑变量等于 p 的断言（例如 `oldp == p`），理由见下面的第（2）点。

再说函数后条件。在函数的出口，若指针实参成了悬空指针，则函数后条件中必须有反映这个事实的断言。下面是具体解释。

指针参数和其他类型的参数一样，函数体中对形参的赋值不会改变对应实参的值。不一样的地方是，指针实参的某些性质可能发生改变。假定形参指针是 p ，和其对应的实参指针是 p' 。在函数入口， p 和 p' 指向同一个节点，或都是 NULL 指针（这是由函数调用的语义决定的）。下面是改变 p' 的性质的两个例子。

(1) 在函数体中， p 被赋值后， p 和 p' 可能指向不同的节点。例如让 p 指向新插入的节点，这个新节点是链表的第 1 个节点，则 p' 指向第 2 个节点。 p' 虽未重新赋值，但体现出的性质有了变化。

(2) 在函数体中， p' 所指向对象被 `free(p)` 操作释放， p' 就成为悬空指针，即 p' 的性质发生重大改变。

对于（1），函数后条件必须反映实参指针的这个变化，否则返回调用点后，实参指针信息丢失。对于（2），函数后条件中必须有反映实参指针是悬空指针断言。方式有两种：

- 若形参 p 是 `const` 指针，则 p 可以出现在后断言中，它描述的就是实参指针的性质。例如，函数后条件中应该有断言 `\dangling(p)`。见安全 C 语言使用手册的例 3.15。

- 若形参 p 是指针，但没有修饰符限定 p 在函数中不能被赋值，从前条件的限制知道，前条件中有断言 `oldp == p`，则函数后条件中可用 `oldp` 来描述实参指针。例如，函数后条件中有断言 `\dangling(oldp)`。

最后说循环不变式和程序点断言。它们没有什么特别的限制。

对于循环不变式，对先前提到的节点的数据断言可放宽。若循环体就是依次沿形状上的各节点，把一个指针逐点推进，没有其它动作。例如，

```
while(q->l != NULL){
    q = q->l;
```

}

则循环不变式重点描述迭代过程中 q 在形状中指向节点的位置变化。涉及节点数据的断言在循环迭代过程中可保持进循环时的整体描述不变(只要它们在循环语句之后的程序点符合先前关于对节点数据断言的限制), 即不必在循环不变式中把它们描述成被 p 分隔成两部分。

再精准一点, 若上述逐点遍历形状的循环, 对部分数据断言有影响, 则在循环不变式中该部分断言需要被 p 分隔成两部分描述, 而其余的数据断言没有这个必要。

3. 描述形状中各节点数据所使用的断言

(1) 表示节点数据域的访问路径的前缀指针型访问路径要符合 8.3.1 节的规定。

(2) 在逐点描述节点的数据性质或相邻节点的数据关系时, 不论是单向链表、循环单向链表、双向链表、循环双向链表还是二叉树, 首尾节点都看成是不相邻的。也就是, 在逐点描述节点的数据性质时, 不能在尾节点描述过后, 继续描述首节点。在逐点描述相邻节点的数据关系时, 不能出现描述首尾节点的数据关系断言。

例 8.7 以安全 C 语言使用手册第 2 章的有序单向链表插入函数为例, 把那儿图 2.5 的内容复制到图 8.11, 其中有序链表和有序表段用归纳谓词而不是全程量化断言来描述。

`sorted_list` 和 `sorted_seg` 两个谓词同时定义了所涉及节点的指针域之间和数据域之间的关系, 因此本例不必像安全 C 语言使用手册图 2.5 那样分开描述指针域之间的关系和数据域之间的关系。两者相比, 本例的函数协议和循环不变式就显得简洁一些。

□

```

#include <stdlib.h>

typedef struct node { struct node * next; int data; } Node;
/*@ shape next : list;

/*@ inductive sorted_list(Node * p) =
    p == \null || p->next == \null || p->data <= p->next->data && sorted_list(p->next);
inductive sorted_seg(Node * p, Node * q) =
    p == q && p != \null || p != q && p->data <= p->next->data && sorted_seg(p->next, q);
lemma property1: \forallall Node* p, Node* q.
    sorted_seg(p, q) && q->data <= q->next->data && q->next != \null ==> sorted_seg(p, q->next);
lemma property2: \forallall Node* p, Node* q. sorted_seg(p, q) && sorted_list(q) ==> sorted_list(p); */
/*@ logic Node* oldhead;
/*@ requires sorted_list(head) && \list(head) && \length(head, next) == m && oldhead == head;
assigns *oldhead;
exits \exit_status == 1;
ensures sorted_list(\result) && \length(\result, next) == m + 1 &&
    (m > 0 && oldhead == \result || m == 0 && oldhead == \null ||
    m > 0 && oldhead == \result->(next:1));*/
Node * listInsert(Node * head, const int data, const int m){
    Node *ptr, *ptr1, *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;
    } else {
        ptr1 = head; ptr = head->next;
        //@ ghost j = 1;
        /*@ loop invariant//分成插在表中 and 表尾两情况。依指在链表中的指针，对链表分段描述。
            1 <= j <= m && oldhead == head && sorted_seg(head, ptr1) &&
            \list_seg(head, ptr1) && ptr1 == head->(next:j-1) &&
            \list(p) && \length(p, next) == 1 && ptr1->data < p->data && p->data == data &&
            (ptr != \null && ptr == ptr1->next && ptr1->data <= ptr->data &&
            sorted_list(ptr) && \length(ptr, next) == m - j ||
            ptr == \null && ptr1->next == \null && j == m); */
        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 8.11 有序单向链表的插入函数

8.3.4 形状标注和形状特征断言小结

形状标注出现的地方可以概述如下。

(1) 用于构造易变数据结构节点的类型声明中的各个指针都需要有标注，除非它作为静态区指针使用。

(2) 对于拟指向堆块的声明指针

- 若从它的类型信息可清楚地知道它指向何种易变数据结构，则不用标注。
- 若从它的类型信息不知道它用于指向易变数据结构，则需要对它使用形状标注。
- 若从它的类型信息知道它指向何种易变数据结构，但缺少进一步信息（例如，指向主形状还是附加单向链表），则需要用形状标注指明细节，例如安全 C 语言使用手册例 3.13 的 `additional_list`。

- 不是用于指向堆块的指针，不能标注。

例如下面两个代码段中的指针 `p` 和 `q` 都不用标注。

```
char *p = "abc"; char *q;  
q = "def"; q = p;
```

和

```
long a[100], b[100]; long *p, *q;  
p = a; q = b; *p ++ = *q ++;
```

形状特征有系统预定义的，它们是单态和多态的命名基本形状的指针特性。在 8.3.3 节已经提到，在函数前后条件、循环不变式和程序点断言中需要描述预定义形状在这些程序点的指针特性。对于复杂形状中出现的预定义形状也是这样。

其余的形状，包括单态和多态的无名基本形状、复杂形状中出现的单态和多态的无名基本形状和复杂形状中的附加部分等，都是通过谓词（例如安全 C 语言使用手册 3.6 的线索二叉树）或弱类型不变式（例如例 8.5 的左孩子右兄弟树）来定义它们的指针特性。

用弱类型不变式定义的形状与系统预定义的形状一样，在函数前后条件、循环不变式和程序点断言中需要描述它们在这些程序点的特殊性。

用谓词定义的形状，与用弱类型不变式定义的形状的不同之处是：弱类型不变式在函数前后条件中可以省略。下面举例来说明。

例 8.8 本例是中序遍历访问线索二叉树节点的函数，见图 8.12，线索二叉树的谓词定义 `hTree` 等见安全 C 语言使用手册 例 3.6。由于线索二叉树是用谓词而不是弱类型不变式定义的，因此图 8.12 中 `InOrderTraverse` 函数的前后条件中有断言 `hTree(T)`。线索二叉树不是单态命名基本形状，因此需程序员提供完整的循环不变式。在图 8.12 中，还有一些相关的谓词定义被略去，证明要用到的引理也都被略去。另外，本例使用了第 9 章才介绍的幽灵变量和幽灵语句。 □

```

/*@ ghost bool allVisited;
/*@ requires hTree(T);
    assigns allVisited;
    ensures hTree(T) && allVisited;
*/
void InOrderTraverse(const Node * const T) { // T 指向头节点, 头节点的 l 链指向根节点 p
    @@ ghost Node *visited, *pred1, *pred2, *succ1, *succ2;
    p = T->l;
    @@ visited = T; allVisited = true; //先默认所有节点都能访问
    @@ ghost pred1 = T;
    @@ ghost pred2 = T;
    @@ ghost succ1 = T;
    @@ ghost succ2 = T;
    @@ loop invariant tTreeSeg(T->l, pred1, succ1, p, pred2, succ2) &&
        tTree(p, pred2, succ2) && visited == leftMost(p)->l && allVisited;
    while (p != T) {
        @@ loop invariant tTreeSeg(T->l, pred1, succ1, p, pred2, succ2) &&
            tTree(p, pred2, succ2) && visited == leftMost(p)->l && allVisited;
        while(p->l_tag == Link) {
            @@ ghost succ2 = p;
            p = p->l;
        }
        // 访问节点 p
        /*@ if (p->l_tag == Thread && visited==p->l) { // 上次访问的是它的前驱
            visited = p; // 把 p 记为上次访问的节点
        } else {
            allVisited = false; visited = p; // 出现未按序访问, allVisited 置为 false
        }
        */
        @@ loop invariant tTreeSeg(T->l, pred1, succ1, p, pred2, succ2) &&
            tTree(p, pred2, succ2) && visited == p && allVisited;
        while(p->r_tag == Thread && p->r != T) {
            @@ ghost pred2 = p;
            p = p->r;
            // 访问节点 p
            /*@ if (p->r_tag == Thread && visited->r == p) { // 同上
                visited = p;
            } else {
                allVisited = false; visited = p;
            }
            */
        }
        @@ ghost pred2 = p;
        p = p->r;
    }
    return;
}

```

图 8.12 中序遍历访问线索二叉树节点的函数

第 9 章 幽灵变量和语句

幽灵变量 (*ghost variable*) 可以作为赋值的对象, 这样的赋值不能出现在程序的可执行代码中, 但可以出现在程序标注中对幽灵变量赋值的幽灵语句里。幽灵语句好像被执行而实际并不执行。好像被执行的意思是指, 在程序验证时认为幽灵语句是程序的一部分, 而在程序执行时这样的语句不存在。显然, 幽灵语句不能对任何程序变量赋值, 以保证不会改变基于程序变量能观察到的程序行为。

引入幽灵变量和幽灵语句是为了便于写程序规范。例如, 遍历单向链表的循环以节点的 `next` 指针是否等于 `NULL` 来判断循环是否该终止了, 并不需要有用来记住循环迭代计算次数的程序变量。但是没有这种记住循环迭代计算次数的变量, 有时写不出循环不变式或者有这种记住迭代次数的变量会使得循环不变式容易写或直观易理解。这时可以用幽灵变量及其赋值的幽灵语句来记住循环迭代次数。

9.1 幽灵代码的语法

幽灵变量和幽灵语句类似于 C 的变量和语句, 但是仅在标注中可见。它们由出现在标注开头的关键字 `ghost` 引入, 以 `/*@` 开始并以 `*/` 结束, 或者以 `//@` 开始到本行结束。标注不能嵌套, 单行的标注有完整的含义, 不能改写成多行。幽灵变量和语句的文法见图 9.1, 但略去单行标注的文法没有给出。各产生式右部的终结符 (`ghost` 除外) 和非终结符都是 ISO 标准的 C 语言文法的终结符和非终结符, 其中非终结符的文法没有任何 SCSL 扩展。该文法表明, 幽灵变量和语句可以出现在 C 相应部分可出现的位置, 只是须出现在标注中。

<i>C-declaration</i>	→ <code>/*@ ghost declaration */</code>	// 幽灵声明
<i>C-struct-declaration</i>	→ <code>/*@ ghost struct-declaration */</code>	// 幽灵域声明
<i>C-direct-declarator</i>	→ <code>direct-declarator (parameter-type-list²)</code> <code>/*@ ghost (parameter-type-list) */</code>	// 函数的幽灵变元
<i>C-postfix-expression</i>	→ <code>postfix-expression (argument-expression-list²)</code> <code>/*@ ghost (argument-expression-list) */</code>	// 带幽灵变元的调用
<i>C-statement</i>	→ <code>/*@ ghost statement⁺ */</code>	// 幽灵语句
<i>statement</i>	→ <code>term [term .. term] = term [term .. term]</code>	// 增加幽灵数组区间赋值语句
<i>C-selection-statement</i>	→ <code>if (expression) statement /*@ ghost else statement */</code>	// 带幽灵 else 子句的条件语句
<i>C-labeled-statement</i>	→ <code>/*@ ghost case constant-expression : statement */</code> <code>/*@ ghost default : statement */</code>	// switch 语句的幽灵分支

图 9.1 幽灵变量和语句的文法

没有指定大小的幽灵数组声明 `ghost int a[]`, 包括 `a` 数组上界是无穷大 (即 `\infinite`) 的情况, 也就是不必单独声明某个幽灵数组是无穷大幽灵数组。另外, 对上界是无穷大的幽灵数组的如下区间赋值

$$a[0..\infinite] = b[0..\infinite]$$

没有意义, 因为 `b` 只能也是幽灵数组。若认可这样的区间赋值, `b[0..\infinite]` 的元素的值又是从哪来的呢? 没有意义的根源是不能为程序数组写出区间 `[0..\infinite]`。因此 `\infinite` 仅使用在 `assigns` 子句中。对于有可能看成无穷大的幽灵数组, 数组元素的下标表达式宜声明为 `integer` 类型。这样, 无穷大数组不会有越界问题, 下标变量也不会有上溢问题。

和 C 文法相比, 有如下几点变化。

1. 出现在幽灵代码中的注释也必须以 `//` 开头并到其所在行的结尾结束, 与第 2 章介绍词法时所说的一致。因为幽灵代码本身出现在 C 注释 `/*@ ... */` 中或 `//@` 之后到所在行的结束,

不是这样可能导致不正确的 C 代码。

2. 逻辑类型，例如 `integer` 和 `real`，也可以使用在幽灵代码中。

3. 关于幽灵函数、变量和域名的作用域，可以这么解释：把它们当成程序的一部分，按安全 C 语言的作用域规则来确定它们的作用域。

4. 非幽灵函数可以有幽灵形参。如果幽灵形参表子句出现在非幽灵函数声明的形参表之后，那么幽灵形参表必须非空，并且形参个数固定。该函数的调用者必须提供同样个数的幽灵实参。

5. 任何 C 代码的 `if` 语句，若它没有非幽灵的 `else` 子句，则可以用幽灵 `else` 子句来扩大。幽灵 `else` 子句只能含对幽灵变量的赋值语句。类似地，非幽灵 `switch` 语句可以用幽灵 `default` 开启的子句来扩大，如果原来不存在非幽灵 `default` 子句的话。非幽灵 `switch` 语句可以用幽灵 `case` 开启的子句来扩大。

6. 幽灵数组在某些方面与程序数组不同而与逻辑数组类似。首先，幽灵数组的声明有两种方式，指定大小和未指定大小的数组，后者可接受元素类型相同的任意大小的数组对它的赋值。幽灵数组的名字不被看成幽灵数组首元素的指针。

幽灵代码中允许对幽灵数组进行区间赋值，对幽灵数组区间赋值的语法见图 9.1。一个幽灵数组可以有若干个数组区间赋值语句，在这些语句中，幽灵数组的区间不能出现重叠。

幽灵数组，例如例 9.5 的 `ghost array tempa`，主要用来记住一个程序数组 `a` 的当前值，以便当前的 `a` 仍可以作为一个整体（用 `tempa`）出现在后面的断言中，见例 9.5 和 9.6。

9.2 对幽灵代码的限制

对于幽灵代码来说，语义上要求幽灵语句不改变程序的执行，因此，需要如下一些限制。

1. 幽灵代码不能修改任何非幽灵变量。

2. 幽灵代码不能修改任何结构体的非幽灵域。

3. 对于一个幽灵数组，区间赋值把一个数组分成一个或多个不相交区间。区间赋值只对幽灵数组的这些区间赋值。

4. 如果幽灵指针 `p` 指向非幽灵的内存单元，则 `p` 必须伴有 `const` 修饰符，用以表明不能通过 `p` 来修改非幽灵的内存单元。不允许出现指向堆易变数据结构的幽灵指针。

5. 幽灵函数的函数体是幽灵代码，不允许它修改任何非幽灵的变量和域。

6. 如果非幽灵的 C 函数被幽灵代码调用，则函数不能修改任何非幽灵的变量和域，还不能出现动态存储分配和释放。

7. 函数的控制流图不因幽灵语句而改变。具体说，不允许幽灵的 `return`、`abort`、`exit` 或 `_exit` 出现在非幽灵的 C 函数的函数体中，幽灵的 `goto`、`break` 和 `continue` 只能跳到幽灵代码，并且不能跳出包围该幽灵语句的最内非幽灵程序块。

在上述限制基础上，幽灵代码的语义是这样描述的。带幽灵代码的程序在执行时，包括了幽灵内存栈，它有别于通常的栈。幽灵变量和幽灵域都存储于幽灵栈，因此每个有内存副作用的操作都可以分成幽灵和非幽灵的，幽灵代码的内存副作用只会出现在幽灵栈中。

例 9.1 图 9.2 给出一些非法的幽灵指针赋值语句。 □

例 9.2 图 9.3 给出一些非法的幽灵控制流语句。 □

```
void f(int x, int *q) {
    //@ ghost int *p = q;
    //@ ghost *p = 0;    // The assignment is wrong: it modifies *q which lies in regular memory heap.
    //@ ghost p = &x;
    //@ ghost *p = 0;    //The assignment is wrong: it modifies x which lies in regular memory stack.
}
```

图 9.2 非法的幽灵指针赋值语句

```

int f(int x, int y) {
    //@ ghost int z = x + y;
    switch(x) {
        case 0: return y;
        //@ ghost case 1: z = y; // The statement is correct.
        //@ ghost case 2: { z++; break;} // Invalid, would bypass the non-ghost default.
        default: y++;
    }
    return y;
}

int g(int x) {
    //@ ghost int z = x;
    if (x > 0) {return x;}
    //@ ghost else { z++; return x;} // Invalid, would bypass the non-ghost return.
    return x + 1;
}

```

图 9.3 非法的幽灵控制流语句

例 9.3 SCSL 不直接支持对程序输入输出数据的描述，但是利用幽灵变量和幽灵语句，对于一些简单的情况，可以找到语义等价的替代办法，见图 9.4。 □

```

#include <stdio.h> // 为简洁起见，本例没有给出检查输入和输出数据是否溢出的标注。
//@ ghost long input, output;
/*@ assigns input, output;
    ensures \result == 0 && output == 2 * input || \result == 1 && output == 0; */
int main() {
    long x;
    //@ ghost output = 0;
    if( scanf("%ld", &x) != 1 ) return 1; // scanf 函数需要检查返回值，读取失败时做出错处理
    //@ ghost input = x;
    printf("%ld\n", 2*x);
    //@ ghost output = 2*x;
    return 0;
}

```

图 9.4 利用幽灵变量的一个简单例子

例 9.4 图 9.5 是使用静态局部变量函数的例子。因为静态局部变量不能出现在函数协议中，这时可通过幽灵变量和幽灵语句的帮助来表示这个函数的协议。在图 9.5 的代码中，用幽灵的外部变量 `maxForbidden` 来跟踪静态局部变量 `x` 的值，并让该幽灵变量出现在函数协议中。该函数类似一个计数器，每次被调用时 `x` 的值加 1，产生一个新值直至到达整型的最大值 (`INT_MAX`)。幽灵变量 `maxForbidden` 记录已经由该函数产生的最大整数。因涉及先前各次调用的结果，很难仅用逻辑变量来表达，所以使用幽灵变量是一个很好的选择。 □

```

#include <limits.h>
/*@ ghost int maxForbidden = 0;  //@ global invariant M: 0 <= maxForbidden <= INT_MAX; //变量不变式
/*@ assigns maxForbidden;
    ensures \result == maxForbidden && 0 <= maxForbidden <= INT_MAX; */
int gen() {
    static int x = 0;          //@ global invariant I: 0 <= x <= INT_MAX; //变量不变式
    if (x != INT_MAX) x++;    //防止数值计算溢出
    //@ ghost maxForbidden = x;
    return x;
}

```

图 9.5 利用幽灵变量的另一个简单例子

例 9.3 和 9.4 都是在特殊情况下，用幽灵变量来写函数协议。一般情况下，都应该是程序变量出现在函数协议中，因为函数协议就是表达该函数所操作程序变量和其它程序数据的性质变化情况。下面例 9.5 和 9.6 是一种幽灵变量经常使用的场合。

例 9.5 本例完成例 3.9 的冒泡排序函数，例 3.9 仅证明冒泡排序得到的结果数组是有序的，本例还证明了另一个同样重要的性质：排序后的数组是排序前数组的一种置换，即两者的数据集一样，见图 9.6。标注中给出数组置换的归纳定义。

在这个例中，逻辑数组 `olda` 用来记住排序前的数组，幽灵数组 `tempa` 用来记住每次交换两个相邻元素后的数组，它是交换前数组的一种置换，根据数组置换的归纳定义，它也是排序前数组的一种置换。 □

例 9.6 本例是一个较大的例子，是著名的 Dijkstra 荷兰国旗算法，见图 9.7。将乱序的红白蓝三色小球排列成有序的红白蓝三色的且同颜色在一起的小球组。这个问题之所以叫荷兰国旗，是因为可以将红白蓝三色小球想象成条状物，有序排列后正好组成荷兰国旗。

本例和例 9.5 的主要区别是置换谓词 `permut` 的含义不一样。例 9.5 是面向任意一组数据的置换，而本例是面向只有 3 种不同值的置换，因而本例的 `permut` 定义相对简单。 □

按例 9.5 中归纳谓词 `permut(array t1, array t2)` 的定义方式来改写例 9.6 的 `permut` 定义是可行的。但是由于例 9.6 的问题简单，没有必要采取适用面更宽的定义。比较两个例子可以看到，例 9.5 的 `permut` 是归纳谓词，而例 9.6 的 `permut` 是一般谓词。例 9.5 需要引理，而例 9.6 不需要。例 9.6 的验证比例 9.5 的要简单。

在需要引入谓词定义、归纳定义和引理等才能完成代码的验证时，定义和引理的使用尽量恰到好处，能完成代码验证就行，而不必让定义和引理在更大的范围内适用，以免增加验证的负担。下面可以用例 9.5 的归纳谓词 `permut` 再讨论一下。

例 9.5 的归纳谓词 `permut(array t1, array t2)` 的定义把 `t1` 经过多少个一次置换考虑在内，而对这个函数的验证来说，并不关心需要经过多少次置换，只要 `t2` 是 `t1` 的置换就可以。可以用一种较简单的形式来定义 `permut`：

$$\begin{aligned}
 \text{inductive permut(array t1, array t2) =} \\
 & (\exists \text{ integer } i:[0..\text{length}(t1)-1], \text{ integer } j:[0..\text{length}(t1)-1]. i \leq j \ \&\& \\
 & \quad (\forall \text{ integer } k:[0..i-1]. t1[k] == t2[k]) \ \&\& \ t1[i] == t2[j] \ \&\& \\
 & \quad (\forall \text{ integer } k:[i+1..j-1]. t1[k] == t2[k]) \ \&\& \ t1[j] == t2[i] \ \&\& \\
 & \quad (\forall \text{ integer } k:[j+1..\text{length}(t1)-1]. t1[k] == t2[k]) \ \parallel \\
 & \quad \exists \text{ array } t3. \text{permut}(t1, t3) \ \&\& \ \text{permut}(t3, t2);
 \end{aligned}$$

注意，该定义第 1 行有断言 $i \leq j$ ，而例 9.5 中 `permut` 定义第 1 行上的是 $i < j$ ，因为后者的定义单独考虑了 0 次置换。

```

#define LEN 1000
typedef int array[LEN];
/*@ predicate permutOne(array t1, array t2) =
    (exists integer i:[0..\length(t1)-1], integer j:[0..\length(t1)-1]. i < j &&
        (\forall integer k:[0..i-1].t1[k] == t2[k]) && (\forall integer k:[i+1..j-1].t1[k] == t2[k]) &&
        (\forall integer k:[j+1..\length(t1)-1].t1[k] == t2[k]) && t1[i] == t2[j] && t1[j] == t2[i]);
    inductive permutAny(array t1, array t2, int n) =
        n == 0 && (\forall integer i:[0..\length(t1)-1].t1[i] == t2[i]) || n == 1 && permutOne(t1, t2) ||
        n > 1 && (\exists array t3.permutAny(t1, t3, n-1) && permutOne(t3, t2)); */
/*@ ghost array tempa; */      /*@ logic array olda; */
/*@ requires olda[0..LEN-1] == a[0..LEN-1] && \length(a) == LEN;
    assigns tempa, a[0..LEN-1];
    ensures (\forall integer i:[0..LEN-2]. a[i]<=a[i+1]) && (\exists integer n. n >= 0 &&
        permutAny(olda, a, n)) && \length(a) == LEN; */
void bubbleSort(int* const a) {
    int low, up, j, k, temp;      /*@ ghost int n; */
low = 0; up = LEN-1; k = up;    //@ ghost n = 0;
/*@ lemma permutAny_trans: \forall integer n.
    n >= 1 && permutAny(olda, tempa, n-1) && permutOne(tempa, a) ==> permutAny(olda, a, n); */
/*@ loop invariant
    low == 0 && up == LEN-1 && k == up && \length(a) == LEN && 0 == n &&
        (\forall integer i:[k+1..up-1].a[i]<=a[i+1]) && permutAny(olda, a, 0) ||
    low == 0 && up == LEN-1 && low <= k < up && \length(a) == LEN && 0 <= n &&
        (\forall integer i : [low..k]. a[i]<=a[k+1]) &&
        (\forall integer i : [k+1..up-1]. a[i]<=a[i+1]) && permutAny(olda, a, n); */
while(k != low) { j = low;
    /*@ loop invariant
        low == 0 && up == LEN-1 && k == up && low <= j <= k && \length(a) == LEN && 0 == n &&
            (\forall integer i : [low..j-1]. a[i]<=a[j]) && (\forall integer i : [k+1..up-1]. a[i]<=a[i+1]) &&
            permutAny(olda, a, n) ||
        low == 0 && up == LEN-1 && k == up && low <= j <= k && \length(a) == LEN && 0 < n &&
            (\forall integer i : [low..j-1]. a[i]<=a[j]) && (\forall integer i : [k+1..up-1]. a[i]<=a[i+1]) &&
            permutAny(olda, a, n) ||
        low == 0 && up == LEN-1 && low < k < up && low <= j <= k && \length(a) == LEN && 0 < n &&
            (\forall integer i : [low..j-1]. a[i]<=a[j]) && (\forall integer i : [low..k]. a[i]<=a[k+1]) &&
            (\forall integer i : [k+1..up-1]. a[i]<=a[i+1]) && permutAny(olda, a, n); */
while(j != k) {
    if (a[j] > a[j+1]) {
        /*@ ghost tempa[0..LEN-1] = a[0..LEN-1];
            temp = a[j]; a[j] = a[j+1]; a[j+1] = temp;
            /*@ ghost n = n + 1;
        }
        j = j + 1;
    }
    k = k - 1;
}
return;
}

```

图 9.6 冒泡排序函数的代码和标注

```

typedef enum {BLUE, WHITE, RED} color;
/*@ logic int numof(color* t, int n, color c) = (n <= 0) ? 0 : ((t[n-1] == c ? 1 : 0) + numof(t, n-1, c));
    predicate permut(color* t1, color* t2, int n) =
        \length(t1) == n && \offset(t1) == 0 && \length(t2) == n && \offset(t2) == 0 && n > 0 &&
        numof(t1, n, RED) == numof(t2, n, RED) && numof(t1, n, WHITE) == numof(t2, n, WHITE) &&
        numof(t1, n, BLUE) == numof(t2, n, BLUE); */
/*@ ghost color tmpColors[];

/*@ requires 0 <= i <= j && \length(t) > j && \forall int k:[0..\length(t)-1].tmpColors[k] == t[k];
    assigns t[i],t[j];
    ensures t[i] == tmpColors[j] && t[j] == tmpColors[i] &&(\forall int k:[0..i-1].tmpColors[k] == t[k]) &&
        (\forall int k:[i+1..j-1].tmpColors[k] == t[k])&&(\forall int k:[j+1..\length(t)-1].tmpColors[k] == t[k]);*/
void swap(color* const t, int const i, int const j){
    color tmp;
    tmp = t[i]; t[i] = t[j]; t[j] = tmp;
}
#define N 100
typedef struct flag{int n; color* colors;}flag;
/*@ strong type invariant is_colored(flag f) = 0 < f.n <= N && \length(f.colors)==f.n && \offset(f.colors)==0;
/*@ predicate isMonochrome(color* t, int i, int j, color c) = \forall int k:[i..j]. t[k] == c;
/*@ logic flag oldf;
/*@ requires oldf == f;
    assigns tmpColors[0..f.n-1], f.colors[0..f.n-1];
    ensures (\exists int b:[0..f.n]. \exists int r:[0..f.n].
        b <= r && isMonochrome(f.colors, 0, b-1, BLUE) && isMonochrome(f.colors, b, r-1, WHITE) &&
        isMonochrome(f.colors, r, f.n-1, RED) && permut(oldf.colors, f.colors, f.n)); */
void dutchFlag(const flag f){
    int b = 0,i = 0; int r; color* t; color tmp;
    /*@ lemma counting: (\forall int i:[0..f.n-1].tmpColors[i] == f.colors[i])
    ==> numof(tmpColors, f.n, RED) == numof(f.colors, f.n, RED) && numof(tmpColors, f.n, WHITE) ==
        numof(f.colors, f.n, WHITE) && numof(tmpColors, f.n, BLUE) == numof(f.colors, f.n, BLUE); */
    t = f.colors; r = f.n;
    /*@ ghost tmpColors[0..f.n-1] = f.colors[0..f.n-1];
    /*@ loop invariant
        0 <= b <= i <= r <= f.n && oldf == f && \length(t) == f.n && \offset(t) == 0 && t == f.colors &&
        isMonochrome(f.colors, 0, b-1, BLUE) && isMonochrome(f.colors, b, i-1, WHITE) &&
        isMonochrome(f.colors, r, f.n-1, RED) &&
        permut(oldf.colors, tmpColors, f.n) && tmpColors[0..f.n-1] == f.colors[0..f.n-1]; */
    while(i < r) {
        switch (t[i]) {
            case BLUE: swap(t, b, i); b = b+1; i = i+1; break;
            case WHITE: i = i+1; break;
            case RED: r = r-1; swap(t, i, r); break;
        }
        /*@ ghost tmpColors[0..f.n-1] = f.colors[0..f.n-1];
    }
}
}

```

图 9.7 Dijkstra 荷兰国旗算法的代码和标注

下面通过一个例子介绍幽灵函数的语法, 主要问题是怎样表达幽灵函数的函数协议和循环不变式。若需要将下面这样一个函数

```
#define N 100
int count(int* a){ // 统计长度为 N 的数组 a 中值为 0 的元素个数
    int k, j;
    k = 0;
    for(j = 0; j < N; j = j + 1){
        if(a[j] == 0){ k = k+1; }
    }
    return k;
}
```

作为幽灵函数定义, 则需要修改成下面的形式。

```
#define N 100 // 全局的定义和声明等照旧
/*@ ghost // 幽灵函数定义的开始
    /@ requires \length(a) == N && \offset(a) == 0;
        ensures 0 <= \result <= N;
    @/ // 幽灵函数协议的前后是用/@和@/做标记
int count(int* a){ // 统计长度为 N 的数组 a 中值为 0 的元素个数
    int k, j;
    k = 0;
    /@ loop invariant \length(a) == N && \offset(a) == 0
        && 0 <= k <= j && 0 <= j <= N;
    @/ // 循环不变式的前后也是/@和@/
    for(j = 0; j < N; j = j + 1){
        if(a[j] == 0){ k = k+1; }
    }
    return k;
}
*/// 幽灵函数定义的结束
```

幽灵函数的使用小结。

1. 和幽灵语句一样, 幽灵函数的标注也是以/*@ ghost 开始, 一直到该标注的结束。
2. 幽灵函数的函数协议和函数中的循环不变式、循环变式都以/@开始并且都以@/结束。因为标注不能嵌套, /@... @/是对幽灵函数内的标注(如函数协议)进行特殊处理的提示。非幽灵函数中的幽灵循环语句的循环不变式、循环变式也按此方式处理。其它标注中若出现/@... @/, 将被作为非法字符, 不会被按此方式处理。
3. 在幽灵函数中, 对幽灵语句的限制同本节开头提到的那几点是一致的。
4. 幽灵函数用于帮助收集程序点的状态能满足的断言, 一般来说不会复杂, 毕竟它不能破坏当前程序状态。

9.3 幽灵变量和逻辑变量的区别

幽灵变量和逻辑变量一致之处是, 它们都是为了方便写程序规范而引入的, 它们都不能

被程序访问。但是它们是有区别的，因而不能相互取代。逻辑变量仅可以出现在断言中，用于表达程序代码（主要是函数）对上下文的不变性质。而幽灵变量出现在程序标注里面的幽灵语句中，在幽灵语句中可以对幽灵变量赋值。幽灵语句好像被执行而实际并不执行。幽灵变量用于帮助表达函数代码本身的性质，因而幽灵变量也可以出现在断言中。

- 幽灵变量的优点是，可以通过对幽灵变量的赋值，记住程序变量在某个程序点的值。逻辑变量做不到这一点。

- 使用逻辑变量的场合，也可以通过给函数增加带 `const` 限定的幽灵参数而取代逻辑变量。这么做的缺点是对被验证函数增加的标注较多，没有逻辑变量方式直观。

- 幽灵变量的声明可以出现在头文件中。与逻辑变量不同的是，包含该头文件的各个源文件认定该幽灵变量代表同一个幽灵对象。

附录一：供导入的逻辑模块

1. 模运算模块

```
module \mod_module={
  lemma \mod_property_1:
    \forall integer a,integer b.a >= 0 && b > 0 ==> a%b == a-a/b*b;
  lemma \mod_property_2:
    \forall integer a,integer b.a >= 0 && b > 0 && a < b ==> a%b == a;
  lemma \mod_property_3:
    \forall integer a,integer b.b > 0 && b <= a < 2*b ==> a%b == a-b;
  lemma \mod_add_property:
    \forall integer a,integer b, integer p.a>=0 && b>=0 && p>0
    ==> (a+b)%p==(a%p+b%p)%p;      //模加法
  lemma \mod_sub_property:
    \forall integer a,integer b, integer p.a>=0 && b>=0 && p>0
    ==> (a-b)%p==(a%p-b%p)%p;      //模减法
  lemma \mod_mul_property:
    \forall integer a,integer b, integer p.a>=0 && b>=0 && p>0
    ==> (a*b)%p==(a%p*b%p)%p;      //模乘法
  lemma \mod_eq1_property:
    \forall integer a,integer b. a>=0 && b>0
    ==> (a%b)%b==a%b;              //恒等式
  lemma \mod_ap_a_property:
    \forall integer a,integer b,integer c,integer p.a>=0&& b>=0 && c>=0 && p>0
    ==> ((a+b)%p+c)%p == (a+(b+c)%p)%p;      //加法结合律
  lemma \mod_apm_property:
    \forall integer a,integer b,integer c,integer p.a>=0&& b>=0 && c>=0 && p>0
    ==> ((a*b)%p*c)%p == (a*(b+c)%p)%p;      //乘法结合律
  lemma \mod_cpa_property:
    \forall integer a,integer b, integer p.a>=0 && b>=0 && p>0
    ==> (a+b)%p==(b+a)%p;          //加法交换律
  lemma \mod_cpm_property:
    \forall integer a,integer b, integer p.a>=0 && b>=0 && p>0
    ==> (a*b)%p==(b*a)%p;          //乘法交换律
  lemma \mod_dl_property:
    \forall integer a,integer b,integer c,integer p.a>=0&& b>=0 && c>=0 && p>0
    ==> ((a+b)%p*c)%p == ((a*c)%p+(b*c)%p)%p; //分配律
  lemma \mod_congruence_1:
    \forall integer a,integer b,integer p.a>=0 && b>=0 && p>0 && a%p==b%p
    ==> (a-b)%p==0;                //同余整除性
  lemma \mod_congruence_2:
    \forall integer a,integer b,integer c,integer p.
    a>=0 && b>=0 && p>0 && a%p==b%p && b%p==c%p
```

```

=> a%p==c%p; //同余传递性
lemma \mod_congruence_3:
  \forall integer a,integer b,integer c,integer p.
    a>=0 && b>=0 && c>=0 && p>0 && a%p==b%p
  => (a+c)%p==(b+c)%p;
lemma \mod_congruence_4:
  \forall integer a,integer b,integer c,integer p.
    a>=0 && b>=0 && c>=0 && p>0 && a%p==b%p
  => (a*c)%p==(b*c)%p;
lemma \mod_congruence_5:
  \forall integer a,integer b,integer c,integer d,integer p.
    a>=0 && b>=0 && c>=0 && d>=0 && p>0 &&
    a%p==b%p && c%p==d%p
  => (a+c)%p==(b+d)%p;
lemma \mod_congruence_6:
  \forall integer a,integer b,integer c,integer d,integer p.
    a>=0 && b>=0 && c>=0 && d>=0 && p>0 &&
    a%p==b%p && c%p==d%p
  => (a-c)%p==(b-d)%p;
lemma \mod_congruence_7:
  \forall integer a,integer b,integer c,integer d,integer p.
    a>=0 && b>=0 && c>=0 && d>=0 && p>0 &&
    a%p==b%p && c%p==d%p
  => (a*c)%p==(b*d)%p;
lemma \mod_congruence_8:
  \forall integer a,integer b,integer p,integer k.a>=0 && b>=0 && p>0 && k>0
  => (a%p==b%p <==> (a*k)%(p*k)==(b*k)%(p*k));
}

```

2. 整除运算模块

```

module \div_module={
  lemma \div_property_1:
    \forall integer a,integer b,integer c.a >= 0 && b >= 0 &&
    c>0 && a%c==0 && b%c==0
  => (a+b)%c==0;
  lemma \div_property_2:
    \forall integer a,integer b,integer c.a >= 0 && b >= 0 &&
    c>0 && a%c==0 && b%c==0
  => (a-b)%c==0;
  lemma \div_property_3:
    \forall integer a,integer b,integer c.a >= 0 && b >= 0 &&
    c>0 && a%b==0
  => (a*c)%b==0;
  lemma \div_property_4:

```

```

        \forall integer a,integer b,integer c.a >= 0 && b >= 0 && c>0
    ==> (a+b*c)/c==a/c+b;
lemma \div_property_5:
    \forall integer a,integer b,integer c.a >= 0 && b > 0 && c>0
    ==> a/b/c==a/(b*c);
lemma \div_property_6:
    \forall integer a,integer b.a>=0 && b > 0 && a%b==0
    ==> (a+b)%b == 0;
}

```

3.数学模块

```

module \math_module={
    logic real \rabs(real a) =
        a > 0.0 ? a : -a;
    logic integer \sum(integer low,integer up) =
        up < low ? 0 : \sum(low, up - 1) + up;
    logic integer \power(integer x,integer n) =
        n <= 0 ? 1 : x * \power(x, n - 1);
    lemma \rabs_property1:
        \forall real x, real y. \rabs(x) <= y <==> -y <= x <= y;
    lemma \rabs_property2:
        \forall real x, real y. \rabs(x) >= y <==> x >= y || x <= -y;
    lemma \rabs_property3:
        \forall real x, real y. \rabs(x + y) <= \rabs(x) + \rabs(y);
    lemma \rabs_property4:
        \forall real x, real y. \rabs(x * y) == \rabs(x) * \rabs(y);
    lemma \rabs_property5:
        \forall real x,real y,real z. \rabs(x - z) <= \rabs(x - y) + \rabs(y - z);
    lemma \sum_property1:
        \forall integer low, integer up. 0 <= low <= up
        ==> \sum(low, up) == (low + up) * (up - low + 1) / 2;
    lemma \sum_property2:
        \forall integer low, integer middle, integer up. 0 <= low <= middle <= up
        ==> \sum(low, up) == \sum(low, middle) + \sum(middle + 1, up);
}

```

4.数组模块 //目前仅适用于整型数组

```

module \array_module={
    predicate \array_sorted_inc(integer* a) =
        \forall integer n:[0..\length(a) - 2].a[n] < a[n + 1];
    logic integer \array_sum(integer* a, integer low, integer up) =
        up < low ? 0 : a[up] + \array_sum(a, low, up - 1);
    lemma \array_sorted_inc_property1:
        \forall integer* a,integer value.

```

```

        \array_sorted_inc(a) && a[\length(a) - 1] <= value
    ==> \forall integer k:[0..\length(a)-1].a[k] <= value;
lemma \array_sorted_inc_property2:
    \forall integer *a,integer value.
        \array_sorted_inc(a) && (\forall integer k:[0..\length(a) - 1].a[k] < value
    ==> (\forall integer n:[0..k].a[n] < value));
lemma \array_sum_property:
    \forall integer* a,integer low,integer up,integer middle.
        0 <= low <= middle <= up
    ==> \array_sum(a, low, up) ==
        \array_sum(a, low, middle) + \array_sum(a, middle + 1, up);
}

```

5.形状图_单向链表

/*仅适用于以下结构定义

```

typedef struct Node {
    int data;
    struct Node* next;
}Node;*/
module \list_module{
    inductive \sorted_list(Node* p) = //有序链表
        p == \null ||
        p != \null && p->next == \null ||
        p != \null && p->next != \null &&
            p->data <= p->next->data && \sorted_list(p->next);
    inductive \sorted_list_seg(Node *p, Node *q) = //有序表段
        p == q && p != \null ||
        p != q && p != \null && p->next != \null && p->data <= p->next->data &&
            \sorted_list_seg(p->next, q);
    lemma \sorted_list_seg_property1: //有序表段性质
        \forall Node *p, Node *q.
            \sorted_list_seg(p, q) && q->data <= q->next->data && q->next != \null
    ==> \sorted_list_seg(p, q->next);
    lemma \sorted_list_seg_property2: //有序表段性质
        \forall Node *p, Node *q.
            \sorted_list_seg(p, q) && \sorted_list(q)
    ==> \sorted_list(p);
}

```

6.形状图_双向链表

/*仅适用于以下结构定义

```

typedef struct Node {
    int data;
    struct Node* r;

```

```

    struct Node* l;
}Node;*/
module \dlist_module{
  inductive \inc_sorted_dlist_seg(Node * p, Node * q) =
    p == q && p != \null ||
    p != q && p->r->l == p &&
      p->data <= p->r->data && \inc_sorted_dlist_seg(p->r, q);
  predicate \inc_sorted_dlist(Node * p) =
    p == \null ||
    p->l == \null && p->r == \null ||
    p->l == \null &&
      (\exists Node* q.\inc_sorted_dlist_seg(p, q) && q->r == \null);
  predicate \almost_inc_sorted_dlist(Node * p) =
    \exists Node *q. \inc_sorted_dlist_seg(p, q) && q->r == \null;
  inductive \dec_sorted_dlist_seg(Node * p, Node * q) =
    p == q && p != \null ||
    p != q && p->r->l == p &&
      p->data >= p->r->data && \dec_sorted_dlist_seg(p->r, q);
  predicate \dec_sorted_dlist(Node * p) =
    p == \null ||
    p->l == \null && p->r == \null ||
    p->l == \null &&
      \exists Node* q.\dec_sorted_dlist_seg(p, q) && q->r == \null;
  predicate \almost_dec_sorted_dlist(Node * p) =
    \exists Node *q. \dec_sorted_dlist_seg(p, q) && q->r == \null;
  lemma \inc_sorted_dlist_seg_push_back_lemma: \forall Node *p, Node *q.
    \inc_sorted_dlist_seg(p, q) && q->r->l == q && q->data <= q->r->data
    ==> \inc_sorted_dlist_seg(p, q->r);
  lemma \dec_sorted_dlist_seg_push_back_lemma: \forall Node *p, Node *q.
    \dec_sorted_dlist_seg(p, q) && q->r->l == q && q->data >= q->r->data
    ==> \dec_sorted_dlist_seg(p, q->r);
  lemma \inc_sorted_dlist_seg_pop_back_lemma:
    \forall Node *p, Node *q, Node *pre, int n.
      n >= 0 && \inc_sorted_dlist_seg(p, q) && p->(r:n) == pre && pre->r == q
      ==> \inc_sorted_dlist_seg(p, pre);
  lemma \inc_sorted_dlist_seg_back_lemma: \forall Node *p, Node *q, Node *pre, int n.
    n >= 0 && \inc_sorted_dlist_seg(p, q) && p->(r:n) == pre && pre->r == q
    ==> pre->data <= q->data;
  lemma \dec_sorted_dlist_lemma: \forall Node *p, Node *q.
    \dec_sorted_dlist_seg(p, q) && p->l == \null && \almost_dec_sorted_dlist(q)
    ==> \dec_sorted_dlist(p);
}

```

7. 形状图_树

/*仅适用于以下结构定义

```
typedef struct Node {
    int data;
    struct Node* r;
    struct Node* l;
}Node;*/

module \tree_module{
    inductive \gt(integer x, Node* p) =
        p == \null ||
        p != \null && x > p->data && \gt(x, p->l) && \gt(x, p->r);
    inductive \lt(integer x, Node* p) =
        p == \null ||
        p != \null && x < p->data && \lt(x, p->l) && \lt(x, p->r);
    lemma \lt_property:
        \forall integer x, integer y, Node* p.
            x < y && \lt(y, p) ==> \lt(x, p);
    lemma \gt_property:
        \forall integer x, integer y, Node* p.
            x > y && \gt(y, p) ==> \gt(x, p);
}
```

参考文献

- [1] Patrick Baudin et al. ACSL: ANSI/ISO C specification language. Version 1.8. 2014. <http://frama-C.com/download/acsl.pdf>, <http://frama-C.com/download/frama-c-acsl-implementation.pdf>.
- [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] 张志天, 李兆鹏, 陈意云, 刘刚, 一个程序验证器的设计和实现, *计算机研究与发展*, 50(5): 1044-1054, 2013. 见 <http://staff.ustc.edu.cn/~yiyun/>.
- [4] 冯舜玺译, 数据结构与算法分析—C语言描述, 机械工业出版社, 2004.
- [5] 张昱、陈意云、李兆鹏, 形状图理论的定理证明, *计算机学报*. 见 <http://staff.ustc.edu.cn/~yiyun/>.
- [6] 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.
- [7] 赵世忠, 算术表达式的一种可信计算算法及其软件 ISReal, *中国科学: 信息科学*, 第46卷第6期, 698-713. 2016.
- [8] 陈意云等, 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, 2020