# 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

# 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*），可用来编写程序标注。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式定义、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

# 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*），可用来规范安全 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，文法规则本质上用 BNF（Backus-Naur Form，巴科斯-诺尔范式）。在文法规则中，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 |
|---|---|---|
| >= | ≥ | 0x2265 |
| <= | ≤ | 0x2264 |
| ==> | ⇒ | 0x21D2 |
| <==> | ⇔ | 0x21D4 |
| && | ∧ | 0x2227 |
| \|\| | ∨ | 0x2228 |
| ^^ | ⊕ | 0x22BB |
| ! | ¬ | 0x00AC |
| - (unary minus) | - | 0x2212 |
| \forall | ∀ | 0x2200 |
| \exists | ∃ | 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。

| | | | |
|---|---|---|---|
| *term* | → | *literal* | //直接常量 |
| | \| | *id* | //变量 |
| | \| | *unary-op term* | //一元项 |
| | \| | *term bin-op term* | //二元项 |
| | \| | *term* [ *term* ] | //数组元素访问 |
| | \| | *term* [ *term* .. *term* ] | //数组区间访问 |
| | \| | *term* [ *term* .. \infinity] | // \infinity 的含义见第 9 章 |
| | \| | *term* . *id* | //结构体域访问 |
| | \| | *term* -> *id* | //指向对象的域访问 |
| | \| | *term* -> (*id* (, *id*)*: *term*) | //带折迭域的指向对象的域访问 |
| | \| | ( *type-exp* ) *term* | //类型强制 |
| | \| | *id* ( *term* (, *term*)* ) | //函数应用 |
| | \| | ( *term* ) | //加括号的项 |
| | \| | *term* ? *term* : *term* | //三元的条件项 |
| | \| | \length ( *term* , *id* ) | //链表长度函数 |
| | \| | \length ( *term* ) | //一维数组长度和指针指向内存块长度函数 |
| | \| | \dimension( *term, term* ) | //多维数组中，求各维长度的函数 |
| | \| | \offset ( *term* ) | //指针指向位置在所属内存块中的偏移 |
| | \| | \string (*term, term* ) | //取字符数组某个长度的逻辑字符串函数 |
| | \| | \strlen (*term*) | //计算物理字符串的长度 |
| | \| | \prefix ( *term, term* ) | //字符串前缀函数 |
| | \| | \suffix ( *term, term* ) | //字符串后缀函数 |
| | \| | \index(*term, term* ) | //字符串在另一个字符串中首次出现位置 |
| | \| | \char(*term, term*) | //取字符串中某个位置的字符 |
| | \| | \result | //函数的结果 |
| | \| | \lambda *binders* . *term* | //抽象 |
| | \| | \old ( *term* ) | //term 在函数入口点的值 |
| | \| | *ext-quantifier* ( *term* , *term* , *term* ) | //新增的 |
| *literal* | → | \true \| \false | //布尔常量 |
| | \| | \null | // NULL 指针常量 |
| | \| | *integer* | //整型常量 |
| | \| | *real* | //实型常量 |
| | \| | *string* | //串常量 |
| | \| | *character* | //字符常量 |
| *bin-op* | → | + \| - \| * \| / \| % \| << \| >> | //算术运算符 |
| | \| | == \| != \| <= \| >= \| > \| < | //关系运算符 |
| | \| | && \| \|\| \| ^^ | //布尔运算符 |
| | \| | & \| \| \| ^ | //按位运算符 |
| *unary-op* | → | + \| - | //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 |
| *ext-quantifier* | → | \max \| \min \| \sum \| \product \| \numof | //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: ∧）、||（UTF8: ∨）和!（UTF8: ¬）作为逻辑连接词，新增逻辑连接词有蕴涵==>（UTF8: ⇒）、等价<==>（UTF8: ⇔）和异或^^（UTF8: ⊕）。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）。

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

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 $t_1\ relop_1\ t_2\ relop_2\ t_3\ \ldots\ t_k$ 是$(t_1\ relop_1\ t_2)$ && $(t_2\ relop_2\ t_3)$ && … && $(t_{k-1}\ relop_{k-1}\ t_k)$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 τ *x.e* 或\forall τ *x*:[$a_1$…$b_1$, …, $a_n$…$b_n$].*e* 表示，存在量化由\exists τ *x.e* 或\exists τ *x*:[$a_1$…$b_1$, …, $a_n$…$b_n$].*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 | (), [], ->, . |
| 一元 | 右 | !, ~, +, -, *, &, (*type*), sizeof |
| 乘除 | 左 | *, /, % |

| | | |
|---|---|---|
| 加减 | 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∀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：

- $|d \times q| \le n$，$|q|$对这个性质来说是最大的；
- 如果$|n| < |d|$，那么 $q$ 是 0；
- 如果$|n| \ge |d|$并且 $n$ 和 $d$ 有同样的符号，则 $q$ 是正的；
- 如果$|n| \ge |d|$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 $2^{8\times sizeof(t)}$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) mod 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 == ...00111 & ...001100 == ...00100 == 4
- -8 | 5 == ...11000 | ...00101 == ...11101 == -3
- ~5 == ~...00101 == ...111010 == -6
- -5 << 2 == ...11011 << 2 == ...11101100 == -20
- 5 >> 2 == ...00101 >> 2 == ...0001 == 1
- -5 >> 2 == ...11011 >> 2 == ...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*）、+∞和-∞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”、+∞和-∞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 0x*hh.hh*p±*dd*，其中 *h* 是 16 进制数字，*d* 是 10 进制数字，0x*hh.hh*p±*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 $13421773 \times 2^{-27}$，后者等于 0.100000001490116119384765625。如果源实数太大，则导致结果是+∞和-∞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 $e_1$ op $e_2$ 的浮点结果和逻辑表达式(float)($e_1$ 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 ≡ 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、+∞和-∞。

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 == (float)(clockSum(i-1)+(float)0.1) ==> t == 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 == (float)(0.1*(i-1)) + (float)0.1==> t == (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：$3 \le n \le 31$。

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 ≤ 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 <= \offset(p) < \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。

## 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。

这个函数仅在(\is_string(str, m) || \is_pstring(str, m)) && m >= n 情况下使用。先前介绍说\string(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 >= 0 且 n >= 0）

它们的类型都是 STRING × STRING → 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 “==” (STRING \string(str1, m), STRING \string(str2, n)) =
    m == n && (\forall integer i:[0..m-1].str1[i] == str2[i]);
```

这是字符串\string(str1, m)和\string(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 “<” (STRING \string(str1, m), STRING \string(str2, n)) =
    \exists integer k:[0..n-1].                    //若 n== 0，则结果一定为\false
        ((\forall integer i:[0..k-1].str1[i] == str2[i]) &&    //这两行针对 m >= 0, n > 0
         (k >= m || str1[k] < str2[k]));       //若 k >= m，则 n > m 且两者前 k-1 个
                                               //字符相同，则结果一定为\true
```

6．逻辑字符串并置函数“+”（使用条件为 m >= 0 且 n >= 0）

该函数的类型是: STRING × STRING → STRING。

```
logic STRING “+” (STRING \string(str1, m), STRING \string(str2, n)) =
                                                    \string(str3, m+n)，
```

其中 str3 是新的字符数组，它满足：

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

else 结果\string(str3, m+n)满足

str3[0..m-1] == str1[0..m-1] && str3[m..m+n-1] == str2[0..n-1]。

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

该函数的类型是：STRING × integer → STRING。

```
logic STRING \prefix(STRING \string(str, m), integer n) = \string(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 >= 0，n >= 0 且 n <= m）

该函数的类型是：STRING × integer → STRING

```
logic STRING \suffix(STRING \string(str, m), integer n) =
                                    \string(str+(m-n), n);   (= 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 >= 0）

该谓词的类型是：STRING × char → boolean，它只用于有长度信息的逻辑字符串。

```
inductive \membership(STRING \string(str, m), char c) =
    m == 0 ?
        \false :
```

```
m >= 1 && str[0] == c ?
    \true:
    \membership(\string(str+1, m-1), c);
```

10．逻辑字符串 str1 是否包含 str2 的谓词 \contains（使用条件为 m >= 0 且 n >= 0）

该归纳谓词的类型是：STRING × STRING → boolean

```
inductive \contains(STRING \string(str1, m), STRING \string(str2, n)) =
    m < n ?
        \false :
        \prefix(\string(str1, m), n) == \string(str2, n) ?
            \true :
            \contains(\string(str1+1, m-1), \string(str2, n));
```

11．计算 str2 首次出现在 str1 中的索引的函数 \index（使用条件为 m >= 0 且 n >= 0）

该函数的类型是：STRING × STRING → long

```
logic long \index(STRING \string(str1, m), STRING \string(str2, n)) =
    m < n ?
        n-1 :
        \prefix(\string(str1, m), n) == \string(str2, n) ?
            0 :
            1+ \index(\string(str1+1, m-1), \string(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 \char（使用条件为 m > = n > 0）

该函数的类型是 STRING × long → char。

```
logic char \char(STRING \string(str, m), long n) = 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，函数协议的文法见图 4.1。从文法上看，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。

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

图 4.1 函数协议的文法

## 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 $P_1$ && … && $P_m$ 成立。$P_1$ && … && $P_m$ 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，只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 $P_i(1 \le i \le m)$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 $Q_1$ && … && $Q_k$ 成立，$Q_1$ && … && $Q_k$ 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 $Q_i(1 \le i \le k)$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．$V_1, \ldots, V_n$ 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）$V_1, \ldots, V_n$ 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，$V_1, \ldots, V_n$ 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]（$0 \le i < 100$）都是一维数组名，同样不可能被赋值。

另一种特殊情况是，$V_1, \ldots, V_n$ 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 $V_1, \ldots, V_n$ 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）$V_1, \ldots, V_n$ 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 $V_i$（$1 \le i \le n$）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 $V_i$ -> 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 $V_1, \ldots, V_n$ 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 >= 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，这些行为的名字必须有区别。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 子句在语法上有两点限制。

（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\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\old(x) == x。
- 若代码中有不允许函数对 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会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\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 $A_i$ && $R_i$（$1 \le i \le 2$），则在调用点后除了得到 Q 以外，还能得到 $Q_1$ 和/或 $Q_2$。

3．$A_i$ && $R_i$（$1 \le i \le 2$）还有这样的特点，若 $A_i$ 为真则 $R_i$ 为真，即 $A_i$ ==> $R_i$。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 && ($A_1$ ==> $R_1$) && ($A_2$ ==> $R_2$)成立。

2．被调用函数必须保证，其返回状态使得性质 Q && ($A_1$ ==> $Q_1$) && ($A_2$ ==> $Q_2$)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 $V_j$（$j \ne i$）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 $b_1$, …, $b_n$; */

其含义是，行为 $b_1$, …, $b_n$ 的集合是完备的，即

$P \Longrightarrow (A_1 \| \ldots \| A_n)$

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 $Q \&\& (\old(A_1) \Longrightarrow Q_1) \&\& \ldots \&\& (\old(A_n) \Longrightarrow Q_n)$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 $b_1$, …, $b_n$; */

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

$P \Longrightarrow !(A_i \&\& A_j)$

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 $Q \&\& (\old(A_1) \Longrightarrow Q_1) \&\& \ldots \&\& (\old(A_n) \Longrightarrow Q_n)$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 $Q \&\& (\old(A_1) \Longrightarrow Q_1) \&\& \ldots \&\& (\old(A_n) \Longrightarrow Q_n)$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：(!(oldsrc == olddest+k) || (oldsrc == olddest+k)) == \true 并且(!(oldsrc == olddest+k) && (oldsrc == olddest+k)) == \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。 □

```
//早年的串复制库函数 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 &&
          (\forall 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 &&
            (\forall int i:[0..n].olddest[i] == \old(olddest[i+k])) &&
            (\forall 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 早年的库函数 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。 □

```
/*@ 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

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 $A_1$、$A_2$、$A_3$ 和 $A_4$ 分别是 a == 1，a == 2，b == 1 和 b == 2，则 $A_1$ 和 $A_2$ 及 $A_3$ 和 $A_4$ 两个集合都是正交的。

## 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。

```
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

**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。 □

```
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

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。

## 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，就构成这里定义的树堆。

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，也包括了将和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的断言文法中的断言\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持见下面两点。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 和验证系统的支持如下。

- 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。

| | | |
|---|---|---|
| *compound-statement* | → | { *declaration** *statement** *assertion-check*? } |
| *statement* | → | *assertion-check*? *statement* |
| *assertion-check* | → | *assertion* |
| | \| | *check* |
| *assertion* | → | /*@ assert *assert* ; */ |
| *check* | → | /*@ check *assert* (, *assert* )* ; */ |
| *statement* | → | /*@ *loop-annot* */ while (*expr* ) *statement* |
| | \| | /*@ *loop-annot* */ for ( *expr* ; *expr* ; *expr* ) *statement* |
| | \| | /*@ *loop-annot* */ do *statement* while ( *expr* ) ; |
| *loop-annot* | → | *loop-invariant** *loop-variant*? |
| *loop-invariant* | → | loop invariant *assert* ; |
| *loop-variant* | → | loop variant *term* ; |
| | \| | loop variant *term* for *id* ; //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 ==> Q 得证，则把 P && Q 作为当前程序点断言，继续向下验证。若 P ==> 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 ==> $Q_1$，再证 P && $Q_1$ ==> $Q_2$。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 ==> Q 和 P && Q ==> R 都能证明，但因所用定理证明器的能力问题，P ==> 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 ==> 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。

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

$$\frac{\{P\}\ \mathrm{s}\ \{I\} \quad \{I \wedge \mathrm{c}\}\ \mathrm{s}\ \{I\}}{\{P\}\ \text{do s while (c)}\ \{I \wedge \neg \mathrm{c}\}}$$

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，elements[j] >= elements[j/2]（elements 是存放数据的一维数组）。element 是全局数组，elements[0]等于 INT_MIN，任何插入的值都不小于 elements[0]。size 是全局变量，指示 elements 数组中当前有多少个元素（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 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 && (\forall integer j:[1..size].elements[j] >= elements[j/2]) &&
        \old(elements[1]) == elements[1];
assigns elements, size;
ensures size >=0 && size <= CAPACITY-1 && (\forall 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]) &&
              (\forall 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]) &&
              (\forall 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] &&
              (\forall 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 <= x < 11。但是仅把 0 <= 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 <= 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。 □

```
//@ 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

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。

# 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) <==> x > y && x >= 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(τ x, τ 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。

```
/*@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

**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\old(n)表示 n 在函数入口点的值）：

n >= 0 && \old(n) == n && n != 0) && n != 1 ==> \old(n) > n-1 && n-1 >= 0，和

n >= 0 && \old(n) == n && n != 0) && n != 1 ==> \old(n) > n-2 ∧ n-2 >= 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选择不恰当。

```
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

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各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。

## 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 <= 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。