程序验证系统中断言语言的设计*

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

Design of Assertion Languages in Program verification systems

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

Abstract: There are a lot of prototype systems for program verification based on deduction inference, but there are almost no program verification systems which can be used in practical software development. The problems come from expressiveness of assertion languages, automatic inference of loop invariants and ability of automated theorem provers. In this paper, we consider how to divide the extra burden into pieces for programmers, verification condition generators and automated theorem provers respectively, while enhancing the expressiveness of assertion languages by allowing use-defined predicates and logical variables. This paper adopts a method of assisting automated theorem provers by providing inductive properties among user-defined predicates and designs an automated proof method for inductive properties of data structures. This paper also designs a method of transferring the extra burden from logical variables to verification condition generators instead of automated theorem provers.

Key words: Program Verification; Shape Graph Logic; Domain-Specific Logic; Inductive Theorem; Automated Theorem proving

摘要：基于演绎推理的程序验证系统原型已屡见不鲜，但目前几乎没有能真正用于软件系统开发的程序验证系统。根源在断言语言的表达能力、循环不变式的自动推断和自动定理证明器的能力等方面。本文研究，当允许在断言中使用用户自定义谓词和逻辑变量来增强断言语言的表达能力时，怎样把由此引起的负担分解到程序员、验证条件生成器和自动定理证明器上。本文采取由程序员提供自定义谓词之间的归纳性质来帮助自动定理证明器的方式，并设计了数据结构归纳性质的一种自动证明方法。其次，本文设计了将引入逻辑变量的额外负担加到验证条件生成器而不是自动定理证明器的方法。

关键词： 程序验证；形状图逻辑；领域专用逻辑；归纳定理；自动定理证明
中图法分类号: TP301 **文献标识码**: A

* Supported by the National Natural Science foundation of China under Grant No. 61170018, 61003043（国家自然科学基金）and China Postdoctoral Science Foundation (NO.2012M521250，中国博士后基金)

1 引言

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

形式验证是提高软件可信程度的重要方法。粗略地说，软件的形式验证有两条途径。第一条途径是模型检测，它通过遍历系统所有状态空间，能够对有穷状态系统进行自动验证，并自动构造不满足验证性质的反例。模型检测方法已经在工业界逐步得到应用。第二条途径是演绎推理，它使用形式方法对软件系统进行数学推理，其中通常要用到像 Isabelle/HOL[1] 或 Coq[2] 这样的定理证明软件。在这种途径中，大部分的研究围绕采用某种演算来产生验证条件，然后用某个或多个定理证明器来证明验证条件，如 Ynot[3]、Spec#[4]和 ESC/Java[5]。有些研究依靠符号计算及其过程中的定理证明来避免验证条件生成步骤，如 smallfoot[6]和 jStar[7]。还有的研究采用经严格证明的变换，从抽象规范逐步求精得到具体程序，如 Perfect Developer[8]。虽然这些工具已在实验室研发出来，但是尚无可供工业界使用的产品问世。究其原因，根源在于自动定理证明方面的困难。因为不管是断言语言表达能力的提升、领域专用逻辑的设计、循环不变式的推断、别名判断和验证条件的证明等，最终都受到自动定理证明器的能力的影响。

在研究自动定理证明技术的同时，也应该考虑怎样降低对自动定理证明器的能力的期待。例如，可以设计新的编程语言机制来提高合法程序的门槛，例如形状系统[9]，用以排除一些有逻辑错误的程序，减轻自动定理证明器的负担。另外，在设计领域专用逻辑的同时，设计其专用的自动定理证明器，降低对通用自动定理证明器的要求[10]。基于这些思路，我们设计了 PointerC 语言的程序验证系统原型[11, 12]，该验证系统首先进行形状分析，然后用形状分析得到的形状图[9]来支持随后验证条件的生成和证明，证明由可满足性模理论求解器 Z3[13]完成。

本文研究，当允许在断言中使用用户自定义谓词和逻辑变量来增强断言语言的表达能力时，怎样把由此引起的负担分解到程序员、验证条件生成器和自动定理证明器上，使得实现断言语言的这些增强成为可能。本文的贡献有如下三点。

第一，采取由程序员提供自定义谓词之间的归纳性质来帮助自动定理证明器的方式，用以克服基于演绎推理的自动定理证明器难以发现数据类型和数据结构归纳性质的困难，并设计了数据结构归纳性质的一种自动证明方法。

程序性质的描述可能需要用到归纳谓词，例如，若二叉树节点的类型是

```
typedef struct node{int data; Node* l; Node* r;}Node
```

则在假定所有指针都不相等的情况下，定义二叉树的归纳谓词是：

$$\text{tree(Node* p)} \triangleq \text{p == NULL} \vee \text{p!= NULL} \wedge \text{tree(p->l)} \wedge \text{tree(p->r)} \quad (1)$$

程序验证器一般不可能用内建谓词来囊括各种数据结构的定义，因此通常允许程序员自行定义归纳谓词，用以描述数据结构和程序的性质。

验证条件的证明可能还要用到数据结构的归纳性质。例如，图 1 所示的二叉排序树删除根节点的函数，用根节点的中序前驱（左子树的最右叶节点）的值来代替根节点的值，再删除原来的中序前驱节点。要证明结果仍然是二叉排序树，需要用到二叉排序树的两个性质：左子树上任何节点的数据都小于右子树上的所有数据，左子树上最右叶节点的数据大于左子树上其他所有数据。它们都是二叉排序树的归纳性质，但基于演绎推理的自动定理证明器难以从二叉排序树的归纳定义证明它们。

允许程序员提供性质引理带来的主要问题是，怎么检查这些引理的正确性。

第二，研讨了断言语言中的逻辑变量在程序验证中的使用方式，设计了把引入逻辑变量的额外负担加到验证条件生成器而不是自动定理证明器的方法。

断言语言中的逻辑变量用来表达程序对上下文的不变性质，这些性质用程序中的变量不足以表达。例如，

- 对任何 x，若 x 大于变量 m，则经过赋值语句 m = m – 1 后，x 大于 m + 1。
- 对任何 y，若 y 大于二叉排序树 t 上的所有数据，并且 y 大于插入的数据 data，则 y 大于结果二叉排

序树上的所有数据。

若允许对 Hoare 三元式加量词，则上述两句话的含义分别是

$\forall x{:}\mathbb{Z}.(\{x > m\}\ m = m - 1\ \{x > m + 1\})$ 和 $\forall y{:}\mathbb{Z}.(\{gt(y, t) \wedge y > data\}\ S\ \{gt(y, t)\})$

其中 S 代表二叉排序树插入函数的函数体，谓词应用 gt(y, t)表示 y 大于 t 所指向二叉排序树上的所有数据。

为避免对 Hoare 三元式加量词，把它们分别写成

$\{x > m\}\ m = m - 1\ \{x > m + 1\}$ 和 $\{gt(y, t) \wedge y > data\}\ S\ \{gt(y, t)\}$

其中区别于程序变量的 x 和 y 称为逻辑变量。

Hoare 三元式{P}S{Q}的断言 P 和 Q 中的逻辑变量有下述两个特点：

- 程序段 S 不能访问出现在 P 和 Q 中的逻辑变量。
- P 和 Q 中逻辑变量可以被含程序变量和逻辑变量的表达式代换。

引入逻辑变量带来的主要问题是，怎么证明函数调用点的断言蕴涵该函数的前条件。

第三，用上述方法扩展了 PointerC 语言的程序验证系统原型，使得该系统原型能够验证操作平衡二叉排序树、伸展树、AA 树和树堆等复杂易变数据结构的程序。

本文其余部分组织如下：第 2 节介绍用户自定义谓词以及它们之间的性质引理的证明方法，第 3 节讨论逻辑变量及其实现技术，第 4 节是系统概述与证明实例，第 5 节是与相关研究工作进行比较，第 6 节是总结。

2 用户自定义谓词及其实现技术

Hoare 三元式{P} S {Q}中，S 是程序段，P 和 Q 是断言。程序验证系统的断言语言一般是在相应编程语言的无副作用布尔表达式上扩展而成。扩展是为了便于描述各种构造类型的数据的性质，如使用量词来描述数组中存在某个值和数组的有序性会比较方便，而编程语言的布尔表达式中无量词。引言中提到，描述归纳数据结构的性质时，需要用到谓词。由于归纳数据结构的多样性，不可能通过系统内建谓词的方式来满足各种需要，因此系统需要让程序员定义自己所需的谓词，称为用户自定义谓词。描述归纳数据结构的谓词往往是归纳谓词，断言中若有归纳谓词的应用，一般来说是不可能通过谓词展开来使得谓词应用消失的。

下面三个谓词用来定义二叉排序树：

$$gt(int\ x, Node*\ p) \triangleq p{==}NULL \vee p\ !{=}\ NULL \wedge x > p{->}data \wedge gt(x, p{->}l) \wedge gt(x, p{->}r) \quad (2)$$

$$lt(int\ x, Node*\ p) \triangleq p{==}NULL \vee p\ !{=}\ NULL \wedge x < p{->}data \wedge lt(x, p{->}l) \wedge lt(x, p{->}r) \quad (3)$$

$$BST(Node*\ p) \triangleq p{==}NULL \vee p\ !{=}\ NULL \wedge BST(p{->}l) \wedge BST(p{->}r) \wedge gt(p{->}data, p{->}l) \wedge lt(p{->}data, p{->}r) \quad (4)$$

其中归纳谓词 gt 和 lt 分别表示 x 大于和小于 p 所指向的二叉树上的所有数据，归纳谓词 BST 基于前两个谓词来定义二叉排序树。注意，本文给出的谓词定义和性质引理，建立在已经用形状图定义了链表和二叉树等基本数据结构的基础上[9]。本文都是以操作归纳定义的易变数据结构的程序为例。

2.1 用户自定义谓词在程序验证中的作用

除了定义数据结构的谓词外，根据程序的特点，可能还需要定义其它谓词才能写出函数前后条件和循环不变式。以图 1 删除二叉排序树根节点的函数为例，在左右子树都非空时，它通过循环语句找根节点的中序前驱，然后用它取代根节点。需要再定义 2 个谓词：

$$LtAll(Node*\ p, Node*\ q) \triangleq p{==}NULL \vee p{!=}NULL \wedge lt(p{->}data, q) \wedge LtAll(p{->}l, q) \wedge LtAll(p{->}r, q) \quad (5)$$

$$BSTseg(Node*\ p, Node*\ q) \triangleq p{!=}NULL \wedge p{==}q \wedge gt(q{->}data, q{->}l) \wedge BST(q{->}l) \vee p{!=}NULL \wedge p{->}r{!=}NULL \wedge p{!=}q \wedge gt(p{->}data,p{->}l) \wedge BST(p{->}l) \wedge p{->}data < p{->}r{->}data \wedge lt(p{->}data,p{->}r{->}l) \wedge BSTseg(p{->}r, q) \quad (6)$$

谓词 LtAll(p, q)表示 p 树上任何数据都小于 q 树上所有数据。谓词 BSTseg(p, q)表示从 p 指向的节点到 q 指向的节点之间的有序树段所具有的性质。

```
typedef struct node{int data; Node* l; Node* r;}Node;
Node * delete(Node*p) {
    Node*q; Node*s; int n;
    q = p;
    if(p->r==NULL) {                /* 右子树为空，返回 p 的左子树 */
        p = p->l;   free(q);
    } else if(p->l==NULL) {         /* 左子树为空，返回 p 的右子树 */
        p = p->r;   free(q);
    } else {                        /* 左右子树均不空 */
        s = p->l;
        if(s->r == NULL) {          /* 重接*q 的左子树 */
            q->l = s->l;   p->data = s->data;   free(s);
        } else {
            q = s; s = s->r; n = 0;
            while(s->r != NULL) {  /* 在 p 的左子树上向右前进到尽头 */
                q = s; s = s->r; n = n +1;
            }
            p->data = s->data; q->r = s->l;      /* 重接*q 的右子树 */
            free(s);
        }
    }
    return p;
}
```

Fig. 1 Function deleting root node of a binary search tree

图 1 删除二叉排序树根节点的函数

图 1 中函数的前后条件分别是 p!=NULL ∧ BST(p)和 BST(p)，循环不变式是：

BST(p->r) ∧ LtAll(s, p->r) ∧ BSTseg(p->l, q) ∧ lt(q->data, s) ∧ BST(s)

图 2 是该函数的循环不变形状图，其中指向含$\mathcal{P}$的节点的指针是能使谓词(1)为真的指针，灰色节点是 n 个（$n \geq 0$）二叉树节点的浓缩表示[9] ，n 等于循环迭代次数。声明指针 p′代表函数调用的实参。

该函数的循环不变式也可以不用 BSTseg 谓词，而改用全称断言表示如下：

$$\begin{aligned}&\mathrm{BST}(p\text{->}r) \wedge \mathrm{LtAll}(s, p\text{->}r) \wedge \mathrm{lt}(q\text{->}d, s) \wedge \mathrm{BST}(s) \wedge \mathrm{gt}(p\text{->}d, p\text{->}l\text{->}l) \wedge \mathrm{BST}(p\text{->}l\text{->}l) \wedge n >= 0 \wedge\\ &\quad \forall i{:}0..n{-}1.(p\text{->}l(\text{->}r)^{i}\text{->}d < p\text{->}l(\text{->}r)^{i+1}\text{->}d \wedge \mathrm{lt}(p\text{->}l(\text{->}r)^{i}\text{->}d, p\text{->}l(\text{->}r)^{i+1}\text{->}l) \wedge \mathrm{gt}(p\text{->}l(\text{->}r)^{i+1}\text{->}d,\\ &\qquad p\text{->}l(\text{->}r)^{i+1}\text{->}l) \wedge \mathrm{BST}(p\text{->}l(\text{->}r)^{i+1}\text{->}l))\end{aligned}$$

其中 $p(\text{->}r)^{m}$ 代表 p->r...->r，共 m 个“->r”。

2.2 谓词之间的性质引理及其给验证带来的问题

谓词(2)~(6)虽方便了程序员写图 1 函数的前后条件和循环不变式，但也导致 Z3 证明不了该函数的验证条件。因为该函数利用了由这些谓词能归纳证明的二叉排序树的部分性质，例如左子树的最右叶节点是根节点的中序前驱。Z3 没有能力基于这些谓词定义推导出这些归纳性质，从而证明不了验证条件。

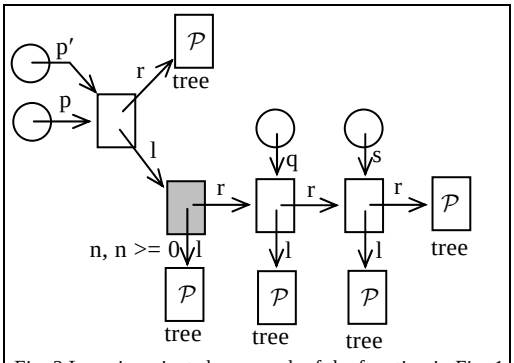

Fig. 2 Loop invariant shape graph of the function in Fig. 1

图 2 图 1 函数的循环不变形状图

本文采用的解决办法是，由程序员提供谓词之间的归纳性质，自动定理证明器把它们当作引理，用于验证条件的证明。

对于图 1 的函数，若提供下面 6 个性质引理，Z3 就能完成验证条件的证明。

$$x < y \wedge lt(y, p) \Rightarrow lt(x, p) \quad (a)$$

$$x > y \wedge gt(y, p) \Rightarrow gt(x, p) \quad (b)$$

$$gt(x, p) \wedge lt(x, q) \Rightarrow LtAll(p, q) \quad (c)$$

$$BSTseg(p, q) \wedge q\text{->}data < q\text{->}r\text{->}data \wedge lt(q\text{->}data, q\text{->}r\text{->}l) \wedge gt(q\text{->}r\text{->}data, q\text{->}r\text{->}l) \wedge BST(q\text{->}r\text{->}l) \Leftrightarrow BSTseg(p, q\text{->}r) \quad (d)$$

$$BSTseg(p, q) \wedge lt(q\text{->}data, q\text{->}r) \wedge BST(q\text{->}r) \Rightarrow BST(p) \quad (e)$$

$$BSTseg(p, q) \wedge q\text{->}data < x \wedge gt(x, q\text{->}r) \Rightarrow gt(x, p) \quad (f)$$

这里以引理(d)为例来解释为何需要这些引理。谓词 BSTseg(p, q)是从右向左（按图 2 形状图的节点次序）归纳定义的，即在 BSTseg(p->r, q)所定义有序树段的基础上把 p 指向的节点从左边加入该树段，得到 BSTseg(p, q)所定义的有序树段。而该函数的循环语句是从左向右地逐步挪动指针，在已遍历有序树段的右边将当前遍历的节点并入该树段，有了引理(d)才可能证明循环出口的验证条件。引理(d)本身需基于谓词 BSTseg(p, q)的定义，通过归纳证明得到。若使用 2.1 节那个带全称断言的循环不变式，无需引理(d)，但引理(e)和(f)需修改。

这种方式给程序员带来的挑战是，为帮助系统完成程序验证，应该提供哪些性质引理。它给系统带来的挑战是，怎样检查程序员所提供的确实是谓词之间的性质引理。

2.3 谓词之间性质引理的证明方法

自定义谓词用来写程序规范，它们的正确性由程序员负责是合情合理的，系统可进行一些简单的检查。而对于程序员提供的归纳性质引理，系统应能证明这些引理，以保证系统验证的结果可靠。

一般而言，不存在算法可自动发现任意归纳定理的归纳证明方式。但对易变数据结构而言，其归纳性质的证明往往都是基于结构归纳，而其结构特征从相应数据类型的定义中可以知道，因此完全可以分析出对一个归纳性质的归纳证明方式。这里介绍本文设计的与易变数据结构有关的性质引理的证明方法，该方法由基于演绎推理的自动定理证明器所支持，可推广到谓词变元是其他数据类型的情况。

该方法分成如下 4 步，前 3 步是分析，最后一步才是证明。

（1）检查归纳谓词定义的合理性和谓词之间的依赖关系

检查归纳谓词定义是否有归纳和非归纳两种分支，是否有最小不动点，并从其定义式的分析中获得其归纳方式。例如谓词(2)~(4)的变元是二叉树指针，它们按二叉树的结构进行归纳。谓词(5)的两个变元都是二叉树指针，很容易分析出它是按第 1 个变元进行结构归纳的。谓词(6)的两个变元也都是二叉树指针类型，它也是按第一个变元进行归纳，但只是按节点的 r 指针进行归纳。因为该谓词把二叉树看成一种特殊的链表结构（见图 2 根节点的 l 指针右边的那部分图）。

还需要分析谓词之间的依赖关系，即一个谓词是基于其它哪些谓词定义的，称依赖者的依赖性高于被依赖者。其目的是防止有相互依赖的谓词定义，以避免下面性质引理证明的困难。这个分析是直截了当的，例如，在谓词定义(2)~(6)中，谓词 BSTseg 依赖于谓词 BST、gt 和 lt。

（2）分析性质引理之间的依赖关系

原则上说，所有的性质引理都可以独立地仅依据谓词定义来证明，但若把其他一些相关性质引理作为证明某个性质引理的前提时，可以方便后者的证明。例如，引理(e)和(f)的证明分别依赖于引理(a)和(b)。一般来说，性质引理之间的依赖关系和这些性质引理所引用的谓词之间的依赖关系是不矛盾的。根据（1）得到的谓词之间的依赖关系，可以确定每个性质引理中依赖性最高的谓词，然后可以据此确定性质引理之间可能存在的依赖关系。只要每个性质引理仅可能依赖比它低的性质引理，则没有相互依赖的性质引理。

这样，在交给自动定理证明器证明某个引理时，一种简单的做法是，把所有谓词定义和该性质引理可能

依赖的其他引理都作为证明该引理的前提。

（3）确定每个性质引理的归纳证明方式

每个性质引理的证明按其中依赖性最高的谓词的归纳方式（从（1）得到），包括按哪个变元和怎样进行归纳。

例如，对于引理(d)，由于它可能依赖于引理(a)、(b)和(c)，因此它的证明按谓词 BSTseg 的第 1 个变元进行归纳。

（4）对每个引理，按下面 3 小步进行证明，直至所有引理得证、否证或证明失败。

- 归纳基始的证明。根据上一步所选谓词的定义，按基始情况将该引理化简，并连同它的前提一起交给自动定理证明器进行证明。若基始情况得证则进入下一小步。

例如，对于引理(e)，就是要证

$$p!=NULL \wedge p==q \wedge gt(q->data, q->l) \wedge BST(q->l) \wedge lt(q->data, q->r) \wedge BST(q->r) \Rightarrow BST(p)$$

- 归纳假设的形成。根据所选谓词及其归纳定义方式，可得归纳假设。

例如，对于引理(e)，归纳假设就是

$$BSTseg(p->r, q) \wedge lt(q->data, q->r) \wedge BST(q->r) \Rightarrow BST(p->r)$$

- 归纳证明。将该引理连同它的前提（包括上述归纳假设）一起交给自动定理证明器进行证明。若得证则进入下一个引理的证明。

3 逻辑变量及其实现技术

断言语言中的逻辑变量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。和引入自定义谓词不同，引入逻辑变量仅给系统带来挑战，主要困难是怎么证明函数调用点的断言蕴涵该函数前条件。

3.1 逻辑变量在程序验证中的作用

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，则使用逻辑变量是一种简单的解决办法。例如，若要表达经过赋值 m = m + 2 后，m 的值增 2，若使用逻辑变量 oldm，则可写成{oldm == m} m = m + 2 {oldm == m – 2}。

逻辑变量的一种典型应用是表达函数参数在函数返回点具有的性质。对于在函数体中可对形参赋值的语言，一般禁止形参出现在函数后条件中。这是因为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点之后的性质。这里以复制单向链表的函数 listcopy(Node* p){ …; return q;} 为例，解释函数前后条件{list(p) ∧ oldp == p}和{ list(q) ∧ list(oldp)} 中逻辑变量 oldp 的作用。若某次调用的实参是 t，它具有性质 list(t)。把函数前条件中的形参 p 代换成实参 t，再把函数前后条件中的逻辑变量 oldp 也代换成 t，则调用点的断言蕴涵前条件 list(t) ∧ t == t，因而调用后有断言 list(q) ∧ list(t)，即实参 t 在调用之后仍然指向单向链表。

逻辑变量的另一种典型应用是表达程序变量在函数调用前后具备的性质。在设计函数的前后条件时，若需要表达其调用点实参以外的某些变量在调用前后的性质变化，则可用逻辑变量来描述。以图 3 二叉排序树的 insert 函数为例，其前后条件都仅是 BST(p)是不够的。因为若如此，则递归调用 insert(p->l, data)之后仅有 BST(p->l)而无 gt(p->data, p->l)，从而不足以推导 BST(p)。引入逻辑变量 y 和 z，把函数前后条件分别增强为

$$BST(p) \wedge y > data \wedge gt(y, p) \wedge z < data \wedge lt(z, p) \text{ 和 } BST(p) \wedge gt(y, p) \wedge lt(z, p)$$

由于在调用 insert(p->l, data)之前有 p->data > data 和 gt(p->data, p->l)，它们能够与函数前条件中的 y > data 和 gt(y, p->l)匹配（匹配代换是[p->data/y]），因而调用之后有 BST(p->l)和 gt(p->data, p->l)。逻辑变量在本例体现的作用是：调用语句之前程序点的状态满足越多的性质，则其后程序点的状态也满足越多的性质。

```
typedef struct node{int data; Node* l; Node* r;}Node;
Node* insert(Node* p, int data) {
    if ( p == NULL) {
        p = malloc(Node);    p->l = NULL; p->r = NULL; p->data = data;
    } else if (p->data > data) {
        p->l = insert(p->l, data);
    } else if (p->data < data) {
        p->r = insert(p->r, data);
    }
    return p;
}
```

Fig. 3 Insertion function of binary search tree

图 3 二叉排序树的插入函数

3.2 逻辑变量给验证带来的问题

引入逻辑变量给系统带来的挑战是，怎么证明函数调用点的断言蕴涵该函数的前条件。在此通过实例分析来介绍其中的主要问题。

首先，若函数前条件中的逻辑变量存在多种代换方式，它们都能使函数调用点的断言蕴涵函数前条件，选择哪种代换。

以图 3 的 insert 函数为例，在递归调用语句 p->l = insert(p->l, data)之前的程序点，其完整的断言是

$$\begin{aligned}&\mathrm{BST}(p\text{->}l) \wedge \mathrm{BST}(p\text{->}r) \wedge \mathrm{gt}(p\text{->}data, p\text{->}l) \wedge \mathrm{lt}(p\text{->}data, p\text{->}r) \wedge p\text{->}data > data \wedge y > data \wedge \\ &\quad y > p\text{->}data \wedge \mathrm{gt}(y, p\text{->}l) \wedge \mathrm{gt}(y, p\text{->}r) \wedge z < data \wedge z < p\text{->}data \wedge \mathrm{lt}(z, p\text{->}l) \wedge \mathrm{lt}(z, p\text{->}r)\end{aligned}$$

由于这是递归调用，为便于区分，将被调用者前后断言中的逻辑变量加撇号以示区别。若对函数前条件进行[p->data/y′, z/z′]代换，则上述断言能蕴涵 insert 函数的前条件。若进行[y/y′, z/z′]代换，则上述断言也能蕴涵 insert 函数的前条件。

判断哪种代换方式对继续往下的验证有用是困难的。按最强后条件演算方式，应该把这样的代换都记录下来。事实上，对于图 3 的 insert 函数，这两个代换对于往下的验证都是需要的。

其次，函数前条件中有涉及多个逻辑变量的基本断言，例如逻辑变量之间的关系断言，这时如何证明函数调用点的断言蕴涵函数前条件。

以消除 AA 树左水平连接（即进行右旋）的 skew 函数为例（见图 4）。AA 树是一种平衡的二叉排序树。非空 AA 树的节点层次存放在节点的 level 域中，而空树的层次为 0。AA 树的平衡谓词 balance 定义如下：

$$\begin{aligned}&\mathrm{balance}(\mathrm{Node}^*\ p, \mathrm{int}\ m) \triangleq p == \mathrm{NULL} \wedge m == 0 \vee \\ &\quad p != \mathrm{NULL} \wedge p\text{->}level == m \wedge \mathrm{balance}(p\text{->}l, m-1) \wedge \mathrm{balance}(p\text{->}r, m-1) \vee \\ &\quad p != \mathrm{NULL} \wedge p\text{->}r != \mathrm{NULL} \wedge p\text{->}level == m \wedge p\text{->}r\text{->}level == m \wedge \\ &\qquad \mathrm{balance}(p\text{->}l, m-1) \wedge \mathrm{balance}(p\text{->}r\text{->}l, m-1) \wedge \mathrm{balance}(p\text{->}r\text{->}r, m-1)\end{aligned} \tag{7}$$

根据 skew 函数的代码和 balance 的定义，skew 函数前后条件的一种情况分别是：

$$\begin{aligned}&t != \mathrm{NULL} \wedge t\text{->}l != \mathrm{NULL} \wedge \mathrm{BST}(t) \wedge \mathrm{gt}(y, t) \wedge \mathrm{lt}(z, t) \wedge t\text{->}level == m \wedge \\ &\quad t\text{->}l\text{->}level == m \wedge \mathrm{balance}(t\text{->}l\text{->}l, i) \wedge \mathrm{balance}(t\text{->}l\text{->}r, j) \wedge t\text{->}r\text{->}level == m \wedge \\ &\qquad \mathrm{balance}(t\text{->}r\text{->}l, k) \wedge \mathrm{balance}\ (t\text{->}r\text{->}r, n) \wedge \\ &\qquad\quad m - i == 1 \wedge m - j == 1 \wedge m - k == 1 \wedge m - n == 1\end{aligned}$$

和

t != NULL ∧ t->r != NULL ∧ BST(t) ∧ gt(y, t) ∧ lt(z, t) ∧ t->level == m ∧
t->r->level == m ∧ t->r->r->level == m ∧ balance(t->l, i) ∧ balance(t->r->l, j) ∧
balance(t->r->r->l, k) ∧ balance(t->r->r->r, n) ∧
m – i == 1 ∧ m – j == 1 ∧ m – k == 1 ∧ m – n == 1

它们的第 4 行都是逻辑变量之间的关系断言。

显然，在证明函数调用点的断言蕴涵函数前条件时，需要检查对逻辑变量所做代换能否使函数前条件中逻辑变量的关系断言成立。

```
typedef struct node{int d; int level; Node * l; Node * r;}Node;
Node* skew (Node* t) {
    Node* p;
    /* 交换水平左链的指针 */
    p = t->l; t->l = p->r; p->r = t; t = p; return t;
}
```

Fig. 4 Function for removing a left horizontal link

图 4 消除左水平连接的函数

3.3 函数调用点的断言蕴涵函数前条件的证明方法

在允许使用逻辑变量的情况下，若函数 *f* 的形参向量和前后条件中的逻辑变量向量分别是 $\vec{p}$ 和 $\vec{x}$，则函数调用规则是：

$$\frac{\{Q\}f(\vec{p})\{R\} \qquad P \Rightarrow Q[\vec{e_1}/\vec{p}][\vec{e_2}/\vec{x}]}{\{P\}v = f(\vec{e_1})\{R[\vec{e_2}/\vec{x}][v/result]\}}$$

其中 result 表示存放函数值的变量。基于该规则，若函数调用点的前条件是 *P*，且有适当的代换 $\vec{e_2}/\vec{x}$（$\vec{e_2}$ 是逻辑变量向量 $\vec{x}$ 的代换向量），使得 $P \Rightarrow Q[\vec{e_1}/\vec{p}][\vec{e_2}/\vec{x}]$，那么就可得到调用点后条件。

把上述蕴涵式改成 $P \Rightarrow \exists \vec{x}.Q[\vec{e_1}/\vec{p}]$ 并直接交给 Z3，由 Z3 寻找对 $\vec{x}$ 的代换并完成证明，这种方式有两点困难。首先，验证条件中出现存在量词时，Z3 不一定能证。其次，即使能证，Z3 也只会告知对 $\vec{x}$ 所进行的一种代换，但提供一种代换没有解决问题，因为从先前的分析知道，所有可能的代换都要记录下来。

本文所设计的方法是在验证条件生成过程中寻找合适的代换 $\vec{e_2}/\vec{x}$，然后把 $P \Rightarrow Q[\vec{e_1}/\vec{p}][\vec{e_2}/\vec{x}]$ 交由 Z3 去证明。在介绍该方法前，需要先有下面 4 点约定。

（1）已经将函数前条件中的形参代换成相应的实参表达式。若是递归调用，已对函数前后断言中的逻辑变量进行了必要的改名，以避免和调用点断言有名字冲突。

（2）函数调用点的前断言 P、被调用函数前后条件 Q 和 R 分别是析取范式 $P_1 \vee \ldots \vee P_m$、$Q_1 \vee \ldots \vee Q_n$ 和 $R_1 \vee \ldots \vee R_k$。假定出现在各个 Q_i（$1 \le i \le n$）中的逻辑变量的集合是一样的。还需假定对每个 P_i（$1 \le i \le m$），若存在断言 p 使得 $P_i \Rightarrow p$，则 p 已经是 P_i 中的一个子句。举个简单例子，若调用点有 data != t->data ∧ data >= t->data，函数前条件有 data > z，其中 z 是逻辑变量。若不从 data != t->data ∧ data >= t->data 推出 data > t->data，则下面证明方法的第(3)步难以为逻辑变量 z 找到匹配代换。还有，每个 P_i 无需在证明过程中展开谓词，变成 $P_{i,1} \vee P_{i,2}$。

（3）对每个 Q_i（$1 \le i \le n$），其各子句可根据含逻辑变量的情况分成如下三类（各类都可能为空）：

- 不含逻辑变量的子句的合取 $Q_{i,1}$。
- 含一个逻辑变量的子句的合取 $Q_{i,2}$。限定这样的逻辑变量直接作为谓词变元或关系运算的对象，即不出现 gt(x+2, p)这样的情况。这个限定是合理的，因为通过增加新逻辑变量 y 并把 gt(x+2, p)变换成 gt(y, p) ∧ y

== x+2 即可满足限定，其中 y == x+2 可归属下一类断言。

• 至少出现两个逻辑变量的子句（逻辑变量的关系断言，如 y == x+2）的合取 $Q_{i,3}$。限定这里的逻辑变量都已出现在 $Q_{i,2}$ 中。

（4）还需要通过例子来非形式地引入对断言进行逻辑变量代换的运算。若有断言 y > data ∧ gt(y, p) ∧ z < data ∧ lt(z, p)，对该断言进行逻辑变量 y 和 z 的 3 种代换[a/y, c/z]、[a/y, /z]和[/y, /z]的结果如下（/y 表示不对 y 进行代换）：

• (y > data ∧ gt(y, p) ∧ z < data ∧ lt(z, p))[a/y, c/z] = a > data ∧ gt(a, p) ∧ c < data ∧ lt(c, p)

• (y > data ∧ gt(y, p) ∧ z < data ∧ lt(z, p))[a/y, /z] = a > data ∧ gt(a, p)

• (y > data ∧ gt(y, p) ∧ z < data ∧ lt(z, p))[/y, /z] = true

这种运算的一个重要特点是：若只代换其中部分逻辑变量，则剩余逻辑变量的断言被略去。代换[a/y, c/z]和[a/y, /z]相比，前者更一般，因为它们的 y 代换相同，前者有 z 代换，而后者没有。同样，代换[a/y, c/z]和[a/y, /z]比代换[/y, /z]更一般。[a/y, c/z]和[b/y, /z]之间不能比较，因为它们对 y 的代换不一样。

在上面 4 点约定下，所设计的证明方法概述如下（它包括了对框架规则（frame rule）的使用）。

(1) 若 Q_1 中出现逻辑变量，则执行(2)。否则：

将 $P \Rightarrow Q$ 交给自动定理证明器（假定证明器回答真或假）。若结果为假，则证明结束；否则令 $A_j = R$（$1 \le j \le m$），执行(5)。

(2) 对每个 P_j（$1 \le j \le m$）执行步骤(3)~(5)。

(3) 令代换集合 S 为空。然后依次对 $Q_{1,2}, \ldots, Q_{n,2}$ 中的每个 $Q_{i,2}$（$1 \le i \le n$），执行下面的操作：

对 $Q_{i,2}$，寻找未出现在 S 中并且是 $Q_{i,2}$ 最一般的代换$[\vec{e}/\vec{x}]$，使得 P_j 能蕴涵 $Q_{i,2}[\vec{e}/\vec{x}]$中的所有断言。若 $P_j \Rightarrow Q_i[\vec{e}/\vec{x}]$得证，则将该代换加入 S。

注意，若对 $Q_{i,2}$ 存在多个未出现在 S 中并且是最一般的代换，则对它们都要进行上述操作。

(4) 若 S 仍为空，意味着找不到合适的代换使得 P_j 蕴涵函数前条件，则证明结束；否则：

用 S 中所有 *l* 个代换$[\vec{e}_1/\vec{x}], \ldots, [\vec{e}_l/\vec{x}]$对函数后条件 $R_1 \vee \ldots \vee R_k$ 进行逻辑变量代换，令

$$A_j = (R_1[\vec{e}_1/\vec{x}] \wedge \ldots \wedge R_1[\vec{e}_l/\vec{x}]) \vee \ldots \vee (R_k[\vec{e}_1/\vec{x}] \wedge \ldots \wedge R_k[\vec{e}_l/\vec{x}])。$$

不含逻辑变量的断言在 $R_i[\vec{e}_1/\vec{x}] \wedge \ldots \wedge R_i[\vec{e}_l/\vec{x}]$（$1 \le i \le k$）中会重复，可以删除重复出现的部分。

(5) 将 P_j 中不能保持到调用点之后的断言删除，得到 P_j'。其做法与未引入逻辑变量的情况一致。

(6) 调用点之后的断言是$(P_1' \wedge A_1 \vee \ldots \vee P_m' \wedge A_m)[v/result]$（$1 \le j \le m$）。

例如，语句 p->l = insert(p->l, data)的前断言 P 只有一种情况（见 3.2 节），其中能保持到调用点之后的断言 P′是

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

后断言 R 经过代换得到的 A 是

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

因为对 y′有 2 个最一般的代换。

调用点之后完整断言是上述两部分断言的合取。

4 系统

我们所研发的程序验证系统原型[11, 12]的流程分成下面三步。

1、**预处理阶段** 该阶段为源代码生成抽象语法树并完成通常的静态检查。

2、**形状分析阶段** 该阶段遍历语法树，基于形状图逻辑[9,10]生成各程序点的形状图。它有如下 3 个特点。

（1）在需要形状检查的程序点进行形状推断和形状检查，以帮助排除源程序中的错误。

（2）对循环语句需要遍历多次，以推断循环不变形状图。

（3）对递归函数也需要遍历多次，以推断函数后条件中的形状图。

3、**程序验证阶段**　该阶段在逻辑上可分成验证条件生成和自动定理证明两个子阶段。验证条件生成子阶段遍历语法树，根据程序员提供的有关非指针型数据的函数前后条件和循环不变式，基于形状图逻辑，按最强后条件演算方式生成验证条件。验证条件的一般形式是 $G, T \triangleright Q \Rightarrow Q'$，其中 G 是产生验证条件那个程序点的形状图，T 是程序员提供的谓词定义和性质引理，它们是 $Q \Rightarrow Q'$的证明环境，Q 和 Q'是符号断言。

自动定理证明阶段把 G 上指针相等信息转换为符号断言，称为 P。P 中可能还包含一些其他信息。例如，若 G 中包括浓缩节点，则约束其节点个数的表达式 e 的断言 a 也包括在 P 中。然后将$\neg(P \wedge T \wedge Q \Rightarrow Q')$交给可满足性模理论求解器 Z3。若$\neg(P \wedge T \wedge Q \Rightarrow Q')$不可满足，则 $P \wedge T \wedge Q \Rightarrow Q'$得证。

该原型可以验证易变数据结构上较为复杂的程序，表 1 给出部分程序的代码规模以及在 Windows 7 PC，Intel Core i5-2400 3.1GHz CPU 和 4G 内存上的运行时间和空间等的统计数据。这些程序大部分包括了插入和删除函数，以及被它们调用的函数。操作二叉平衡树的程序的函数最多，因为除了插入和删除函数外，还有左旋、右旋、左平衡和右平衡等函数。二叉排序树和伸展树的谓词和性质引理相对较多，因为这两个程序采用循环而非递归算法，需要额外有关二叉树上有序树段的谓词定义（例如 2.1 节的谓词定义(6)）以及该谓词的一些性质（例如 2.2 节的性质引理(d)~(f)）。

Table 1 Statistical data about shape graphs and verification conditions

表 1　有关形状图和验证条件的统计数据

数据结构	函数(个)	逻辑变量(个)	谓词(个)	性质定理(条)	验证条件(个)	循环(个)	验证时间(ms)
有序单向链表：插入、倒置、合并	3	0	7	5	11	4	5,942
有序双向链表：倒置	1	0	4	4	5	2	11,790
有序循环双向链表：插入	1	1	0	0	3	1	539
二叉排序树：递归和非递归插入、删除	4	3	8	21	15	3	18,68
AA 树：插入	3	7	6	6	8	0	24,274
二叉平衡树：插入、删除	8	15	5	4	32	0	96,556
树堆：插入、删除	7	3	5	6	18	0	5,912
伸展树：插入、删除	6	5	5	9	16	2	122,424

下面以图 5 的 AA 树插入函数为例，进一步展示逻辑变量在程序验证中的作用。该函数的前后条件分别如下：

t == NULL ∧ BST(t) ∧ balance(t, m) ∧ m == 0 ∧ y > data ∧ gt(y, t) ∧ z < data ∧ lt(z, t) ∨
t != NULL ∧ BST(t) ∧ t->level == m ∧ balance(t->l, k) ∧ balance(t->r, n) ∧
n == m-1 ∧ k == n ∧ y > data ∧ gt(y, t) ∧ z < data ∧ lt(z, t) ∨
t != NULL ∧ BST(t) ∧ t->level == m ∧ t->r->level == n ∧ balance(t->l, k) ∧ balance(t->r->l, i) ∧
balance(t->r->r, j) ∧ k == m-1 ∧ n == m ∧ k == i ∧ k == j ∧ y > data ∧ gt(y, t) ∧ z < data ∧ lt(z, t)

和

t != NULL ∧ BST(t) ∧ t->level == m +1 ∧ balance(t->r, m) ∧ balance(t->l, m) ∧ n == m ∧
gt(y, t) ∧ lt(z, t) ∨
t != NULL ∧ BST(t) ∧ t->level == m +1 ∧ balance(t->r, m) ∧ balance(t->l, m) ∧ m == 0 ∧
gt(y, t) ∧ lt(z, t) ∨
t != NULL ∧ BST(t) ∧ balance(t, m) ∧ gt(y, t) ∧ lt(z, t)

其中用到谓词定义(2), (3), (4)和(7)，用到逻辑变量 m, n, i, j, k, y 和 z。验证过程中需要用到的性质引理是(a)和(b)。

上述前条件实际上是 balance 谓词的三种展开形式，为什么不把它们折叠起来，让

BST(t) ∧ balance(t, m) ∧ y > data ∧ gt(y, t) ∧ z < data ∧ lt(z, t)

作为前条件。同样，为什么不合并后条件的前两种情况，写成

t != NULL ∧ BST(t) ∧ balance(t->r, m+1) ∧ gt(y, t) ∧ lt(z, t) ∨

t != NULL ∧ BST(t) ∧ balance(t->r, m) ∧ gt(y, t) ∧ lt(z, t)

这是因为前后条件需要表达出 AA 树插入操作的一个性质：在 AA 树插入一个节点后，若根节点的 level 增加，则插入前的 AA 树一定符合 AA 树的第 1 或第 3 种形式，而不符合第 2 种形式，即

- 空树，或者
- 非空，右子树也非空，且 t->r->level == t->level。

函数后条件中前两种情况出现的逻辑变量等式断言 m == n 和 m == 0 就是用来刻画 AA 树的这个性质，若没有它们则证明不出插入函数保平衡性。

AA 插入函数调用的 split 函数是用来消除连续的右水平连接，它类似于 skew 函数。

```
Node* AAinsert (Node* t, int data) {
    if (t == NULL) {
        t = malloc(Node); t->l = NULL; t->r = NULL; t->d = data; t->level = 1;
    } else if (data < t->d) {
        t->l = AAinsert(t->l, data);
        if (t->l->level == t->level) {t = skew(t);}
        if (t->r->r != NULL && t->r->r->level == t->level) {t = split(t);}
    } else if (data > t->d) {
        t->r = AAinsert(t->r, data);
        if (t->r->r != NULL && t->r->r->level == t->level) {t = split(t);}
    }
    return t;
}
```

Fig 5 Insert function of AA tree

图 5 AA 树的插入函数

5 相关研究工作比较

在 Hoare 三元式{P} S {Q}中，P 或 Q 中会出现带量词的断言，如∀i:1..n-1.a[i-1]<a[i]，其中的约束变元 i 也称为逻辑变量，是显式地受全称量词约束的逻辑变量。本文讨论的逻辑变量是可以同时出现在 P 和 Q 中自由逻辑变量。它们在外围的三元式逻辑层次上隐式地受全称量词约束的逻辑变量。为区分这两种情况，后者经常被称为辅助变量[14, 15, 16]，但也有研究人员仍称其为逻辑变量[17, 18]。逻辑变量还有另一个容易混淆的名字是幽灵（ghost）变量，因为幽灵变量在一些文献[5, 19, 20]中另有含义。在这些文献中，幽灵变量可被赋值，但它们不出现在程序的可执行代码中，而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。由此可见，幽灵变量与逻辑变量不是一回事，因为逻辑变量仅出现在断言中。引入幽灵语句和幽灵变量也是为了便于写程序规范。例如，程序中并非一定需要记住循环迭代计算次数的程序变量，但为方便写涉及迭代次数的循环不变式，可以用幽灵变量及对其赋值的幽灵语句来记住循环迭代次数。例如 2.1 节图 1 的程序中的局部变量 n 仅用于写那个带全称量词的循环不变式，可以用幽灵变量来取代它。幽灵变量和逻辑变量一致之处是，它们都不能被程序访问，但它们不能相互取代。幽灵变量用于表达程序代码本身的性质，而逻辑变量用于表达程序代码对上下文的不变性质。

在断言语言中引入逻辑变量后，Hoare 逻辑不再是完备的。典型的例子是推论规则不完备。例如，若 x 和 X 分别是程序变量和逻辑变量，{X == x} S {X == x}和{X == x+1} S {X == x+1}这两个三元式都表示 S 没有改变 x 的值。但是，它们两者不能通过 Hoare 逻辑的推论规则来相互推导。Kleymann[14]通过把对逻辑变量的代换引入推论规则，解决了这个问题。

本文把 Kleymann[14]的无参函数的函数调用规则扩展到有参数场合，即 3.3 节给出的规则。基于该规则，3.3 节设计了函数调用点的断言蕴涵函数前条件的证明方法。该方法在证明前先对函数前条件中的逻辑变量进行匹配代换，并且证明成功后得到函数最强后条件。“最强”的体现之一是，若对逻辑向量有多种最一般的代换，则把这多种代换的结果都放在函数后条件中。

近年来有关程序验证的一些研究工作[16, 17, 18]都使用逻辑变量来描述程序规范，但它们都没有像本文这样系统地讨论逻辑变量的使用和实现技术。

VeriFast 是一个可用于单线程和多线程的 C 和 Java 程序验证工具的原型[21]。为提高程序规范的表达能力，程序员可以定义归纳数据类型及其上的递归函数和分离逻辑谓词。为便于验证用它们所描述的程序规范，程序员可以进一步编写用于证明他们定义的归纳数据类型、递归函数和分离逻辑谓词之间性质的引理函数。和 VeriFast 相比，本文的系统原型未允许程序员定义数据结构上的递归函数，因为它们意欲定义的数据结构的一些简单属性可以在断言中隐式地表达而免去定义递归函数。在 AA 树的平衡谓词 balance(p, m)中，m 就是节点层次。若定义如下的节点层次函数：

lev(Node* p) ≜ if p == NULL then 0 else p->level

并把 3.2 节的 balance(p, m)谓词修改如下：

balance(p) ≜ p == NULL ∧ lev(t) == 0 ∨
p != NULL ∧ p->level == lev(p) ∧ balance(p->l) ∧ lev(p->l) == lev(p) – 1 ∧
balance(p->r) ∧ lev(p->r) == lev(p) – 1 ∨
p != NULL ∧ p->r != NULL ∧ p->level == lev(p) ∧ p->r->level == p->level ∧
balance(p->l) ∧ lev(p->l) == lev(p) – 1 ∧ balance(p->r->l) ∧
lev(p->r->l) == lev(p) – 1∧ balance(p->r->r) ∧ lev(p->r->r) == lev(p) – 1

则 3.2 节所列的 skew 函数前条件无需逻辑变量。

VeriFast 与本文的主要区别在于谓词之间性质的表达和证明。VeriFast 用引理函数的前后断言来表达归纳数据类型、递归函数和分离逻辑谓词之间的性质，函数本身是其前后断言的一个证明，而函数调用是该性质的一个应用。其中最重要的区别是，在 VeriFast 中性质的证明由程序员提供，而本文方法则自行发现性质引理的归纳证明方式，然后按此逐步交给 Z3 完成证明，因而免去程序员提供性质引理的证明。对程序员来说，本文方法显得简洁直观，但是仅仅面向易变数据结构的归纳性质的自动证明是本文的局限。Dafny 是一个面向程序功能验证的自动验证器[20]，它的最新实现[22]类似本文方法，它比 VeriFast 所用方法前进一步，相当于能够自动地为引理函数发现归纳证明方式，因而不必程序员提供引理函数的证明。

一些验证器原型[23, 24]允许程序员使用自定义谓词，但未进一步要求程序员提供谓词之间的性质引理。这些验证器的一个共同特点是将程序限制到只能用递归而不能用循环迭代，也未见它们有处理双向循环链表的例子。的确，本文性质引理(d)~(f)是由于图 1 程序使用循环语句引起的（性质引理(a)~(c)在断言语言允许集合类型的情况下可以免去）。禁止使用循环语句对程序员来说是难以接受的，让程序员提供谓词之间的性质引理或许是一种可接受的解决办法。

6 总结

在基于逻辑推理的程序验证方法中，断言语言的能力影响着系统的实用性和系统实现的难易性, 断言语言的设计受制于自动定理证明器的能力。专门讨论断言语言设计的文献很少，本文是断言语言设计中的部分经验和总结。就本文所讨论的内容，我们拟从下面两个方面进行改进。

（1）增加用户自定义函数。虽然很多场合可避免引入自定义函数，但使用自定义函数有可能减少逻辑变量的使用。自定义函数所允许的运算必须有所限制，以避免给验证条件的证明带来困难。

（2）增加幽灵变量。幽灵变量和逻辑变量不能相互取代。使用幽灵变量和幽灵语句，可以避免程序中

出现仅与验证有关但并不影响程序结果的变量和语句。幽灵变量的实现并不困难。

References:

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

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

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

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

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

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

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

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

[9] Zhao-Peng Li, Yu Zhang, and Yi-Yun Chen. A shape graph logic and a shape system. *Journal of Computer Science and Technology*. 28(6):1063-1084, 2013.11.

[10] Yu Zhang, Zhao-Peng Li, and Yi-Yun Chen. Theorem Proving for Theory of Shape Graphs. 已投 *Journal of Computer Science and Technology*.中文版见 http://staff.ustc.edu.cn/ ~yiyun.

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

[12] Z.P. Li, Y. Zhang, Y.Y. Chen, Y.H. Song, and J.C. Meng. The verifier prototype system of PointerC based on the Shape Graph Logic and Shape System. URL: http://kyhcs.ustcsz.edu.cn/SGL, Apr. 2013.

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

[14] T. Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11:541-566, 1999. 12.

[15] Maria João Frade and Jorge Sousa Pinto. Verification conditions for source-level imperative programs. *Computer Science Review* 5(3): 252-277, 2011.

[16] David von Oheimb and Tobias Nipkow. Hoare Logic for NanoJava: Auxiliary Variables, Side Effects and Virtual Methods Revisited. In *Proc. Formal Methods in Europe(FME)*, volume 2391 of LNCS, pages 89–105, 2002.

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

[18] Matthew J. Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as Resource in Hoare Logics. In

Proceedings of LICS, pages 137-146, 2006.

[19] Martin Hofmann and Mariela Pavlova. Elimination of Ghost Variables in Program Logics. In *Proc. Trustworthy Global Computing*, *LNCS 4912*, pages 1-20, 2007.

[20] K. Rustan and M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR-16*, *LNCS 6355*, pages 348-370, 2010.

[21] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*, pages 41-55, 2011.

[22] K. Rustan and M. Leino. Automating Induction with an SMT Solver. In *VMCAI 2012*, *LNCS 7148*, pages 315-331, 2012.

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

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

附中文参考文献：

[11] 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.