

# 一种用于指针程序安全性证明的指针逻辑

陈意云 华保健 葛琳 王志芳

(中国科学技术大学计算机科学与技术系 合肥 230026)

(中国科学技术大学苏州研究院软件安全实验室 江苏 苏州 215123)

**摘 要** 在高可信软件的各种性质中,安全性是被关注的重点,其中软件满足安全策略的证明方法是研究的热点之一.文中根据作者所设想的安全程序的设计和证明框架,为类 C 语言的一个子集设计了一个指针逻辑系统.该逻辑系统是 Hoare 逻辑系统的一种扩展,它用推理规则来表达每一种语句引起指针信息的变化情况.它可用来对指针程序进行精确的指针分析,所获得的信息用来证明指针程序是否满足定型规则的附加条件,以支持程序的安全性验证.该逻辑系统也可用来证明指针程序的其它性质.

**关键词** 软件安全;指针逻辑;Hoare 逻辑;指针分析;类型系统

中图法分类号 TP301

## A Pointer Logic for Safety Verification of Pointer Programs

CHEN Yi-Yun HUA Bao-Jian GE Lin WANG Zhi-Fang

(Department of Computer Science, University of Science and Technology of China, Hefei 230026)

(Software Security Laboratory, Suzhou Institute for Advanced Study,

University of Science and Technology of China, Suzhou, Jiangsu 215123)

**Abstract** Safety is an important issue among the properties of high-assurance software and developing the verification methods for software to meet safety policies is one of the hot research. In terms of the authors' sketch of design and verification of safety programs, a pointer logic system is designed for a subset of C-like language. This logic system is an extension of Hoare logic system and inference rules are designed to express the modification of pointer information for every kind of statements. It can be used for accurate pointer analysis of pointer programs. The information from the analysis can be used to verify if pointer programs satisfy the side conditions of typing rules and then support safety verification for programs. The logic system can also be used to verify other properties of pointer programs.

**Keywords** software safety; pointer logic; Hoare logic; pointer analysis; type system

## 1 引 言

在高可信的各种要求中,安全性(包括 safety 和 security)是关注的重点. Safety 是指软件运行时不

引起危险、灾难的能力,而 security 是指软件系统对数据和信息提供保密性、完整性、可用性、真实性保障的能力.本文所讲的安全性主要是指 safety,但是软件的 safety 和 security 是有联系的,黑客通常就是利用缓冲区溢出、数组访问越界、悬空指针访问等

低级的 safety 错误,来破坏系统和获取未经授权的控制等.因此提高 safety 有助于保证 security.

程序性质证明(而不是传统的程序正确性证明)领域近十年来有了很大的发展,许多学者提出了不同的思路,这些思路主要采取基于类型的或基于逻辑的方法,用于高级语言程序或低级语言程序的性质证明.基于类型方法的典型研究有类型化汇编语言(Typed Assembly Language)<sup>[1]</sup>和类型细化(type refinement)理论<sup>[2]</sup>的研究.基于逻辑方法的典型研究有携带证明的代码(Proof-Carrying Code,PCC)<sup>[3]</sup>和 FPCC(Foundational Proof-Carrying Code)框架<sup>[4]</sup>.Shao 的携带证明汇编编程项目 CAP(Certified Assembly Programming)<sup>[5]</sup>和基于栈的 CAP(SCAP)<sup>[6]</sup>是典型的基于逻辑的研究项目.基于逻辑的方法和基于类型论的方法有很大的互补性,近年来出现了一些结合这两种方法的研究.一种结合两者的研究是 Xi 等进行的 ATS(Applied Type System)项目的研究<sup>[7]</sup>,他们扩展类型系统,将程序状态引入类型系统,依靠 ATS 与 Hoare 逻辑的相似性,以 ATS 来编码 Hoare 逻辑,从而可以在他们的类型系统上模拟 Hoare 逻辑的推理.

基于国际上这些研究,我们认为,对于那些有高安全性要求的软件,程序设计和证明的一种新方式将是:

- (1) 程序设计者将软件的安全策略等描述成程序应满足的规范,连同程序一起提交给编译器;
- (2) 编译器生成为证明程序满足规范所需的验证条件,并且利用内嵌的定理证明器自动地或交互地证明这些验证条件;
- (3) 编译器在把源程序翻译成目标代码的同时,将源程序满足规范的证明翻译成目标代码满足等效规范的证明,这样的编译器称为出具证明的编译器(certifying compiler);
- (4) 在目标代码一级由证明检验器利用代码所携带的证明自动进行代码满足规范的检验.

该框架的优点是,它向程序设计者提供源级而不是目标级的程序性质证明方法,以提高安全程序的开发效率,同时它将编译器、证明器等排除出受信任的计算基础(Trusted Computing Base,TCB),以尽量缩小系统的 TCB.

本文介绍我们在这个框架的初步实现中,为类 C 语言的一个子集 PointerC 设计的一个指针逻辑系统,它是 Hoare 逻辑的一种扩展,本质上是一种精确的指针分析(pointer analysis)工具.它可用来

从前向后收集各指针是 NULL 指针、悬空指针(dangling pointer)还是有效指针(有指向对象的指针)的信息,收集各有效指针之间相等与否的信息.所收集信息用来证明指针程序是否满足定型规则(typing rule)的附加条件,以支持对指针程序的安全性验证及其它性质的验证.

本文第 2 节介绍有关指针安全的一些基本概念;第 3 节是指针逻辑的设计;第 4 节给出一个证明实例;第 5 节是相关工作比较;第 6 节是总结.

## 2 基本知识

首先介绍 PointerC 在指针运算方面的限制.在 PointerC 中,指针类型的变量只能用于赋值、相等和不相等比较、存取指向对象等运算以及作为函数(包括 free)的参数,指针算术和取地址运算(&)被禁止.malloc 和 free 被看成是 PointerC 预定义的函数,并且满足安全程序的最基本要求.例如 malloc 任何一次调用都能成功并且所分配空间与尚未释放空间无任何重叠.

上述限制的目的是为了便于静态检查程序的安全性.程序运行时出现对 NULL 指针或悬空指针进行存取指向对象的操作、把 NULL 指针或悬空指针作为 free 函数调用的实在参数、发生内存泄漏等都被认为不满足基本安全策略(类型安全和内存安全等).该语言定型规则中的附加条件就是用来禁止这些情况的出现,本文指针逻辑的用途之一就是用来完成对这些附加条件的静态检查.

下面明确本文有关指针类型的一些术语和约定.程序中显式声明的变量称为声明变量,由 malloc 函数显式和动态分配的空间称为动态对象.在程序中,动态对象的域只能通过指针类型的声明变量来访问,如  $s \rightarrow data$  和  $s \rightarrow next \rightarrow prior$  等,这种把脱引用(dereference)和域访问等组合的语法表达式称为相应声明变量或动态对象域的访问路径,它是一个语法概念,是变量的名字.注意,若  $s$  是 NULL 指针或悬空指针时, $s \rightarrow next$ , $s \rightarrow data$  等在本文中都不看成访问路径.下面用  $p$ , $q$  和  $r$  作为代表一般访问路径的元变量,它们最简单的情况就是声明变量的名字.若访问路径  $p$  的后面并置一个非空字符串后形成访问路径  $q$ ,则称  $p$  是  $q$  的前缀.在用此定义时,需要把  $*p$  这种语法形式看成  $p*$  的形式.为方便起见,对访问路径中重复出现的部分使用缩写表示,如  $s(\rightarrow next)^i$  用来表示  $s \rightarrow next \rightarrow next \cdots \rightarrow$

$next$  (其中  $\rightarrow next$  出现  $i$  次), 若  $i=0$ , 则  $s(\rightarrow next)^i$  就表示  $s$ .

各种类型的指针变量(包括动态对象中的指针类型的域)都简称为指针, NULL 指针和悬空指针统称为无效指针, 有指向对象的指针称为有效指针(effective pointer). 区分 NULL 指针和悬空指针是由程序通过判断指针是否等于 NULL 来区别的. 访问路径为  $p$  和  $q$  的两个有效指针相等时, 访问路径  $*p$  和  $*q$  (或  $p \rightarrow next$  和  $q \rightarrow next$  等) 互为别名(alias). 由于 PointerC 对指针运算的限制, 再加上函数的参数都是传值方式, 一个声明变量的名字不会和其它变量的名字互为别名(本文没有讨论数组元素的动态别名问题); 当两个有效指针的值相等时, 在代表它们的访问路径上添加公共后缀后, 所得两条访问路径形成别名. 显然, 若能掌握有效指针相等与否的信息, 就能判断两条访问路径是否互为别名并且帮助选择访问路径的别名.

### 3 指针逻辑的设计

为证明程序满足基本安全策略, 除了要为 PointerC 设计一个类型系统外, 还需要设计一个证明系统. 因为该类型系统的某些定型规则中有附加条件, 例如, 下标表达式不能越界,  $s \rightarrow next$  必须是一条访问路径等, 它们不能由通常的类型系统来检查, 本文采用一个证明系统来证明这些附加条件.

我们可通过对 Hoare 逻辑的扩展来设计这样一个证明系统, 因为我们在目标语言级采用 CAP 方式. CAP 证明目标程序的性质所采用的办法是: 把 Hoare 逻辑的方法直接绑定到目标机器的操作语义上<sup>[6-7]</sup>. 我们在源语言级使用 Hoare 逻辑方式有助于证明的翻译. 该逻辑系统也需要类型系统的支持. 例如, 不同类型的赋值语句需采用不同的推理规则.

我们把 Hoare 逻辑的这个扩展称为指针逻辑, 它的设计基于下面的考虑.

由于别名问题, Hoare 逻辑不能直接用于有指针类型的语言, 需要对 Hoare 逻辑的规则增加一些约束并且需要增加一些规则来解决问题. 增加一些基本规则来表达值相等的访问路径或互为别名的访问路径的性质是简单的, 下面是这类性质的一些例子:

(1) 值相等的访问路径中, 若其中一个代表有效指针, 则其它的也都是;

(2) 给值相等的访问路径添加同样的后缀, 若

形成访问路径, 则结果互为别名;

(3) 互为别名的访问路径的值一定相等;

(4) 访问路径的别名关系满足自反性、传递性和对称性.

在 Hoare 逻辑的公式  $\{P\}S\{Q\}$  中,  $S$  是语法结构, 通常是语句,  $P$  和  $Q$  分别是它的前后条件. 下面考虑两种语句, 首先是指针类型的赋值语句  $p = q$ , Hoare 逻辑的正向赋值公理是

$\{Q\}p = q\{\exists p'.(p = q[p \leftarrow p'] \wedge Q[p \leftarrow p'])\}$ , 其中  $p' \notin \{p\} \cup FV(q) \cup FV(Q)$ ,  $FV$  是得到变元中自由变量集合的函数. 考虑  $p$  是有效指针的情况, 下面的约束得到满足时才能使用该公理.

(1) 前条件  $Q$  没有  $p$  的其它别名(其它别名指不是  $p$  本身). 若不满足, 可以尝试用上面提到的基本规则把  $Q$  变换到满足这个条件.

(2) 访问路径  $q$  也不以  $p$  的其它别名为前缀(在此对程序加这点限制是为了简化讨论).

(3) 前条件  $Q$  中一定有  $p == r$  这样的断言( $r$  不是  $p$  的别名). 这是为了保证该赋值不会引起内存泄漏.

再考虑为  $free(p)$  设计推理规则, 这里仅考虑  $p$  所指向对象不含有有效指针这种比较简单的情况. 考虑该规则的前条件和使用该规则的约束:

(1)  $p$  应该是有效指针. 它直接出现在该规则的前条件中.

(2) 前条件中没有以  $p$  (或与  $p$  相等的访问路径) 为前缀的访问路径, 除非出现在  $p \rightarrow next == \text{NULL}$  或  $p \rightarrow data == e$  ( $e$  是整型表达式) 这样的断言中.

该规则要能体现: 前条件中涉及  $p$  (包括和  $p$  相等的访问路径) 的基本断言, 在后条件中都被删除. 这样的要求难以仅用语法代换来表达.

例如, 若当前程序点的断言是  $p == q \wedge effective(p) \wedge p \rightarrow next == \text{NULL} \wedge p \rightarrow data == 10$ , 下一个语句是  $free(p)$ , 则期望该语句后程序点的断言是  $dangling(p) \wedge dangling(q)$ .

要想完成上述两种语句中的约束检查和断言删除等, 需要寻找新的方式来表达推理规则. 指针逻辑的推理规则设计基于下面的考虑:

(1) 若在某程序点能区分有效指针、NULL 指针和悬空指针, 并且知道有效指针之间是否相等, 则就能判断有关指针的操作是否安全, 还可以得出经过这步操作后指针信息的变化.

(2) 推理规则的设计要有利于用工具来进行自

动推导。

(3) 把相等的指针表达在一个集合中, 便于在推理规则中表示语句执行所引起的指针信息变化。

本文主要介绍证明指针性质的推理规则的设计。

### 3.1 基本运算的定义

在指针逻辑中, 程序点的 NULL 指针集合用  $\mathcal{N}$  表示, 悬空指针的集合用  $\mathcal{D}$  表示, 有效指针集合用  $\mathcal{I}$  表示。  $\mathcal{I}$  中指针的具体值并不重要, 重要的是它们是否相等, 因此基于相等与否把它们划分成若干等价集合。例如, 若  $\mathcal{I}$  中有等价集合  $\{p, q\}$ , 则它表示  $p$  和  $q$  是相等的有效指针, 并且它们不等于其它集合中的指针。一个等价集合不能删掉任何元素, 也不能分成若干子集, 因为这样做都会使指针信息发生变化。因此, 在指针逻辑的断言演算中,  $\mathcal{I}$  中的等价集合被看成命题常元; 同样,  $\mathcal{N}$  和  $\mathcal{D}$  也都被看成命题常元。这些集合只能用本节为指针赋值等设计的推理规则来改变。在语法结构的前后条件中,  $\mathcal{I}$  中的等价集合、 $\mathcal{N}$  和  $\mathcal{D}$  虽以集合方式出现, 但本质上是逻辑表达式, 因此用“ $\wedge$ ”连接它们。作为缩写, 有时用  $\Psi$  表示  $\mathcal{I} \wedge \mathcal{N} \wedge \mathcal{D}$ 。

访问路径是满足一定语法要求的字符串, 本文所说的串都是指构成访问路径的串或子串, 并用  $\text{Paths}$  表示访问路径集合。若访问路径  $p$  是  $q$  的前缀, 则谓词  $\text{prefix}(p, q)$  等于 true, 否则等于 false。符号“ $\cdot$ ”用于两个串的连接; 它也用于串的集合  $S$  和串  $s$  的连接, 使得  $S$  中的每个串连接  $s$ :

$$S \cdot s \triangleq S' \text{ where } s' \cdot s \in S' \text{ iff } s' \in S.$$

若  $s_1 \cdot s_2$  和  $s_1$  ( $s_1$  和  $s_2$  都不是空串) 是值相等的访问路径, 则称  $s_2$  是访问路径  $s_1 \cdot s_2 \cdot s_3$  ( $s_3$  也不是空串) 中的环。符号  $\equiv$  表示语法上等同,  $\equiv\equiv$  表示语法上等同测试。

下面先定义访问路径上的一些函数, 它们都以程序点的指针信息  $\Psi$  或  $\mathcal{I}$  为参数, 下面统一都将参数忽略。这些定义中出现的关键字在一些软件语言中都出现过, 在此忽略它们的解释。需要强调一下, 访问路径  $p$  和  $q$  在本文中几乎总是指称指针, 因此本文也经常直接称它们为指针; 但是, 在下面的函数中, 使用的是它们的语法表达式(访问路径)。

#### (1) 别名集合的计算

$\text{closure}(p)$  计算访问路径  $p$  的最简别名集合, 称为  $p$  的闭包, 它包含且仅包含  $p$  所有的无环别名。

$$\text{closure}(p) \triangleq \\ \text{if } \text{length}(p) = 1 \text{ then } \{p\}$$

else let  $s_1 \cdot s_2 \cdots s_{n-1} \cdot s_n \equiv p$  in

$\text{compression}(\text{expansion}(\text{closure}(s_1 \cdot s_2 \cdots s_{n-1}) \cdot s_n))$ , 其中,  $\text{length}(p)$  计算访问路径  $p$  的长度, 它是指  $p$  由几个有语法意义的部分组成, 而不是指  $p$  的字符个数, 例如  $t \rightarrow \text{next} \rightarrow \text{data}$  的长度为 3。

$\text{expansion}(S)$  用来在别名集合  $S$  中加入与其中访问路径相等的访问路径, 其定义如下:

$$\text{expansion}(S) \triangleq \\ \text{if } \exists S' : (\mathcal{I} \cup \{\mathcal{N}\} \cup \{\mathcal{D}\}). (S \cap S' \neq \emptyset) \\ \text{then let } \{p_1, \dots, p_n\} = S' - S \\ \text{where } S' \in (\mathcal{I} \cup \{\mathcal{N}\} \cup \{\mathcal{D}\}) \wedge S \cap S' \neq \emptyset \\ \text{in } S \cup \text{closure}(p_1) \cup \dots \cup \text{closure}(p_n) \\ \text{else } \emptyset.$$

$\text{compression}(S)$  用来删除别名集合  $S$  中带环的访问路径, 其定义如下:

$$\text{compression}(S) \triangleq S - S' \\ \text{where } (S' \subset S) \wedge ((s_1 \cdot s_2 \cdot s_3) \in S' \text{ iff} \\ (s_1 \neq \epsilon) \wedge (s_2 \neq \epsilon) \wedge (s_3 \neq \epsilon) \wedge \\ ((s_1 \cdot s_3) \in S) \wedge (s_1 \cdot s_2 = s_1)).$$

为清晰起见, 上面给出的是  $\text{closure}$  的一个定义, 而不是  $\text{closure}$  的实现算法, 例如, 该定义没有考虑面临双向循环链表等带环数据结构时, 递归计算的终止问题。在  $\text{closure}$  的实现中是不难把计算的终止等问题考虑进去的。有了  $\text{closure}$  函数, 也很容易删掉访问路径中的环, 为方便讨论, 我们认为程序中给出的都是最简访问路径。

#### (2) 访问路径的单个别名函数 $\text{alias}(p, q)$

该函数从访问路径  $p$  的别名集合中任取  $p'$ , 满足  $p'$  不以访问路径  $q$  的别名为前缀。若找不到这样的  $p'$ , 则结果仍是  $p$ 。

$$\text{alias}(p, q) \triangleq \\ \text{let } S = \{p' : \text{closure}(p) \mid \\ \forall q' : \text{closure}(q). \neg \text{prefix}(q', p')\} \\ \text{in if } S = \emptyset \text{ then } p \text{ else } p' \text{ where } p' \in S.$$

#### (3) 访问路径所在等价集合函数 $\text{equals}(p)$

若  $p$  的别名出现在某个等价集合中, 则返回该集合, 否则返回空集。

$$\text{equals}(p) \triangleq \\ \text{if } \exists S : \mathcal{I}. (S \cap \text{closure}(p) \neq \emptyset) \text{ then} \\ S \text{ where } S \in \mathcal{I} \wedge S \cap \text{closure}(p) \neq \emptyset \\ \text{else } \emptyset.$$

下面介绍在推理规则中直接使用的运算或谓词, 这些运算表达语句后条件中的  $\Psi$  是如何从前条件的  $\Psi$  得到的。

## (4) 有效指针的替换和删除运算

若  $S$  是  $\Pi$  的一个等价集合,  $p$  是一个有效指针, 则  $S/p$  表示对  $S$  中以  $p$  的别名为前缀的每个指针  $q$  都用  $alias(q, p)$  寻找一个别名来代替它, 然后将  $S$  中出现的  $p$  的别名和以它们为前缀的访问路径都删除.

$$S/p \triangleq$$

$$\text{let } S' = \{q:S \mid \forall p': closure(p). \neg prefix(p', q)\} \cup \{q': Paths \mid \exists q:S. \exists p': closure(p). (prefix(p', q) \wedge q' \equiv alias(q, p))\}$$

$$\text{in } \{q:S' \mid \neg(q \in closure(p)) \wedge \forall p': closure(p). \neg prefix(p', q)\}.$$

若需要对  $\Pi$  中每个  $S$  进行替换和删除  $p$  的运算, 则用  $\Pi/p$  表示.

当有效指针  $q$  被赋予一个不等于  $q$  的值时,  $q$  和以  $q$  为前缀的访问路径都需要从原来的等价集合中删除, 例如, 若  $\Pi = \{\{s, t \rightarrow prior\}, \{t, s \rightarrow next\}\}$ , 则  $\Pi/t = \{\{s, s \rightarrow next \rightarrow prior\}, \{s \rightarrow next\}\}$ .

## (5) 无效指针替换运算

$\mathcal{N} \setminus p$  和  $\mathcal{D} \setminus p$  分别用来表示将  $\mathcal{N}$  和  $\mathcal{D}$  中以  $p$  的别名为前缀的访问路径用它们的其它别名来代替.

$$\mathcal{N} \setminus p \triangleq$$

$$\{q:\mathcal{N} \mid \forall p': closure(p). \neg prefix(p', q)\} \cup \{q': Paths \mid \exists q:\mathcal{N}. \exists p': closure(p). (prefix(p', q) \wedge q' \equiv alias(q, p))\}.$$

$\mathcal{D} \setminus p$  的定义类似.

## (6) 无效指针删除运算

$\mathcal{N}/p$  和  $\mathcal{D}/p$  分别用来表示将  $\mathcal{N}$  和  $\mathcal{D}$  中出现的  $p$  的别名删除.

$$\mathcal{N}/p \triangleq \{q:\mathcal{N} \mid \neg(q \in closure(p))\};$$

$$\mathcal{N}/\{p_1, p_2, \dots, p_n\} \triangleq (((\mathcal{N}/p_1)/p_2) \dots /p_n).$$

$\mathcal{D}/p$  的定义类似.

## (7) 指针添加运算

并集算符“ $\cup$ ”直接用来表示向指针集合中添加一个指针, 例如  $S \cup \{p\}$ . 我们为  $\Pi$  中等价集合的增加、删除和替换使用新的记号, 它们基于集合运算符号“ $\cup$ ”和“ $-$ ”及它们的组合来定义.

$$\Pi + p \triangleq \Pi \cup \{\{p\}\}$$

——把仅由  $p$  构成的等价集合加到  $\Pi$  中;

$$\Pi - p \triangleq \Pi - \{equals(p)\}$$

——删掉  $\Pi$  中  $p$  所在的等价集合;

$$\Pi \text{ add } q \text{ to } p \triangleq (\Pi - p) \cup \{equals(p) \cup \{q\}\}$$

——把  $q$  加到  $\Pi$  中  $p$  所在的等价集合.

(8) 有效指针删除是否引起内存泄漏的测试  $leak(p)$ 

对有效指针  $p$  所在等价集合  $S$  进行  $S/p$  计算, 结果为空集合时则表示会出现内存泄漏; 否则不会.

$$leak(p) \triangleq equals(p)/p == \emptyset.$$

## (9) 一些基本谓词的定义

下面这些谓词用来测试指针  $p$  的别名是否在某个集合中.

$$p <: \Pi \triangleq \exists S:\Pi. (S \cap closure(p) \neq \emptyset);$$

$$p <: S \triangleq S \cap closure(p) \neq \emptyset$$

( $S$  是  $\Pi$  中的一个等价集合);

$$p <: \mathcal{N} \triangleq (\mathcal{N} \cap closure(p)) \neq \emptyset;$$

$$p <: \mathcal{D} \triangleq (\mathcal{D} \cap closure(p)) \neq \emptyset;$$

$$p <: \Psi \triangleq (p <: \Pi) \vee (p <: \mathcal{N}) \vee (p <: \mathcal{D}).$$

## 3.2 断言演算

把指针集合看成常元, 断言上的演算基本上仍遵守经典逻辑的演算, 只是对于指针集合, 不能使用  $A \wedge B \Rightarrow A$  和  $A \wedge B \Rightarrow B$ , 因为这会丢失指针信息.

另外, 对指针集合需要引入一些专用的规则, 受篇幅限制, 在此只列举部分规则.

(1) 判断  $\Psi$  是否有矛盾

例如, 下面的规则表示一个有效指针不能同时出现在两个不同的等价集合中.

$$\frac{\exists p: Paths. \exists S_1:\Pi. \exists S_2:\Pi. ((p <: S_1) \wedge (p <: S_2) \wedge (S_1 \neq S_2))}{\Psi \Rightarrow \text{false}}.$$

## (2) 吸收指针相等关系断言

在 PointerC 的程序中, 条件语句和循环语句的规则会把  $p == \text{NULL}$  和  $p == q$  等形式的断言分别引入两条件分支和循环体前程序点的断言中, 需要一些规则来把它们吸收到指针集合中或者推导出矛盾, 下面列出其中的一部分:

$$\frac{(p <: \Pi)}{\Psi \wedge (p \neq \text{NULL}) \Rightarrow \Psi};$$

$$\frac{(p <: \mathcal{N})}{\Psi \wedge (p \neq \text{NULL}) \Rightarrow \text{false}};$$

$$\frac{p <: \mathcal{N}}{\Psi \wedge (p == \text{NULL}) \Rightarrow \Psi};$$

$$\frac{p <: \Pi}{\Psi \wedge (p == \text{NULL}) \Rightarrow \text{false}}.$$

## (3) 别名替换

有时需要用下面的规则来进行别名替换:

$$\frac{q \in closure(p)}{\Psi \wedge Q \Rightarrow \Psi \wedge Q[p \leftarrow q]}.$$

### 3.3 公理和推理规则

下面给出联系到 PointerC 各种语句的公理和推理规则. 这些规则体现了相应语句引起的指针集合的增、减和替换. 在下面的所有规则中, 其前提中的断言都是基于语句前的那个程序点的  $\Psi$  来计算的.

(1) 指针之间的赋值语句  $p=q$  (包括  $q$  是常量 NULL 的情况, 略去了  $q$  是悬空指针的规则);

不同情况的指针赋值用不同的规则.

(a)  $p$  和  $q$  是相等的有效指针, 或都等于 NULL

$$\frac{\exists S:\Pi. (p<:S \wedge q<:S) \vee (p<:\mathcal{N} \wedge (q<:\mathcal{N} \vee q \equiv \text{NULL}))}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\} p=q \{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\}}.$$

(b)  $p$  和  $q$  是不相等的有效指针

$$\frac{\exists S_1:\Pi. S_2:\Pi. (S_1 \neq S_2 \wedge p<:S_1 \wedge q<:S_2) \wedge \neg \text{leak}(p)}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\} p=q \{((\Pi/p) \text{add } p \text{ to } q) \wedge \mathcal{N} \setminus p \wedge \mathcal{D} \setminus p\}}.$$

(c)  $p$  是 NULL 指针,  $q$  是有效指针

$$\frac{p<:\mathcal{N} \wedge q<:\Pi}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\} p=q \{(\Pi \text{ add } p \text{ to } q) \wedge \mathcal{N} / p \wedge \mathcal{D}\}}.$$

(d)  $p$  是悬空指针,  $q$  是有效指针

$$\frac{p<:\mathcal{D} \wedge q<:\Pi}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\} p=q \{(\Pi \text{ add } p \text{ to } q) \wedge \mathcal{N} \wedge \mathcal{D} / p\}}.$$

(e)  $p$  是有效指针,  $q$  等于 NULL

$$\frac{p<:\Pi \wedge (q<:\mathcal{N} \vee q \equiv \text{NULL}) \wedge \neg \text{leak}(p)}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\} p=q \{\Pi / p \wedge (\mathcal{N} \setminus p \cup \{p\}) \wedge \mathcal{D} \setminus p\}}.$$

(f)  $p$  是悬空指针,  $q$  等于 NULL

$$\frac{p<:\mathcal{D} \wedge (q<:\mathcal{N} \vee q \equiv \text{NULL})}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\} p=q \{\Pi \wedge (\mathcal{N} \cup \{p\}) \wedge \mathcal{D} / p\}}.$$

(2) 非指针类型的赋值公理

$$\{\Psi \wedge (Q[y_1 \leftarrow x] \cdots [y_n \leftarrow x][x \leftarrow e])\} x=e \{\Psi \wedge Q\}$$

( $y_1, y_2, \dots, y_n$  构成  $\text{closure}(x)$  的所有成员).

这是考虑了别名情况后的 Hoare 逻辑赋值公理, 它不改变  $\Psi$ . 该公理中的断言  $Q$  只含整型数据的子断言. 以下使用的  $Q$  也都满足这个限制.

(3) 对于指针类型的赋值, 若前条件是  $\Psi \wedge Q$ , 并且  $Q$  中也有访问路径 (包括作为  $Q$  中访问路径前缀的情况), 则除了用先前指针赋值的规则外, 还需要用下面的赋值公理.

$$\{\Psi \wedge Q\} p=q \{\Psi' \wedge Q[r \leftarrow p]\}$$

( $r$  是  $\text{closure}(p)$  中的一个其它成员,

$\Psi'$  由 (1) 的规则从  $\Psi$  得到).

换一种说法是, 对指针  $p$  赋值时, 对  $p$  的别名替换还需要出现在  $\Psi$  以外的断言中. 该规则也用在 malloc 和 free 语句场合.

(4) 复合、条件和循环语句的规则以及推论规则等仍然使用 Hoare 逻辑的规则.

(5) 分配空间语句  $p=\text{malloc}(T)$ , 其中  $T$  是类型. 若  $T$  是结构类型,  $r_1, r_2, \dots, r_n$  是其中的指针域在该类型中的访问路径.

(a)  $p$  是 NULL 指针

$$\frac{p<:\mathcal{N}}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\} p=\text{malloc}(T) \{(\Pi+p) \wedge \mathcal{N} / p \wedge (\mathcal{D} \cup \{p \rightarrow r_1, \dots, p \rightarrow r_n\})\}}.$$

(b)  $p$  是悬空指针的规则和 (a) 类似.

(c)  $p$  是有效指针. 将该语句看成语句序列  $p=\text{NULL}; p=\text{malloc}(T)$ , 并将相应的规则用于证明.

(6) 释放空间语句  $\text{free}(p)$

下面的规则用于  $p$  所指向对象不含有有效指针的场合. 若从  $\Pi$  知道  $p$  所指向的对象中含有有效指针  $p \rightarrow r_1, \dots, p \rightarrow r_n$ , 则可把该语句看成语句序列  $p \rightarrow r_1 = \text{NULL}; \dots; p \rightarrow r_n = \text{NULL}; \text{free}(p)$  来进行证明. 这样做的目的是简化  $\text{free}(p)$  的规则.

$$\frac{p<:\Pi}{\{\Pi \wedge \mathcal{N} \wedge \mathcal{D}\} \text{free}(p) \{(\Pi-p) \wedge (\mathcal{N} / \{p \rightarrow r_1, \dots, p \rightarrow r_n\}) \wedge (\mathcal{D} \cup \text{equals}(p))\}}.$$

## 4 证明实例

我们已经用指针逻辑系统证明了单链表、双向链表和二叉树等数据结构的一些函数. 本节以删除二叉排序树一个结点并重接它的左或右子树的函数 `struct node *DeleteNode(struct node *p)` 为例.

在证明该函数时, 参数所指向的树上, 有效指针和 NULL 指针的布局是不清楚的, 但它必须满足树的定义. 若树结点定义是 `struct node {int data; struct node *l, *r;}`, 那么以  $p$  为根结点指针的树的定义如下:

$$\text{tree}(p) \triangleq \{p\}_{\mathcal{N}} \vee (\{p\} \wedge \text{tree}(p \rightarrow l) \wedge \text{tree}(p \rightarrow r)).$$

如果  $p$  不是空指针, 通过下面的演算可以知道  $p$  是有效指针并且  $\text{tree}(p \rightarrow l) \wedge \text{tree}(p \rightarrow r)$  为真:

$$\begin{aligned} & \text{tree}(p) \wedge (p \neq \text{NULL}) \\ & \equiv (\{p\}_{\mathcal{N}} \vee (\{p\} \wedge \text{tree}(p \rightarrow l) \wedge \text{tree}(p \rightarrow r))) \wedge (p \neq \text{NULL}) \\ & \Leftrightarrow (\{p\}_{\mathcal{N}} \wedge (p \neq \text{NULL})) \vee (\{p\} \wedge \text{tree}(p \rightarrow l) \wedge \text{tree}(p \rightarrow r) \wedge (p \neq \text{NULL})) \\ & \Leftrightarrow \text{false} \vee (\{p\} \wedge \text{tree}(p \rightarrow l) \wedge \text{tree}(p \rightarrow r)) \\ & \Leftrightarrow \{p\} \wedge \text{tree}(p \rightarrow l) \wedge \text{tree}(p \rightarrow r). \end{aligned}$$

程序设计者只要给出函数前后条件和循环不变

式,其它程序点的断言可以通过指针逻辑得到.图 1 仅对形参  $p$  的左右子树都非空的大部分程序点插入了断言(该函数要求参数是非空树),其它部分在关键点插入了断言.在断言中,直接列出各等价集合

来表示  $\Pi, \mathcal{N}$  和  $\mathcal{D}$  用下标  $\{\dots\}_N$  和  $\{\dots\}_D$  来区分,它们为空时则不出现在断言中.我们没有给出 return 语句后的断言,因为本文没有提供 return 语句的推理规则.

```

{p!=NULL ∧ tree(p)}
struct node *DeleteNode(struct node *p)
{
    struct node *q, *s;
    { {p} ∧ tree(p → l) ∧ tree(p → r) ∧ {q,s}_D }
    if(p → r == NULL) /* 右子树为空,只需重接它的左子树 */
        {q=p; s=p → l; free(q); { {p,q}_D ∧ tree(s) } return s; }
    else if(p → l == NULL) /* 左子树为空,只需重接它的右子树 */
        {q=p; s=p → r; free(q); { {s} ∧ {p,q}_D ∧ tree(s) } return s; }
    else /* 左右子树均不空 */
    { { {p} ∧ {p → l} ∧ {p → r} ∧ {q,s}_D ∧ tree(p → l) ∧ tree(p → r) }
      q=p; s=p → l;
      if(s → r == NULL) /* 重接 *q 的左子树 */
          {q → l = s → l; p → data = s → data; free(s); { {p,q} ∧ tree(p) } return p; }
      else
      { { {p,q} ∧ {p → r} ∧ {p → l,s} ∧ {s → r} ∧ tree(p → l → l) ∧ tree(p → l → r) ∧ tree(p → r) }
        q=s; s=s → r;
        { ∃ n: N. ( {p} ∧ {p → r} ∧ ∀ i: 0..n-1. {p → l(→ r)^i} ∧ {p → l(→ r)^n, q} ∧ {p → l(→ r)^{n+1}, s} ∧
          ∀ i: 0..n. tree(p → l(→ r)^i → l) ∧ tree(p → l(→ r)^{n+1}) ∧ tree(p → r) ) } // * 循环不变式 */
        while(s → r != NULL) /* 转左,然后向右前进到头 */
            {q=s; s=s → r; }
        { ∃ n: N. ( {p} ∧ {p → r} ∧ ∀ i: 0..n-1. {p → l(→ r)^i} ∧ {p → l(→ r)^n, q} ∧ {p → l(→ r)^{n+1}, s} ∧ {s → r}_N ∧
          ∀ i: 0..n. tree(p → l(→ r)^i → l) ∧ tree(p → l(→ r)^{n+1}) ∧ tree(p → r) ) }
        p → data = s → data;
        q → r = s → r; /* 重接 *q 的右子树 */
        { ∃ n: N. ( {p} ∧ {p → r} ∧ ∀ i: 0..n-1. {p → l(→ r)^i} ∧ {p → l(→ r)^n, q} ∧ {s} ∧ ( {s → l, q → r} ∧ {s → r}_N )
          ∨ {s → r, s → l, q → r}_N ) ∧ ∀ i: 0..n. tree(p → l(→ r)^i → l) ∧ tree(p → l(→ r)^{n+1}) ∧ tree(p → r) ) }
        free(s);
        { ∃ n: N. ( {p} ∧ {p → r} ∧ ∀ i: 0..n-1. {p → l(→ r)^i} ∧ {p → l(→ r)^n, q} ∧ ( {q → r} ∨ {q → r}_N ) ∧ {s}_D ∧
          ∀ i: 0..n. tree(p → l(→ r)^i → l) ∧ tree(p → l(→ r)^{n+1}) ∧ tree(p → r) ) }
          /* 如果在返回调用者前,忽略 q 和 s 的信息,可以得到 { {p} ∧ tree(p) } */
        return p;
      }
    }
}

```

图 1 删除二叉树结点的程序和断言

## 5 相关工作

Hoare 逻辑的一个重要特征是用变量代换来抓住赋值的语义,本文的指针逻辑系统本质上是一种指针分析工具,它用访问路径的增加、删除和替换来抓住指针操作带来的影响.指针分析已经研究了 20 多年,历史上的指针分析主要回答:对指针类型的变量,它们运行时可能指向的对象集合是什么.这样的指针分析可用在程序的静态分析和优化的很多方面,如寄存器分配和常量传播所需的活跃变量分析,还有像 NULL 指针脱引用这样的潜在运行错误的静态检查等.近年来它还用在发现危及安全的缓冲区溢出和打印格式串错误等.和其它静态技术类似,指针分析受到可判定性问题的困扰,对大多数语言来说,所得到的解总是一个近似.

在指向对象集合的精度上,不同的应用要求不同的粒度.不同的精度要求采用不同的分析方法,有流敏感和流不敏感的区别,路径敏感和路径不敏感的区别以及过程内和过程间的区别.例如,Steensgaard 对 C 语言的一个禁止指针强制和指针算术等的子集,描述了一种基于类型推导的过程间的、流不敏感的和路径不敏感的指针分析<sup>[8]</sup>.该方法基于变量的存储模型,定义了类型系统及推导指针指向的规则集合,用以分析运行时指针变量可能指向的对象集合. Berndt 等首先把二叉决策图用于流不敏感和路径不敏感的指针分析<sup>[9]</sup>,比较成功地解决了这类分析的效率问题. Hind 对这方面的研究做了一个总结<sup>[10]</sup>,列出了一些待解决的问题.

出于软件安全方面的要求,本文实现的是精确而不是近似的指针分析,因此在不影响语言功能的情况下,对 C 语言中难以判定的指针使用方式进行

了限制,正是这种限制使得本文可以采用与通常的 C 语言指针分析完全不同的方法。

本文的指针逻辑和分离逻辑(separation logic)<sup>[11-12]</sup>都是通过对 Hoare 逻辑的拓展,来证明在共享易变数据结构(shared mutable data structure)上带指针操作的程序的性质.分离逻辑适合于对使用这些数据结构的低级命令式语言的程序进行推理.这样的简单语言包括了访问和修改共享结构的命令,包括了显式分配存储和释放存储的命令.分离逻辑在断言语言中引入了“分离合取”等空间连接词,它们可用来断言存储空间分离部分的性质,例如  $P * Q$  表示  $P$  和  $Q$  在两部分不相交的存储空间上分别成立.分离提供了分离逻辑最关键的特征——局部推理.分离逻辑还使用在抽象数据结构上递归定义的谓词,这些谓词允许简洁和灵活地描述有控制的共享的结构.分离逻辑在证明单链表、双向链表和二叉树等易变数据结构的程序上的成功已经展示出它的优点.近来分离逻辑已经出现在用低级语言写的操作系统一些核心程序的正确性证明上<sup>[13]</sup>.

本文的指针逻辑和分离逻辑的主要区别有两点.分离逻辑面向低级编程语言,指针逻辑面向高级编程语言.另外,由于 PointerC 不允许指针指向动态申请存储块的中间,那么从指针是否相等就可以判断它们指向的空间是否分离,因此指针逻辑不必引入分离合取这样的空间连接词。

Bornat 也采用 Hoare 逻辑来证明指针程序的性质<sup>[14]</sup>,采用类似方法的还有 Mehta 和 Nipkow<sup>[15]</sup>.Bornat 把堆看成由指针索引的一群对象,把对象看成由名字索引的一组成员,然后把 Hoare 逻辑赋值公理拓展成能用于对象成员赋值的场合,用它来证明一些指针程序的性质.由指针索引相等与否来判断别名,基于此来拓展赋值公理是他和我们方法的共同特征,但是他的方法只能用于不提供 free 操作并且有无用单元收集(garbage collection)的语言.我们的方法虽然适用于提供 free 操作的场合,但是却导致了指针逻辑的复杂.例如,在  $free(p)$  时,为防止悬空指针引用,需要保证以后不会用  $p$  或与  $p$  相等的指针去访问;再例如,在对有效指针  $p$  赋值时,为防止出现内存泄漏,需要知道有和  $p$  相等的指针存在.这就导致指针逻辑像是从前向后计算最强后条件,而不是从后向前计算最弱前条件。

## 6 结束语

本文提出了一种可对指针程序进行精确分析的

逻辑系统,它可用来证明指针程序是否满足定型规则的附加条件,以支持指针程序的安全性证明及其它性质证明.我们已经利用证明辅助工具 Coq<sup>[16]</sup>完成了指针逻辑对 PointerC 操作语义可靠的证明.PointerC 及其类型系统、指针逻辑的断言语言、指针逻辑可靠性证明、一些应用程序的证明实例以及基于第 1 节所提框架的出具证明编译器的一些实现工作,可在我们的项目网页 <http://ssg.ustcsz.edu.cn/lss/doc/index.html> 上找到。

下一步我们将考虑放宽对指针运算的约束范围,有限制地允许指针算术,以适应编程中经常使用的 calloc 存储分配.加上面向对象的语言构造,使程序性质证明具有更好的模块性也是下一步需要考虑的内容。

## 参 考 文 献

- [1] Morrisett G, Walker D, Crary K, Glew N. From system F to typed assembly language//Proceedings of the 25th ACM Symposium on Principles of Programming Languages. San Diego, 1998: 85-97
- [2] Mandelbaum Y, Walker D, Harper R. An effective theory of type refinements//Proceedings of the 8th International Conference on Functional Programming. Uppsala, Sweden, 2003: 213-225
- [3] Necula G. Proof-carrying code//Proceedings of the 24th ACM Symposium On Principles of Programming Languages. New York, 1997: 106-119
- [4] Appel A W. Foundational proof-carrying code//Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science. Boston, Massachusetts, USA, 2001: 247-258
- [5] Yu D, Hamid N A, Shao Z. Building certified libraries for PCC: Dynamic storage allocation. Science of Computer Programming, 2004, 50(1-3):101-127
- [6] Feng X, Shao Z, Vaynberg A, Xiang S, Ni Z. Modular Verification of Assembly Code with Stack-Based Control Abstractions//Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation. Ottawa, Canada, 2006: 401-414
- [7] Xi H. Applied type system: Extended abstract//Proceedings of TYPES 2003. LNCS 3085. Springer-Verlag, 2004: 394-408
- [8] Steensgaard B. Points-to analysis in almost linear time//Proceedings of the 23th Annual ACM Symposium on Principles of Programming Languages. Florida, USA, 1996: 32-41
- [9] Berndl M, Lhoták O, Qian F, Hendren L, Umanee N. Points-to analysis using BDDs//Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. San Diego, 2003: 103-114



- [10] Hind M. Pointer analysis: Haven't we solved this problem yet? //Proceedings of the ACM Workshop on Program Analysis for Software. Tools and Engineering. Snowbird, Utah, USA, 2001: 54-61
- [11] Reynolds J C. Separation logic: A logic for shared mutable data structures//Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science. Washington, DC, USA, 2002: 55-74
- [12] Parkinson M, Bierman G. Separation logic and abstraction//Proceedings of the 32nd ACM Symposium on Principles of Programming Languages. Long Beach, California, USA, 2005: 247-258
- [13] Bornat R, Calcagno C, O'Hearn P, Parkinson W. Permission accounting in separation logic//Proceedings of the 32nd ACM Symposium on Principles of Programming Languages. Long Beach, California, USA, 2005: 259-270
- [14] Bornat R. Proving pointer programs in Hoare logic//Proceedings of the 5th International Conference on Mathematics of Program Construction, Pontede Lima, Portugal, 2000: 102-126
- [15] Mehta F, Nipkow T. Proving pointer programs in higher-order logic. Information and Computation, 2005, 199(1-2): 200-227
- [16] Bertot Y, Castéran P. Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Construction. Texts in Theoretical Computer Science, an EATCS series. Berlin: Springer Verlag, 2004



**CHEN Yi-Yun**, born in 1946, professor, Ph.D. supervisor. His research interests include theory and implementation of programming languages, formal description technologies, and software safety and security.

**HUA Bao-Jian**, born in 1979, Ph.D. candidate. His research interests include program verification, program logic, and software safety and security.

**GE Lin**, born in 1979, Ph.D. candidate. Her research interests include program verification, software safety and security, and type theory and system.

**WANG Zhi-Fang**, born in 1982, Ph.D. candidate. His research interests include software safety and security, program logic, and program verification.

## Background

This research is supported by the National Natural Science Foundation of China (Verification and Compilation of Software Safety, grant No. 60673126).

Proof-Carrying Code (PCC) brings two grand challenges to the research field of programming languages. One is to seek more expressive logic or type systems to specify or reason about the properties of high-level or low-level programs. The other is to study the technology of certifying compilation in which the compiler generates proofs for programs with annotations. For the first challenge, Typed Assembly Language and the theory of type refinements are two typical research projects in type-based approaches, while PCC, Foundational Proof-Carrying Code and Certified Assembly Programming are typical research projects on logic-based techniques. Type-based and logic-based techniques are complementary to each other, some researchers have tried to combine those techniques.

This paper presents the authors' research progress in the

first challenge. A pointer logic is designed for PointerC language, a subset with explicit memory allocation and deallocation of C-like programming languages, in the authors' research. As an extension of Hoare logic, the pointer logic is used to prove properties of pointer programs. The main characteristic of the pointer logic is that its inference rules are used to catch the modification of pointer information caused by the execution of each statement. Based on these rules, we can reason out the null pointers, dangling pointers and equality of effective pointers at each point in a program, and then calculate alias set of each pointer. These are the base information to prove safety and other properties of pointer programs.

The main contribution of this paper is the elementary design of the pointer logic. In the project, the authors' have implemented a certifying compiler based on the pointer logic and proved safety of PointerC language and soundness of the pointer logic.