一种用于指针程序验证的指针逻辑*

陈意云，李兆鹏+，王志芳，华保健

(中国科学技术大学 计算机科学系，安徽 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)

A Pointer Logic for Verification of Pointer Programs

CHEN Yi-Yun , LI Zhao-Peng+, WANG Zhi-Fang, HUA Bao-Jian

(Department 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: Phn: +86-512-87161322, E-mail: zpli@mail.ustc.edu.cn, http://ssg.ustcsz.edu.cn
Received 2008-00-00; Accepted 2008-00-00

Chen YY, Li ZP, Wang ZF, Hua BJ. A Pointer Logic for Verification of Pointer Programs. *Journal of Software*, 2009, 15(1):0000~0000.
http://www.jos.org.cn/1000-9825/15/0000.htm

Abstract: Proof-Carrying Code brings two grand challenges to the research field of programming languages. One is to seek more expressive program logics 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. This paper improves and extends the pointer logic we have designed for verifying pointer programs. The main contribution is that we present a concept of legal sets of access paths, which simplifies elementary operations on access paths and make inference rules of the logic easy to understand. Furthermore, we extend the logic with inference rules for local reasoning and for function construction, making the logic conveniently used in the context of function calls.
Key words: Software Safety; Hoare Logic; Pointer Logic; Proof-Carrying Code; Certifying Compiler

摘 要: 携带证明的代码作为一种新的代码范型，给编程语言的研究领域带来两项巨大挑战。其一是寻找表达力更强的逻辑或类型系统来规范或推理高级语言程序或低级语言程序的性质；其二是研究出具证明的编译技术。本文改进并扩展先前为验证指针程序提出的指针逻辑，主要贡献是提出了合法访问路径集合的概念，大大简化了访问路径上的基本运算，并使得指针逻辑推理规则变得易理解，另外，增加了局部推理规则和函数构造的推理规则，使得指针逻辑可以方便地用于有函数调用的场合。
关键词: 软件安全;Hoare 逻辑;指针逻辑;携带证明的代码;出具证明的编译器
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. 60673126，90718026（国家自然科学基金）

作者简介: **陈意云**（1946-）：男，湖南湘乡人，教授，博士生导师，主要研究领域为程序设计语言的理论和实现技术、形式描述技术、软件安全。**李兆鹏**（1978-）：男，博士，主要研究领域为程序验证、软件安全、类型系统和理论。**王志芳**（1982-）：男，博士生，主要研究领域为指针程序的安全性证明。**华保健**（1979-）：男，博士，主要研究领域为程序验证、类型系统、软件安全。

携带证明的代码（Proof-Carrying Code，PCC）[1]作为一种新的代码范型（paradigm），给编程语言的研究领域带来两项巨大挑战。其一是寻找表达力更强的逻辑或类型系统来规范或推理高级语言或低级语言程序的性质；其二是研究出具证明的编译技术。

对于第一个挑战，类型化汇编语言[2]和类型细化的理论[3]是两种基于类型方式的典型研究，而PCC、携带基础逻辑证明的FPCC[4]、经过验证的汇编编程CAP[5]和基于栈的经过验证的汇编编程SCAP[6]是基于逻辑方式的典型研究。基于类型的技术和基于逻辑的技术是互补的，近年来一些研究人员倾向于组合这两种技术。应用类型系统ATS[7]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，从而可以在类型系统上模拟Hoare 逻辑的推理。

对于第二个挑战，Necula 作为先驱者实现了一个叫做 Touchstone 的出具证明编译器（certifying compiler）[8]，它包含一个类型安全的 C 语言小子集的传统编译器，还有为该编译器生成的汇编代码自动产生类型安全证明的证明器。Touchstone 的主要限制是在指针类型和动态存储分配方面。随后，Colby 等实现了 Special J [9]，这是 Java 语言较大子集的一个出具证明编译器，它把 Java 字节代码编译成英特尔 IA32 体系结构的目标代码。

近年来，我们研究了将出具证明编译和携带证明代码的技术用到有显式内存管理的编程语言。我们设计了有动态内存分配和回收机制的类 C 小语言 PointerC[10]，其基本安全策略要求程序运行时不出现通过 NULL 指针或悬空指针（dangling pointer）进行存取指向对象的操作、不把 NULL 指针或悬空指针作为 free 函数调用的实在参数、不发生内存泄漏等。我们采用一种基于逻辑的技术和基于类型的技术相结合的方式来推导程序的性质。为了使类型系统简单并保证语言的安全性，我们在定型规则（typing rule）中增加了约束语法表达式的值的副条件。为了静态检查这些副条件，我们为 PointerC 语言设计了一种指针逻辑[11, 12]，它是 Hoare 逻辑的一种拓展，新颖之处是将精确的指针分析应用于程序验证。指针逻辑可用来从前向后收集各指针是 NULL 指针、悬空指针还是有效指针（valid pointer，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的副条件，以支持对指针程序的安全性验证和其它性质的验证。我们完成了 PointerC 语言安全性和指针逻辑可靠性的证明[13]，并实现了 PointerC 出具证明编译器的一个原型[14, 12]。

本文是对指针逻辑前期设计的改进和扩展，主要贡献如下：

（1）提出了合法（legal）访问路径集合的概念，大大简化了访问路径上基本运算的定义，并使得指针逻辑推理规则变得直观和易理解。

（2）增加了局部推理规则和函数构造的推理规则，使得指针逻辑可以方便地用于有函数调用的场合。

（3）和广泛使用的分离逻辑进行了深入比较。指针逻辑相当于高级语言层面的分离逻辑，其优于分离逻辑之处是能够防止内存泄漏。

本文全面介绍重新设计的指针逻辑：第一节介绍 PointerC 语言，第二节介绍指针逻辑的重新设计，第三节给出采用指针逻辑的一个证明实例，第四节比较指针逻辑和分离逻辑，第五节是相关工作比较，第六节是总结。

1 PointerC语言

PointerC 是一个强调指针类型的类 C 小语言，它略去函数声明部分的语法如图 1，其类型系统可见[10]。

在 PointerC 中，指针类型的变量只能用于赋值、相等与否比较、存取指向对象等运算以及作为函数的参数；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同尚未释放空间无任何重叠。另外，布尔表达式不采用短路计算，以方便它们出现在断言中，因为短路的与和或运算都不是可交换运算。

以指针类型声明变量开始的 lval 称为访问路径，在本文中，访问路径的前缀（本文所说的前缀都是指真前缀）只指那些仍能构成访问路径的前缀，给访问路径加后缀只指加上后缀仍构成访问路径的情况。

我们用$\mathcal{N}$表示指针类型访问路径的一个集合；用$\mathcal{D}$表示指针类型访问路径的另一个集合，用Π表示指针类

（程序） program ::= structdeclist vardeclist stmtlist
（结构声明列表） structdeclist ::= structdeclist structdec | ε
（结构声明） structdec ::= **struct** ident { vardeclist };
（变量声明列表） vardeclist ::= vardeclist vardec | ε
（变量声明） vardec ::= type idlist;
（标识符列表） idlist ::= idlist, id | id
（类型） type ::= **bool** | **int** | **struct** ident*
（语句序列） stmtlist ::= stmtlist stmt | stmt
（语句） stmt ::= lval = exp; | lval = **malloc**(**struct** id); | **if** (boolexp) block **else** block;
| **while** (boolexp) block; | **free** (exp);
（语句块） block ::= stmt | {stmtlist }
（表达式） exp ::= number | **NULL** | lvar | – exp | exp + exp | exp – exp | exp × exp | …
（布尔表达式） boolexp ::= **true** | **false** | exp == exp | exp != exp | …
| ¬ boolexp | boolexp ∧ boolexp | boolexp ∨ boolexp
（左值表达式） lval ::= id| lval->id
（自然数） number ::= 0 | 1 | 2 | …

Fig 1. Syntax of PointerC Language

图 1. PointerC 语言语法

型访问路径的另外一组集合，Π的每个集合中所有访问路径的类型相同，并且没有访问路径在Π中多次出现。用Ψ作为它们的总称。在语义模型上，$\mathcal{N}$用来表示程序点的 NULL 指针集合，$\mathcal{D}$用来表示程序点的悬空指针集合，Π的每个集合表示程序点一组指向同一个对象的指针（即右值相等的指针类型访问路径），并且Π中不同集合的指针指向不同对象。Π各集合中的指针都称为有效指针。因此，Ψ可以看成程序状态中所有指针的一种强调相等与否而不关心具体值的抽象。区分$\mathcal{N}$和$\mathcal{D}$是因为程序可以通过判断访问路径是否等于 NULL 来区分 NULL 指针和悬空指针。我们需要合法良形的Ψ，因为并非任意的Ψ都能表达上面的意思。先基于Π来定义访问路径的别名。出现在Π同一个集合中的访问路径加上同样后缀形成的访问路径互为别名，访问路径别名关系还是自反的。按此定义，判断两条访问路径是否互为别名的谓词如下：

alias(p, q) ≜

p≡q ∨ ∃s.∃ r,t,p′,q′. (s 不是空串 ∧ r 和 t 属于Π的同一集合∧p≡(p′·s) ∧ q≡(q′·s) ∧ alias(p′, r) ∧ alias(q′, t))

其中≡表示语法上相同，“·”表示串并置。该定义是说，若 p 和 q 不相同，则忽略相同后缀 s 之后，判断它们的别名是否出现在Π的同一个集合中。若递归计算时出现再次判断 p 和 q 是否为别名，则计算将会不终止，这时也认为 p 和 q 不是别名而停止计算。

合法Ψ需要满足下面几个条件：

（1）所有指针类型的声明变量都在Ψ中。

（2）对Π各集合中的任何访问路径 p，若 p 所指向结构类型有指针类型域 r，则 p->r 的某个别名在Ψ中。

（3）对Ψ中任何访问路径，Ψ中的任何其它访问路径都不是它的别名。

（4）若访问路径 p 在Ψ中，则对 p 的每个前缀，它的一个别名在Π中。

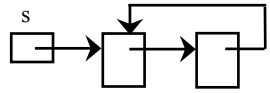

Fig 2. Cyclic Linked List with two nodes

图 2 两个结点的循环链表

和合法Ψ中的某条访问路径互为别名的访问路径都叫做合法的指针类型访问路径。整型访问路径是合法的，若其各前缀都是

（断言）assertion ::= boolexp | ¬ assertion | assertion ∧ assertion
| assertion ∨ assertion | ∀id:domain.assertion
| ∃id:domain.assertion | (assertion) | ident (lval)
| {lvalset} | $\{lvalset\}_N$ | $\{lvalset\}_D$
（值域） domain ::= **N** | exp..exp
（左值表达式集合）lvalset ::= lvalset, lval | lval
（左值） lval ::= … | lval(->ident)^{exp}

Fig 3. Syntax of Assertion Language
图 3. 断言语言语法

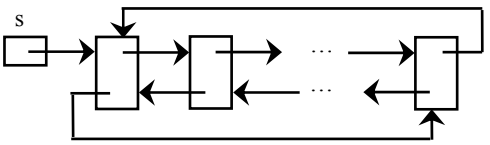

Fig 4. Cyclic Doubly-linked List
图 4 双向循环链表

合法的指针类型访问路径。程序点的所有合法访问路径的集合用$\mathcal{P}$表示。

若某访问路径的两个不同前缀都有别名出现在Π的同一个集合中，则称该访问路径带环。例如，若Π只有两个集合{s, s->next->next}和{s->next}，$\mathcal{N}$和$\mathcal{D}$都是空集，如图 2 所示，则 s->next->next->next 是带环的访问路径。

无环访问路径称为最简访问路径，下面假定程序中仅出现最简访问路径。

2 指针逻辑的设计

为证明程序满足基本安全策略，我们除了为 PointerC 设计一个类型系统外，还为它设计了一个证明系统。因为该类型系统的某些定型规则中有副条件，例如，程序表达式中出现的访问路径必须是合法访问路径，free 操作不能引起内存泄漏等，这些副条件都不能由通常的类型系统来检查，我们用 Hoare 逻辑的一种扩展（称为指针逻辑）来证明它们。本节介绍指针逻辑的重新设计，下面先介绍指针逻辑所用的断言语言。

2.1 断言语言

断言语言的语法如图 3 所示。其中 exp、boolexp 和 lval 的语法见 PointerC 语法，{lvalset}用来表示访问路径集合，带下标 N 或 D 的分别表示$\mathcal{N}$集合和$\mathcal{D}$集合。lval(->ident)^{exp} 是断言语言增加的一种访问路径表示，s(->next)^i 是 s->next->next…->next 的缩写（其中->next 出现 i 次），若 i = 0 则 s(->next)^i 就表示 s。

在该文法的基础上，对断言的一些其它约束如下：

（1）domain 中的 **N** 是指自然数集，exp..exp 中的 exp 和作为访问路径上角标的 exp 限制为最多含一个约束变元 n 的整型表达式 n ± b 或 b，其中 b 可以是含程序变量和常量的只有加减运算的表达式。

（2）当访问路径集合受多个量词约束时，存在量词约束出现在最左边。

（3）访问路径集合之前不可以出现¬。

例如，∀i:0..n-1.{s->l(->r)^i}是指{s->l} ∧ {s->l->r} ∧ …∧ {s->l(->r)$^{n-1}$}，这些访问路径集合都是Ψ中Π的一个集合。有了量词和带上角标的访问路径，图 4 中带头结点的双向循环链表用断言定义如下：

$$dlist(s, n) \triangleq (\forall k:1..n.\ \{s(->r)^k, s(->r)^{k+1}->l\}) \wedge \{s, s(->r)^{n+1}, s->r->l\}$$

其中s是 struct ListNode {struct ListNode *r, *l;}类型的指针，n表示除头结点以外的结点数（非负整数）。全称量词所辖的最后一个集合可以写成{s(->r)^n, s->l}，以删掉s(->r)$^{k+1}$->l中的环（这时k为n），上式是为了把它统一到全称量词辖域中。因此断言中的访问路径允许带环。该定义的n+1个访问路径集合将图2中2n+3个指针的相等关系表达得一清二楚。并且对图4的每个指针，该定义都从能够表示它的互为别名的访问路径中选一条访问路径来表示它。注意，dlist定义中最后一个集合改写成{s, s->l->r, s(->l)$^{n+1}$}看上去是合理的，但这时该定义式不是一个合法的Ψ。因为{s, s->l->r, s(->l)$^{n+1}$}中除了s以外，访问路径都是s->l…的形式，而其它有效指针集合中的访问路径都是s->r…的形式，在n大于0时，在这些集合中找不到s->l->r的前缀s->l的别名。

有些数据结构的断言定义需要引入归纳，例如二叉树的定义如下（s 仍然是 ListNode 类型的指针）：

$$\text{tree(s)} \triangleq \{s\}_N \vee (\{s\} \wedge \text{tree(s->l)} \wedge \text{tree(s->r)})$$

其中$\{s\}_N$表示空树，另一种情况$\{s\} \wedge \text{tree(s->l)} \wedge \text{tree(s->r)}$表示非空树，并且表达出该树上的指针都不相等，因为若把归纳定义的引用展开，代表有效指针的各访问路径都在Π的不同集合中。

2.2 断言演算

利用经典逻辑中有关∧和∨的等价公理可以把含访问路径集合的断言变换成析取范式，在变换过程中每个访问路径集合看成一个逻辑常元。在断言的析取范式中，每个子句中的访问路径集合部分构成一个Ψ。例如，上面二叉树归纳定义右部的断言就是一个析取范式，两个子句中的Ψ代表两种不同情况。注意，子句中可能还有其它断言，有些断言甚至和判断Ψ的合法性有关，例如，数据结构归纳定义的引用、涉及访问路径上角标的断言。

依据不丢失访问路径信息的原则，经典逻辑的蕴涵公理 $A \wedge B \Rightarrow A$ 在 B 含访问路径集合或归纳定义的引用时不可使用，因为使用它会导致访问路径信息的丢失，逻辑蕴涵公理 $A \wedge B \Rightarrow B$ 的情况类似。

我们还需要增加一些和Ψ有关的逻辑等价或蕴涵公理。

1）$\mathcal{N}$和$\mathcal{D}$的等价公理

$\mathcal{N}_1 \wedge \mathcal{N}_2 \Leftrightarrow \mathcal{N}$ （若$(\mathcal{N} == \mathcal{N}_1 \cup \mathcal{N}_2) \wedge (\mathcal{N}_1 \cap \mathcal{N}_2 == \varnothing)$）

$\mathcal{D}_1 \wedge \mathcal{D}_2 \Leftrightarrow \mathcal{D}$ （若$(\mathcal{D} == \mathcal{D}_1 \cup \mathcal{D}_2) \wedge (\mathcal{D}_1 \cap \mathcal{D}_2 == \varnothing)$）

这些等价性存在的依据是等价式两边包含同样的信息。

2）非合法Ψ公理

引入量词和数据结构的归纳定义后，需要将先前合法Ψ的条件加以补充。

（1）合法Ψ的检查是对合取范式的各子句分别进行。

（2）量词的引入并不改变合法Ψ的条件。它给Ψ合法性的检查带来困难，该问题将另文讨论。

（3）出现数据结构归纳定义的引用时，把每个引用式中作为变元的指针类型访问路径当成$\mathcal{N}$集合中的元素，然后按先前的条件来检查Ψ的合法性。不展开归纳定义的引用意味着暂不关心未展开部分，而把变元看成$\mathcal{N}$的元素也就是在检查合法性时暂不管未展开部分。

（4）数据结构的归纳定义必须是合法的。就是说归纳定义右部析取范式的每个子句都必须是合法的。

可以把合法性的概念扩大到析取范式的完整子句，即除了Ψ以外，还有其它断言（用 Q 作为总称）。$\Psi \wedge Q$ 是合法的，若Ψ是合法的，并且 Q 中出现的任何访问路径也都是合法的。

具体的蕴涵公理如下：

$\Psi \Rightarrow \text{false}$ （若Ψ不是合法的）

$\Psi \wedge Q \Rightarrow \text{false}$ （若Ψ或 Q 不是合法的）

3）Ψ等价公理

（1）在合法Ψ下，若访问路径集合$\mathcal{R}_1$和$\mathcal{R}_2$满足$\forall p{:}\mathcal{R}_1.\exists q{:}\mathcal{R}_2.\text{alias(p, q)} \wedge \forall q{:}\mathcal{R}_2.\exists p{:}\mathcal{R}_1.\text{alias(q, p)}$，并且$\mathcal{R}_1$和$\mathcal{R}_2$的元素个数一样，则称它们等价。访问路径集合等价公理如下：

$\mathcal{R}_1 \Leftrightarrow \mathcal{R}_2$ （若在合法Ψ下，$\mathcal{R}_1$和$\mathcal{R}_2$等价）

（2）下面用$\mathcal{S}_1, \ldots, \mathcal{S}_n$表示构成Π的 n 个访问路径集合。则合法Ψ等价公理是：

$\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n \wedge \mathcal{N} \wedge \mathcal{D} \Leftrightarrow \mathcal{S}'_1 \wedge \ldots \wedge \mathcal{S}'_n \wedge \mathcal{N}' \wedge \mathcal{D}'$

（若左右两边都是合法的，并且各对应集合都基于左边等价）

4）Ψ对布尔表达式的吸收或否定

Hoare 逻辑中条件语句和循环语句的推理规则会使程序中的布尔表达式 p==NULL、p!=NULL、p==q 和 p!=q（p 和 q 都是指针类型）等出现在断言中。若它们和Ψ无矛盾则被吸收，有矛盾则断言为假。下面列举一些这样的蕴涵公理。其中记号$\mathcal{S}^{p}$表示访问路径 p 的别名所在的集合，$\mathcal{S}^{p,q}$的含义类似。

$\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^{p} \wedge \mathcal{N} \wedge \mathcal{D} \wedge p==NULL \wedge Q \Rightarrow false$

$\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n} \wedge \mathcal{N}^{p} \wedge \mathcal{D} \wedge p==NULL \wedge Q \Rightarrow \mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n} \wedge \mathcal{N}^{p} \wedge \mathcal{D} \wedge Q$

$\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^{p} \wedge \mathcal{N} \wedge \mathcal{D} \wedge p!=NULL \wedge Q \Rightarrow \mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^{p} \wedge \mathcal{N} \wedge \mathcal{D} \wedge Q$

$\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^{p,q} \wedge \mathcal{N} \wedge \mathcal{D} \wedge p == q \wedge Q \Rightarrow \mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^{p,q} \wedge \mathcal{N} \wedge \mathcal{D} \wedge Q$

$\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^{p,q} \wedge \mathcal{N} \wedge \mathcal{D} \wedge p != q \wedge Q \Rightarrow false$

例如，对于先前的二叉树归纳定义，若 p!=NULL 则有：

$tree(p) \wedge p!=NULL \equiv (\{p\}_N \vee (\{p\} \wedge tree(p->l) \wedge tree(p->r))) \wedge p!=NULL$（≡在此表示将定义展开）

$\Rightarrow \{p\}_N \wedge p!=NULL \vee \{p\} \wedge tree(p->l) \wedge tree(p->r) \wedge p!=NULL$

$\Rightarrow false \vee \{p\} \wedge tree(p->l) \wedge tree(p->r)$

$\Rightarrow \{p\} \wedge tree(p->l) \wedge tree(p->r)$

2.3 指针逻辑的推理规则

先定义一些在推理规则中要用到的基本运算，它们的计算基于相应程序点的Ψ。

（1）删除访问路径：在访问路径集合$\mathcal{R}$中删除访问路径 p 的别名或集合$\{p_1, ..., p_n\}$中各访问路径的别名。

$\mathcal{R} - p \triangleq \{q : \mathcal{R} \mid \neg alias(q, p)\}$

$\mathcal{R} - \{p_1, ..., p_n\} \triangleq ((\mathcal{R} - p_1) ... - p_n)$

（2）增加访问路径：在访问路径集合$\mathcal{R}$中加入访问路径 p 或集合$\mathcal{R}'$中各访问路径。

$\mathcal{R} + p \triangleq \mathcal{R} \cup \{p\}$

$\mathcal{R} + \mathcal{R}' \triangleq \mathcal{R} \cup \mathcal{R}'$

（3）前缀替换：对访问路径集合$\mathcal{R}$中以 p 的别名为前缀的每条访问路径 q，都用它的一个别名 q′来代替，q′不以 p 的别名为前缀；$\mathcal{R}$中其余访问路径维持不变。下面定义中谓词 prefix(r, q)表示 r 是 q 的前缀。

$\mathcal{R}/p \triangleq \mathcal{R}'$ **where** $\mathcal{R}' \Leftrightarrow \mathcal{R} \wedge \forall q:\mathcal{R}'.(\neg\exists r:\mathcal{P}.(alias(r, p) \wedge prefix(r, q)))$

（4）测试内存泄漏 leak($\mathcal{S}$, p)

删除Π集合$\mathcal{S}$中的访问路径 p 时（出现在对 p 赋值时），若$\mathcal{S}$中所有访问路径都是 p 的别名或以 p 的别名为前缀，则会出现内存泄漏。

$leak(\mathcal{S}, p) \triangleq \forall q:\mathcal{S}.(alias(q, p) \vee \exists r:\mathcal{P}.(alias(r, p) \wedge prefix(r, q)))$

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访问路径集合的增、删和替换。在下面所有规则中，访问路径上的计算都是基于语句前那个程序点的Ψ，并且 Q 中能被Ψ吸收的布尔表达式已被吸收。

1）指针类型赋值语句 p = q 和 p = NULL

下面只列举几种情况下的规则，其它情况下的规则类似。

（1）p 和 q 都有别名在Π的同一个集合中

$\{\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^{p,q} \wedge \mathcal{N} \wedge \mathcal{D} \wedge Q\}\ p = q\ \{\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^{p,q} \wedge \mathcal{N} \wedge \mathcal{D} \wedge Q\}$

（2）p 的别名在$\mathcal{N}$中，q 的别名在Π的某个集合中

$\{\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^{q} \wedge \mathcal{N}^{p} \wedge \mathcal{D} \wedge Q\}\ p = q\ \{\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge (\mathcal{S}^{q} + p) \wedge (\mathcal{N}^{p} - p) \wedge \mathcal{D} \wedge Q\}$

（3）p 的别名在Π的某个集合中，q 的别名在$\mathcal{N}$中

$\{\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^p \wedge \mathcal{N}^q \wedge \mathcal{D} \wedge Q\}\ p = q\ \{\mathcal{S}_1/p \wedge \ldots \wedge \mathcal{S}_{n-1}/p \wedge (\mathcal{S}^p/p - p) \wedge (\mathcal{N}^q/p + p) \wedge \mathcal{D}/p \wedge Q/p\}$

（若对于赋值前的Ψ，leak($\mathcal{S}^p$, p)为假）

其中 Q/p 的意思也是把 Q 中以 p 的别名为前缀的访问路径都用它们不以 p 的别名为前缀的别名替换。

（4）赋值语句是 p = NULL 的形式且 p 的别名$\mathcal{D}$中

$\{\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n \wedge \mathcal{N} \wedge \mathcal{D}^p \wedge Q\}\ p = NULL\ \{\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n \wedge (\mathcal{N}^p + p\} \wedge (\mathcal{D}^p - p) \wedge Q\}$

（5）p 的别名和 q 别名在不同的Π中

可以把该语句当作 dummy = p; p = NULL; p = q′; dummy = NULL 语句序列来证明，当然也可以设计专门的规则。其中 q′ = q[p←dummy]，即将 q 中的 p 用 dummy 代换。每个函数可以看成有一个虚拟局部变量 dummy，初值是 NULL，每次遇到这类赋值时都这么处理，在函数结束时将 dummy 从$\mathcal{N}$中去掉。

2）非指针类型的赋值语句 x = e

在此用的是正向的赋值公理，其中 FV 表示变元中的自由变量集合。

$\{\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n \wedge \mathcal{N} \wedge \mathcal{D} \wedge Q\}\ x = e$

$\{\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n \wedge \mathcal{N} \wedge \mathcal{D} \wedge (\exists x'.(x==e[x\leftarrow x'] \wedge Q[y_1\leftarrow x]\ldots[y_n\leftarrow x][x\leftarrow x'])\}$

($y_1, \ldots, y_n$是 Q 中出现的 x 的所有别名，$x' \notin (\{x\} \cup FV(e) \cup FV(Q))$)

3）复合、条件和循环语句的规则

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。

4）分配空间语句 p = malloc(**struct** t)

其中 t 是结构类型，并假设 $r_1, \ldots, r_n$ 是 t 的指针类型域的域名。下面只列举两种情况。

（1）p 的别名在$\mathcal{N}$中

$\{\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n \wedge \mathcal{N}^p \wedge \mathcal{D} \wedge Q\}\ p = malloc(\textbf{struct}\ t)$

$\{\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n \wedge \{p\} \wedge (\mathcal{N}^p - p) \wedge (\mathcal{D} + \{p->r_1 \ldots + p->r_n\}) \wedge Q\}$

（2）p 的别名在Π的某个集合中

可以把该语句看成语句序列 p = NULL; p = malloc(**struct** t)来进行证明，当然也可以设计专门的规则。

5）释放空间语句 free(p)

只有 p 在Π的某个集合中，才能使用 free 函数。根据 p 所指对象的类型可以知道其所有指针类型域的访问路径 $p->r_1, \ldots, p->r_n$。把该语句当成语句序列 $p->r_1=NULL; \ldots; p->r_n=NULL;$ free(p)来进行证明。这样做的目的是简化 free(p)的规则。

$\{\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^p \wedge \mathcal{N} \wedge \mathcal{D} \wedge Q\}\ free(p)$

$\{\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge (\mathcal{N} - \{p->r_1, \ldots, p->r_n\}) \wedge (\mathcal{D} + \mathcal{S}^p) \wedge Q\}$（若 Q 中没有 p 的别名）

6）分情况规则

需要增加一条分情况证明规则，因为一个语句前条件的析取范式可能使Ψ分成多种情况。该规则如下：

$$\frac{\{\Psi_1 \wedge Q_1\}\ S\ \{\Psi_3 \wedge Q_3\} \qquad \{\Psi_2 \wedge Q_2\}\ S\ \{\Psi_4 \wedge Q_4\}}{\{\Psi_1 \wedge Q_1 \vee \Psi_2 \wedge Q_2\}\ S\ \{\Psi_3 \wedge Q_3 \vee \Psi_4 \wedge Q_4\}}$$

其中 S 表示 PointerC 的程序段。Hoare 逻辑原本没有这样的规则，因为{x==1 ∨ x ==2}这样的前条件虽然也可以看成两种不同情况，但 Hoare 逻辑赋值公理可以同时对该析取范式的两个子句进行语法代换。

7）结构规则（frame rule）

为了获得更有效的推理方法，有时需要拆分或合并合法Ψ。若合法Ψ是$\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n \wedge \mathcal{N} \wedge \mathcal{D}$，$\Psi_1$和$\Psi_2$

分别是$\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_i \wedge \mathcal{N}_1 \wedge \mathcal{D}_1$和$\mathcal{S}_{i+1} \wedge \ldots \wedge \mathcal{S}_n \wedge \mathcal{N}_2 \wedge \mathcal{D}_2$，并且$\mathcal{N}_1 \wedge \mathcal{N}_2 \Leftrightarrow \mathcal{N}$，$\mathcal{D}_1 \wedge \mathcal{D}_2 \Leftrightarrow \mathcal{D}$，若$\Psi_1$对一部分指针类型声明变量来说是合法的，$\Psi_2$对剩余的指针类型声明变量来说是合法的，则称$\Psi$可拆分成$\Psi_1$和$\Psi_2$，$\Psi_1$和$\Psi_2$可合并成$\Psi$。显然这时有$\Psi \Leftrightarrow \Psi_1 \wedge \Psi_2$。用于局部推理的规则如下：

$$\frac{\{\Psi_1 \wedge Q_1\}\ S\ \{\Psi_1' \wedge Q_1'\}}{\{\Psi_1 \wedge \Psi_2 \wedge Q_1 \wedge Q_2\}\ S\ \{\Psi_1' \wedge \Psi_2 \wedge Q_1' \wedge Q_2\}}$$（其中$\Psi_1 \wedge Q_1$和$\Psi_2 \wedge Q_2$分别都是合法的）

例如，在函数调用点，$\Psi_1 \wedge Q_1$和$\Psi_2 \wedge Q_2$分别可用来表示和被调用函数有关和无关部分。

2.4 函数构造的推理规则

PointerC 在第 1 节的文法基础上还有函数构造，本小节讨论指针逻辑有关函数构造的规则。为突出重点并简化讨论，我们做如下限制或约定。

（1）函数 f 的声明形式是 f (arg) {vardeclist stmtlist}，arg 是 f 唯一的并且是指针类型的参数，vardeclist 和 stmtlist 是 f 的变量声明表和语句表。函数调用语句的形式是 ret = f (act)，act 是实参。在下面的规则中，$v_1, \ldots, v_k$表示函数 f 声明的所有指针类型局部变量。

（2）假定函数体中有一个局部变量 res，它和用户声明的任何局部变量都不同名，函数体中的语句 return e 都变换成 res = e; return res 来进行推理，以使 return 的规则简单。

有了函数构造后，合法Ψ的检查以按作用域规则可见的全局和局部的指针类型声明变量为基础。

下面仅给出参数和返回值都是指针类型的推理规则，其它情况的规则比这些规则简单。递归函数、过程和递归过程的规则都不难在此基础上得出。

1）函数声明

$$\frac{\{\Psi \wedge \{v_1, ..., v_k, res\}_D \wedge Q\}\ stmtlist\ \{\Psi' \wedge Q'\}}{\{\Psi \wedge Q\}\ f(arg)\ \{vardeclist\ stmtlist\}\ \{\Psi' \wedge Q'\}}$$

在$\Psi \wedge Q$的基础上添加$\{v_1, \ldots, v_k, res\}_D$作为函数语句表的前断言，这是因为这些指针类型局部变量和 res 在函数体执行前都被看成悬空指针。

2）函数调用语句

$$\frac{\{\Psi \wedge \{arg\}_D \wedge Q\}arg = act\{\Psi_1 \wedge Q_1\} \qquad \{\Psi_1 \wedge Q_1\}f(arg)\ \{vardeclist\ stmtlist\}\ \{\Psi' \wedge Q'\}}{\{(\Psi \wedge \{ret\}_N \wedge Q)\ ret = f(act)\ \{(\Psi' \wedge Q')[res \leftarrow ret]\}}$$

参数为指针类型的函数调用必须体现进入被调用函数时，形参和实参在同一个指针集合中，这是通过在该规则的前提中显式增加一个指针赋值语句 arg = act 来体现的。对于返回值，也可以用类似的方法来解决。在这里是调用前限制 ret 为 NULL 指针，调用后通过代换来使 ret 出现在断言中。

3）return 语句

$$\frac{\{\Psi_0 \wedge Q_0\}v_1 = NULL\{\Psi_1 \wedge Q_1\}...\{\Psi_{k-1} \wedge Q_{k-1}\}v_k = NULL\{\Psi_k \wedge Q_k\}arg = NULL\{\Psi_{k+1} \wedge Q_{k+1}\}}{\{\Psi_0 \wedge Q_0\}\ return\ res\ \{\Pi_{k+1} \wedge (\mathcal{N}_{k+1} - \{v_1, ..., v_k, arg\}) \wedge \mathcal{D}_{k+1} \wedge Q_{k+1}\}}$$

函数终止时，指针类型局部变量$v_1, \ldots, v_k$和形参 arg 不再能访问，因此应把Ψ中有关它们的信息删去。

2.5 指针逻辑的可靠性

我们已经完成 PointerC 语言安全性的证明和指针逻辑早先版本对 PointerC 语言操作语义可靠的证明[13]，本文对指针逻辑的完善只影响指针逻辑可靠的证明细节。我们在此概述指针逻辑可靠的证明框架。

（1）建立基于栈和堆的抽象机器模型，定义 PointerC 语言的操作语义。

（2）给出断言语言在模型上的解释，包括几种访问路径集合的含义。并证明有关访问路径集合的蕴涵和等价公理在该模型上是可靠的。

（3）证明指针逻辑的每条推理规则对该模型及操作语义可靠。

3 证明实例

利用 PointerC 语言出具证明编译器的初步实现[14, 12]，我们证明了单链表、双向链表和二叉树等数据结构的一些函数的安全性或正确性。本节以带头结点的双向循环链表（图 4）的结点插入函数 insert (p, i)的安全性证明为例，展示指针逻辑的应用，其中 p 是链表的指针，i 是插入位置，链表的长度为 n。

该函数对空表和非空表采用统一的代码，但是下面的证明一开始就分成两种情况，然后每种情况又分成两种子情况，这是因为不同情况的访问路径集合不一样，很难用统一的方式来表达它们。

（1）插在表的非尾部（1<=i ∧ i<n+1）。两种子情况分别是：插在头部（i==1 ∧ i<n+1）和其它位置（1<i ∧ i<n+1）。

（2）插在表的尾部（i==n+1）。两种子情况分别是：空表（n==0 ∧ i==1）和非空表（n>0 ∧ i==n+1）。

限于篇幅，下面程序中某些程序点的断言被忽略，在所列出的断言中，很多地方只给出析取范式的一种情况。

{ dlist(p, n) ∧ n >= 0 ∧ 1<= i ∧ i <= n +1 }
struct ListNode *insert(struct ListNode *p; int i)
{ struct ListNode *q, *s;
int j;
{ ∀k:1..n.{p(->r)^k, p(->r)$^{k+1}$->l} ∧ {p, p(->r)$^{n+1}$, p->r->l} ∧{q, s}$_D$ ∧ n >= 0 ∧ 1<= i ∧ i <= n +1 }
j = 1; q = p->r;
{ (∀k:1..j-1.{p(->r)^k, p(->r)$^{k+1}$->l} ∧ {q, p(->r)^j, p(->r)$^{j+1}$->l} ∧ ∀k:j+1..n.{p(->r)^k, p(->r)$^{k+1}$->l}
∧ {p, p(->r)$^{n+1}$, p->r->l} ∧{s}$_D$ ∧ n >= 0 ∧ 1<=j ∧ j<n+1) ∨
(∀k:1..n.{p(->r)^k, p(->r)$^{k+1}$->l} ∧ {q, p, p(->r)$^{n+1}$, p->r->l}∧{s}$_D$ ∧ n >= 0 ∧ j==n+1) }
/* 循环不变式，分成两种情况 1<=j ∧ j<n+1 和 j==n+1 */
while (j < i)
{ q = q->r;
{ (∀k:1..j.{p(->r)^k, p(->r)$^{k+1}$->l} ∧ {q, p(->r)$^{j+1}$,(p->r)$^{j+2}$->l}∧ ∀k:j+2..n.{p(->r)^k, p(->r)$^{k+1}$->l}
∧ {p, p(->r)$^{n+1}$, p->r->l} ∧ {s}$_D$ ∧ n >= 0 ∧ 1<=j ∧ j<n+1) ∨ …}
/* j==n+1 的情况被忽略 */
j = j+1;
{ (∀k:1..j-1.{p(->r)^k, p(->r)$^{k+1}$->l}∧ {q, p(->r)^j,(p->r)$^{j+1}$->l}∧ ∀k:j+1..n.{p(->r)^k, p(->r)$^{k+1}$->l}
∧ {p, p(->r)$^{n+1}$, p->r->l} ∧ {s}$_D$ ∧ n >= 0 ∧ 1<=j ∧ j<n+1) ∨ …}
}
{ (∀k:1..i-1.{p(->r)^k, p(->r)$^{k+1}$->l}∧{q, p(->r)^i, p(->r)$^{i+1}$->l}∧ ∀k:i+1..n.{p(->r)^k, p(->r)$^{k+1}$->l}
∧ {p, p(->r)$^{n+1}$, p->r->l} ∧ {s}$_D$ ∧ n >= 0 ∧ 1<=i ∧ i<n+1) ∨ …}
s = malloc(ListNode);
{ (∀k:1..i-1.{p(->r)^k, p(->r)$^{k+1}$->l} ∧ {q, p(->r)^i, p(->r)$^{i+1}$->l} ∧ ∀k:i+1..n.{p(->r)^k, p(->r)$^{k+1}$->l}
∧ {p, p(->r)$^{n+1}$,p->r->l} ∧ {s} ∧ {s->r, s->l}$_D$ ∧ n >= 0 ∧ 1<=i ∧ i<n+1) ∨ …}
s->l = q->l; /* 下面仅给出 1<i ∧ i<n+1 这种子情况的断言 */
{ …∨(∀k:1..i-2.{p(->r)^k, p(->r)$^{k+1}$->l} ∧ {s->l, p(->r)$^{i-1}$,p(->r)^i->l} ∧ {q, p(->r)^i, p(->r)$^{i+1}$->l}
∧ ∀k:i+1..n.{p(->r)^k, p(->r)$^{k+1}$->l} ∧ {p, p(->r)$^{n+1}$, p->r->l}
∧ {s} ∧ {s->r}$_D$ ∧ n >= 0 ∧ 1<i ∧ i<n+1) ∨ …}

q->l->r = s; /* q->l->r 和 $p(->r)^i$ 互为别名，于是一些访问路径需要用它们的别名代替 */

{ … ∨ (∀k:1..i-2.{$p(->r)^k$, $p(->r)^{k+1}$->l} ∧ {s->l, $p(->r)^{i-1}$, q->l} ∧ {q, q->r->l}

∧∀k:i+1..n.{$q(->r)^{k-i}$, $q(->r)^{k-i+1}$->l} ∧ {p, $q(->r)^{n-i+1}$, p->r->l}

∧ {$p(->r)^i$, s} ∧{$s->r\}_D$ ∧ n>=0 ∧ 1<i ∧ i<n+1) ∨…}

s->r = q;

{ … ∨ (∀k:1..i-2.{$p(->r)^k$, $p(->r)^{k+1}$->l} ∧ {s->l, $p(->r)^{i-1}$, q->l} ∧ {s->r, q, q->r->l}

∧∀k:i+1..n.{$q(->r)^{k-i}$, $q(->r)^{k-i+1}$->l} ∧ {p, $q(->r)^{n-i+1}$, p->r->l}

∧ {$p(->r)^i$, s}∧ n>=0 ∧ 1<i ∧ i<n+1) ∨…}

q->l = s;

{ … ∨ (∀k:1..i-2.{$p(->r)^k$, $p(->r)^{k+1}$->l} ∧ {s->l, $p(->r)^{i-1}$} ∧ {s->r, q, q->r->l}

∧ ∀k:i+1..n.{$q(->r)^{k-i}$, $q(->r)^{k-i+1}$->l} ∧ {p, $q(->r)^{n-i+1}$, p->r->l} ∧ {$p(->r)^i$, q->l, s}

∧ n >= 0 ∧ 1<i ∧ i<n+1) ∨ …}

s = NULL; q =NULL;

{ … ∨ (∀k:1..i-2.{$p(->r)^k$, $p(->r)^{k+1}$->l} ∧ {$p(->r)^{i-1}$, $p(->r)^i$->l} ∧ {$p(->r)^{i+1}$, $p(->r)^{i+2}$->l}

∧ ∀k:i+1..n.{$p(->r)^{k+1}$, $p(->r)^{k+2}$->l} ∧ {p, $p(->r)^{n+2}$, p->r->l} ∧ {$p(->r)^i$, $p(->r)^{i+1}$->l}

∧ {$q, s\}_N$ ∧ n >= 0 ∧ 1<i ∧ i<n+1) ∨ …}

⇒

{ dlist(p, n+1) ∧ n >= 0 }

return p;

}

如果语句 r = insert(s, j)的前断言是 dlist(s, n) ∧ $\{r\}_N$ ∧ n >= 0 ∧ 1<= j ∧ j <= n +1，则用函数调用规则得到该语句的后断言是 dlist(r, n+1) ∧ n >= 0。

4 和分离逻辑的比较

指针逻辑和分离逻辑[15]都是 Hoare 逻辑的一种扩展，都可用来对使用共享可变数据结构（shared mutable data structure）的命令式程序进行推理。分离逻辑在汇编语言级程序验证中得到广泛应用，但是它不便于对高级语言编写的程序进行推理。在分离逻辑中，分离合取断言 P*Q 中的 P 和 Q 分别对可访问存储的不相交部分成立，这导致在和分离逻辑相关联的编程语言中，脱引用（dereference）操作序列不能出现在表达式中，而这样的操作在高级语言程序中经常出现。例如，下面的 C 程序段取自删除单链表元素的函数：

s = t->next; t->next = t->next->next; free(s);

其中的表达式 t->next->next 关联到两个分离的堆块。分离逻辑没有应用到这种表达式的推理规则。虽然把该程序段变换成

s = t->next; r = s->next; t->next = r; free(s);

后能被分离逻辑接受。但是若想对一般程序完成这种变换，则需要对程序进行别名分析。

分离逻辑的断言语言不太适合于描述带环的数据结构。本文用指针逻辑给出图 4 双向循环链表的清晰定义。而 Reynolds 用分离逻辑定义双向循环链表时[15]，首先引入带 4 个参数（头结点指针 t，尾结点指针 s，t->l 和 s->r）的双向链表段的归纳定义。在该定义中，t->l 和 s->r 同表段中其它结点的联系没有定义，以便于规范在表段上的插入和删除操作。进一步定义双向循环链表时，还需要加入 t == s->r 和 s == t->l 两个断言。显然，这种定义双向循环链表的方式不是很直观。

在扩展 Hoare 逻辑时，分离逻辑的策略可以这样理解：所有指针被假定有可能指向同一个对象，除非它们显式地表达为指向不同对象。于是分离逻辑需要引入像分离合取“*”这样的连接词。我们认为，在对指针程序进行推理时，最重要的是掌握每个程序点的有效指针相等信息，基于这些信息可以推导出两条访问路径是否互为别名，因而可以克服别名给 Hoare 逻辑带来的困难。我们扩展 Hoare 逻辑的出发点是，不同的访问

路径被认为代表不同的指针，除非能够证明它们互为别名。指针逻辑的推理规则用来从语句前那个程序点的指针信息来推导该语句后那个程序点的指针信息。

分离逻辑不像指针逻辑那样能防止内存泄漏，它的推理规则不能排除会引起内存泄漏的语句。通常，内存泄漏是指存在不能被程序访问的并且尚未释放的堆块。例如，下面程序段的前后断言可以用分离逻辑验证[16]，但是由于对赋值 x:= **nil** 采用通常的 Hoare 逻辑赋值公理，该段程序引起的内存泄漏未被发现。

$$\{\mathbf{emp}\}\quad x:=\mathbf{cons}(10);\quad \{x\mapsto 10\}\quad x:=\mathbf{nil}\quad \{\exists x.x\mapsto 10\}$$

为了解决这个问题，Reynolds 等提出了精确断言（precise assertion）概念[16]，可惜，它的定义基于抽象机器，而不是基于断言的语法特征。

指针逻辑的最大问题是，其推理规则需要使用一些比语法代换复杂得多的基本运算，使得它难以用于手工验证。若对 PointerC 语言和断言语言 lval 的语法添加下面两点限制，本质上得到分离逻辑的另一种表示形式。有了这两点限制，指针逻辑的别名计算被大大简化。

（1）PointerC 的 lval 限制为 lval ::= ident | ident->ident，并且赋值语句的左右两边不同时出现 ident->ident 的形式。

（2）断言语言的 lval 不使用 $\text{lval}(\text{->ident})^{exp}$ 的形式。

指针逻辑的这种形式区别于分离逻辑的主要地方是：

（1）存在一些高级语言特征：类型化、按域名访问而不是按偏移访问等。

（2）断言中采用访问路径集合，而不需要分离合取连接词等。

和分离逻辑比，它的优点是：

（1）断言演算的规则比分离逻辑的简单清晰。

（2）能够验证程序是否会出现内存泄漏。

5 相关工作

指针逻辑本质上是一种精确指针分析的方法，其不同于一般指针分析方法的显著优点是，它用访问路径集合来表示程序点的指针信息，用 Hoare 风格的推理规则来表示语句引起的指针信息变化，这样，它可用于高级语言程序的 Hoare 风格程序验证中。

采用访问路径集合方式来表示指针信息源于 Jonkers 用无存储语义（storeless semantics）来抽象内存地址[17]。在无存储语义中，每个动态分配的对象由一组能够到达它的访问路径来表示，整个堆的无存储表示就由所有访问路径集合的一个划分来表示。在无存储表示中，互为别名的访问路径出现在同一个集合中；最近使用无存储语义的研究[18, 19, 20]仍然采用这种冗余的表示方式。而我们只在访问路径集合中保留互为别名路径中的一条路径，通过推导来得到其它访问路径（若需要的话），并且我们的方法能较好地表示出像双向循环链表这样带环的数据结构。

在证明程序性质方面，Bornat 也使用 Hoare 风格来推导指针程序的性质[21]。他把堆看成由指针索引的一群对象，并把每个对象看成由域名索引的一组成员，这样，堆上对象成员的引用对应到两次索引。他基于域名相同与否，引入了对象成员代换公理，由此扩展了 Hoare 逻辑的赋值公理，用于证明指针程序的性质。在用高阶逻辑系统 Isabelle/HOL[22]证明指针程序性质时，Mehta 和 Nipkow 采用类似的办法。Marché 等人在他们的原型工具 Caduceus[23]中也是这样。Bornat 的方法和我们的方法都可以看成使用两次索引来解决别名，但方式上有较大区别。Bornat 所引入的公理和推理规则比较简洁，它们用于逆向的最弱前条件演算中。我们的公理和推理规则不如 Bornat 的那么简洁，但是它们以正向方式收集更多的指针信息。和 Bornat 的方法比，我们的方法有下列优势。

（1）PointerC 提供显式的释放堆空间函数 free，指针逻辑用来发现潜在的内存泄漏，但是 Bornat 的方式只能用在依赖垃圾收集器进行堆管理的场合。

（2）我们可以管理带环的数据结构，但 Bornat 假定数据结构无环。

（3）在我们的方式中，数据结构的归纳定义比 Bornat 的方式简单得多，例如单链表和二叉树的定义分别是 list(p) $\triangleq \{p\}_N \vee (\{p\} \wedge \text{list}(p\text{->next}))$和 tree(p) $\triangleq \{p\}_N \vee (\{p\} \wedge \text{tree}(p\text{->l}) \wedge \text{tree}(p\text{->r}))$。

高级类型系统也被用在检查指针程序的安全性上。例如，Smith、Walker 和 Morrisett 提出别名类型[24]，其新类型系统的主要特征是一组别名约束。别名约束描述了存储的形状，函数使用这些约束来规范它期望的存储形状。如果当前的存储不符合所给出的约束，那么类型系统保证函数不会被调用。别名类型非常类似于分离逻辑中包含空公式、指向谓词和分离合取的片段，并且也是为低级语言程序设计的。DeLine 和 Fähndrich 设计了一种编程语言 Vault，它带别名类型的一种变异[25]。Vault 提供了一种叫做类型看守的新特征，程序员可以用它来规范领域专用的资源管理协议，Vault 的类型检查器通过穷尽搜索，报告任何对协议的违例。Vault 已经用在设备驱动器的内存管理和软件协议的推理上。程序逻辑和类型系统的主要区别是：类型系统，尤其是 Vault，能较好地支持自动推理，而程序逻辑允许连接词的多样性和复杂的约束，因此有很强的表达能力。

6 结束语

本文改进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 PointerC 语言定型规则的副条件，以支持指针程序的安全性证明及其它性质的证明。指针逻辑目前还只能描述单链表、双链表和二叉树等指针相等与否有规律的数据结构，尚不能恰当描述无环有向图等指针相等与否出现部分不确定的数据结构。我们正在扩展指针逻辑以适应这样的需求。另外，有限制地允许指针算术运算，以适应编程中经常使用的 calloc 存储分配的方案也正在实施中。

References:

[1] George CN. Proof-carrying code, In Proc.24th ACM Symposium on Principles of Programming Languages. 1997, 106-119.

[2] Greg M, David W, Karl C, and Neal G. From system F to typed assembly language. In Twenty-Fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1998, 85-97.

[3] Yitzhak M, David W, and Robert H. An effective theory of type refinements. In *Proceedings of the 8*th *International Conference on Functional Programming*, 2003, 213-225.

[4] Andrew WA. Foundational proof-carrying code, In Proc. 16th Annual IEEE Symposium on Logic in Computer Science, 2001, 247-258.

[5] Dachuan Y, Nadeem AH, and Zhong S. Building certified libraries for PCC: dynamic storage allocation. Science of Computer Programming, 2004, 50(1-3):101-127.

[6] Xinyu F, Zhong S, Alexander V, Sen X and Zhaozhong N. Modular verification of assembly code with stack-based control abstractions. In Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2006, 401-414.

[7] Hongwei X. Applied type system: extended abstract. In the Post-workshop Proceedings of TYPES 2003, volume 3085 of LNCS, Springer-Verlag, 2004, 394-408.

[8] George CN and Peter L. The design and implementation of a certifying compiler. In Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation, 1998, 333-344.

[9] Christopher C, Peter L, George CN, Fred B, Mark P and Kenneth C. A certifying compiler for Java. In Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2000, 95-107.

[10] Baojian H, Yiyun C, Lin G, and Zhifang W. The PointerC programming language specification. (Technical Report) Available at: http://ssg.ustcsz.edu.cn/lss/doc/index.html.

[11] Yiyun C, Baojian H, Lin G, and Zhifang W. A Pointer Logic for Safety Verification of Pointer Programs. Chinese Journal of Computers, 2008, 31(3):372-380.

[12] Yiyun C, Lin G, Baojian H, Zhaopeng L, Cheng L, and Zhifang W. A pointer logic and certifying compiler. Frontiers of Computer Science in China , 2007,1(3):297-312.

[13] Baojian H, Yiyun C, Zhaopeng L, Zhifang W, and Lin G. Design and Proof of a Safe Programming Language. Chinese Journal of Computers, 2008, 31(4):556-564.

[14] Yiyun C, Lin G, Baojian H, Zhaopeng L, and Cheng L. Design of a certifying compiler supporting proof of program safety. In Proceedings of 1st IEEE/IFIP International Symposium on Theoretical Aspects of Software Engineering, IEEE CS press, 2007, 127-136.

[15] John CR. Separation logic: a logic for shared mutable data structures. In Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, 2002, 55-74.

[16] Peter WO, Hongseok Y, and John CR. Separation and information hiding. In Proc.31th ACM Symposium on Principles of Programming Languages, 2004, 268–280.

[17] Hans B.M. J. Abstract storage structures. In De Bakker and Van Vliet, editors, Algorithmic languages, IFIP, North Holland, 1981, 321–343.

[18] Arnaud V. Automatic analysis of pointer aliasing for untyped programs. Science of Computer Programming, 1999, 35(2):223-248.

[19] Marius B, Radu I, and Yassine L. Storeless semantics and alias logic. In Proceedings of PEPM'03, ACM Press, 2003, 55-65.

[20] Noam R, Jörg B, Thomas R, Mooly S, and Reinhard W. A semantics for procedure local heaps and its abstractions. In Proceedings of the 32th Annual ACM Symposium on Principles of Programming Languages, 2005, 296-309.

[21] Richard B. Proving pointer programs in Hoare logic. In Proceedings of the 5th International Conference on Mathematics of Program Construction, 2000, 102-126.

[22] Farhad M and Tobias N. Proving pointer programs in higher-order logic. Information and Computation, 2005, 199(1-2):200-227.

[23] Jean-Christophe F and Claude M. Multi-prover verification of C programs. In Sixth International Conference on Formal Engineering Methods, LNCS3308, 2004, 15-29.

[24] Frederick S, David W, and Greg M. Alias types. In Proceedings of the 2000 European Symposium on Programming, Berlin, 2000, 366-381.

[25] Robert D and Manuel F. Enforcing high-level protocols in low-level software. In Proceedings of the ACM Conference on Programming Language Design and Implementation, 2001, 59-69.

附中文参考文献:

[11] 陈意云、华保健、葛琳、王志芳。一种用于指针程序安全性证明的指针逻辑，计算机学报，2008，31(3):372-380。

[13] 华保健、陈意云、李兆鹏、王志芳、葛琳。一种安全语言的设计及形式证明，计算机学报，2008，31(4):556-564。