安全语言 PointerC 的设计及形式证明

华保健 陈意云 李兆鹏 王志芳 葛 琳

(中国科学技术大学计算机科学技术系 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 程序设计语言本身的安全性在高安全需求软件的设计和实现中起着基础作用.该文在用于系统级编程的安全语言的设计和性质证明方面,做了有益的尝试.作者设计了一个类 C 的命令式语言 PointerC,其主要特点在于其类型系统中包含显式的副条件(side conditions),这些副条件本质上是约束程序语法表达式值的逻辑公式.该文证明了 PointerC 语言的安全性定理,即满足这些副条件的程序,在执行时不会违反语言的安全策略.为静态推理副条件中涉及指针的命题,作者已经提出了一种指针逻辑(pointer logic),文中证明了指针逻辑对操作语义是可靠的.

关键词 软件安全;语言设计;类型系统;Hoare 逻辑;指针逻辑

中图法分类号 TP301

Design and Proof of a Safe Programming Language PointerC

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

(*Department of Computer Science and Technology, 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 The safety property of programming languages plays a fundamental role in the design and implementation of safety-critical software systems. And the authors have made investigation towards the design and proof of safe languages suitable for system programming. This paper presents the design of a C-like imperative programming language PointerC. One novelty of PointerC is that typing rules in its type system are accompanied by logic propositions which are called side conditions. And this paper proves PointerC is safe—The executions of programs will not violate the safety policy of the language, if these side conditions hold. A pointer logic, as an extension of Hoare logic, has been designed for the purpose of proving pointer-related side conditions statically. This paper presents the soundness proof for the pointer logic.

Keywords software safety; language design; type systems; Hoare logic; pointer logic

1 引 言

现代社会的正常运转越来越多依赖计算机软件系统,软件系统所引发的安全问题也日益突出.国家计算机病毒应急处理中心(CNCERT/CC)最近的一份统计报告[1]显示,仅在 2006 年,我国就有超过 74%的计算机受到计算机病毒的感染和破坏,直接经济损失数亿元;并且,近四年国内报告的安全事件数量(不包括扫描攻击),年平均增长超过 200%,去

收稿日期:2007-11-26.本课题得到国家自然科学基金(60673126)、Intel 中国研究中心资助.**华保健**,男,1979 年生,博士研究生,主要研究方向为程序验证、软件安全.E-mail: huabj@mail.ustc.edu.cn.**陈意云**,男,1946 年生,教授,博士生导师,主要研究领域为程序设计语言理论和实现技术、程序验证、软件安全.**李兆鹏**,男,1978 年生,博士研究生,主要研究方向为程序验证、软件安全、类型系统和理论.**王志芳**,男,1982 年生,博士研究生,主要研究方向为指针程序的安全性证明.**葛 琳**,女,1979 年生,博士研究生,主要研究方向为程序验证、软件安全、类型系统和理论.

年已超过 26000 件.

安全问题如此严峻，最基本的原因是计算机系统存在可被渗透(exploit)的脆弱性(vulnerability)或者称为安全漏洞(security hole).由于操作系统内核等系统软件的实现语言是 C 或 C++等不安全的编程语言，因此，编程语言本身的不安全特性，导致这些系统软件很容易出现安全漏洞.很多入侵者就是利用这些语言的下标越界、栈溢出、字符串攻击等不安全特性，对系统进行攻击或实现非授权控制.

在为应对软件安全挑战而进行的研究工作中，基于程序设计语言的软件安全(language-base software security)研究引人注目，其中典型的工作包括程序分析、类型安全的语言、高级类型系统、程序验证研究等.

基于程序分析理论和方法，很多代码查错工具被开发出来，如 LC-Lint、PREFix 和 Purify 等，它们可以用于查找和定位程序中的安全漏洞.但这类工具查错的不完备性导致其分析结果不是精确的，因此，难以单独依赖这些工具构造无安全漏洞的软件系统.

类型安全的语言，如 ML、Java 或 C♯等，提供类型安全保证；并且依赖自动垃圾收集技术，他们保证存储安全(memory safety).虽然用这些语言实现系统软件的尝试仍在继续[2]，但和 C、C++相比，这些语言运行效率的低下使得这些尝试目前仍面临难以克服的困难.细化的类型系统[3]着眼于为语言设计更高级的类型，以表达更精细的安全属性.近年来典型的研究包括依赖类型(dependent type)[4-5]、应用类型(applied type)[6]、单元素类型(singleton type)[7]等.尽管这些系统扩展了传统类型系统的表达能力，但它们都受到共同问题的困扰——静态类型系统有限的表达能力，因此，这些研究都仅在整数域等几个简单论域上取得了部分结果.

程序验证和公理语义[8-9]通过从程序中抽取逻辑命题来进行程序性质证明，其中程序性质包括安全属性和部分正确性(partial correctness)等.但程序验证一般需要把写程序规范和写代码本身结合起来，这增大了程序设计的难度；而且，由于不存在证明逻辑命题的通用算法，很多程序的证明无法自动完成.

针对软件安全研究面临的挑战和现有研究工作的不足，我们在适用于系统级编程的安全语言设计和实现以及性质证明方面，做了有益的尝试.本文工作的主要目的是探索一种类型系统和逻辑系统相结合保证语言安全性的静态机制，为此，我们设计了一个保留 C 指针特性的类 C 语言 PointerC.和 C 语言的类型系统相比，PointerC 的新颖之处在于它的定型规则(typing rules)中除了包括普通的定型断言外，还包括作为副条件的逻辑命题，这些副条件给出了对程序语法表达式值的约束.我们形式定义了语言所要满足的安全策略，并且，基于这些安全策略，证明了 PointerC 语言是安全的.

为了静态检查这些副条件，我们在前期工作中已经为 PointerC 语言设计了一种指针逻辑[10-11]，它是 Hoare 逻辑的一种拓展，本质上是一种精确的指针分析工具.指针逻辑可用来从前向后收集各指针是空指针、悬空指针还是有效指针(effective 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附加条件，以支持对指针程序的安全性验证和其它性质的验证.我们证明了指针逻辑对 PointerC 语言的操作语义是可靠性的.

和已有研究工作相比，本文工作的主要特色和贡献在于：

(1)探索了一种类型系统和逻辑系统相结合保证语言安全性的静态机制.从程序设计角度看，和依赖类型等方式相比，使用副条件更有利于程序员的理解.

(2)本文证明了 PointerC 语言是安全的，该证明基于一组形式定义的安全策略.基于安全策略进行证明可以保证语言的安全属性，例如，我们证明了合法的 PointerC 程序不会出现内存泄露(memory leak).

(3)我们证明了指针逻辑对 PointerC 语言的操作语义是可靠的.和 Hoare 逻辑的可靠性证明相比，指针逻辑的可靠性证明具有明显的区别.

本文第 2 节介绍所设计的源语言 PointerC，包括它的语法、类型系统、操作语义，在形式定义安全策略后，本节证明 PointerC 语言是安全的；第 3 节概要介绍指针逻辑，重点介绍指针逻辑的可靠性证明；第 4 节与相关工作进行比较；最后给出结论和进一步的工作.

2　源语言 PointerC 的设计及其安全性证明

本文所讨论的带有副条件的类型系统以及为证明这些副条件而设计逻辑系统来保证语言安全性的

静态机制，并不只限于适用某种特定的语言，但为考察这种静态机制在系统级编程语言上的有效性，我们设计了一个类 C 的命令式语言，其主要出发点是：首先，C 语言不是类型安全的，在高安全需求的应用中，为它提供更强的安全保障机制非常必要；其次，C 作为最重要的系统程序设计语言之一，其代码运行效率非常关键，因此尝试以类型系统和逻辑系统相结合的静态机制来保证语言的安全，而不是依赖动态检查，可以不牺牲代码的运行效率；最后，本文工作的主要目标是研究保证指针操作安全性的静态机制——这也是程序分析和程序验证的难点，而 C 灵活的指针特性为此提供了平台.

本节首先概要介绍我们所设计的类 C 源语言 PointerC，包括它的语法、类型系统、操作语义；然后给出 PointerC 语言的安全性证明. 篇幅所限，无法列出的细节包括在相应技术报告[12]中.

2.1 PointerC 的语法

PointerC[12]是我们设计的一个类 C 的命令式语言，它保留了 C 语言核心语法特征：结构体声明、全局变量声明、函数定义、控制流语句、表达式等. 因 PointerC 的语法和 C 接近，这里不再赘述. PointerC 类型系统的语法特征将在 2.2 节讨论.

和其它保证系统级编程语言安全性的工作[13-14]不同，PointerC 支持动态存储的手工管理，保留了 C 高效存储管理的特征，但这对安全性的静态检查提出了挑战. 因此，除静态类型系统外，我们还使用指针逻辑系统和定理证明的方法，它们将在第 3 节讨论.

2.2 PointerC 的类型系统

类型系统是一种静态检查程序的语法方法，它一般根据程序短语的值的类别来对程序短语进行分类，排除程序中上下文有关的错误. 尽管类型系统已经成为现代程序设计语言最流行的静态机制之一，但现有语言的类型系统并不令人满意. C 和 C++ 的类型系统不是可靠的，其类型系统不能对程序行为提供安全保证；ML、Java、C♯等语言的类型系统虽然是可靠的，但对于表达式值或程序资源有关的部分，要依赖低效的运行时动态检查(尽管某些编译器能够静态移除部分检查).

PointerC 包括 C 语言大部分的类型，如整型、指针、数组、结构等. 类型强制很容易导致难以排除的程序漏洞，如把整型数据用作指针等，因此 PointerC 禁止进行类型强制；同样，通常难以静态保证多次对同一联合(union)数据的读取是一致的，联合类型也被禁止. 这些限制不影响 PointerC 的表达能力和实际程序设计的可用性，例如，借鉴 C++ 和 Java，PointerC 引入了布尔类型，代替 C 对整型进行类型强制来模拟布尔类型的做法. 部分 C 类型被忽略了，如浮点类型、字符类型等，因为它们和本文内容关系不大.

PointerC 类型系统[12]的定型规则除了包括通常的定型断言外，还包括称为副条件的逻辑命题，这些逻辑命题用于表达对值的约束(constraints). 和已有的研究相比，PointerC 采用的这种结合副条件的方式有显著的优点：首先，类型系统简单明了，避免了使用依赖类型等带来的复杂性；其次，对这些命题的证明可以使用指针逻辑系统(将第 3 节讨论)和自动定理证明工具.

图 1 列出了 PointerC 类型系统部分代表性定型规则，其它规则可参考技术报告[12]. 其中，Γ 是定型环境(typing context)，由变量和类型的二元组列表组成. 位于横线上下的是通常的定型断言 $\Gamma\vdash e:\tau$，即表达式 e 在定型环境 Γ 下的类型为 τ. 为和通常的定型断言区别，副条件被写在一对大花括号内，并置于定型规则右侧.

$$\frac{\Gamma\vdash e:\tau^*}{\Gamma\vdash {}^*e:\tau}\{e\in\textbf{effective}\} \quad \text{(Deref)}$$

$$\frac{\Gamma\vdash l:\tau[N] \qquad \Gamma\vdash e:\ \text{int}}{\Gamma\vdash l[e]:\tau}\{0\le e<N\} \quad \text{(ArrayField)}$$

$$\frac{\Gamma\vdash lval:\tau^*}{\Gamma\vdash lval=\text{NULL};:unit}\{\neg\,\text{leak}(lval)\} \quad \text{(AsnNull)}$$

图 1 PointerC 的代表性定型规则

图 1 的 3 条规则说明了副条件的 3 种典型使用. 其中，对数组下标越界检查的规则 ArrayField 比较简单，只需检查相应的下标表达式 e 在合法的取值区间内；规则 Deref 给指针引用定型，除了要求 e 在定型环境 Γ 下具有指针类型 τ^* 外，还规定表达式 e 的值必须是有效指针(即有具体的指向对象，见第 3 节)，由于这个条件涉及 e 的值，因此它由副条件 $e\in$ **effective** 给出；规则 AsnNull 给指针赋值语句定型，它说明了另外一类重要的副条件：当赋 NULL 值给某个左值 $lval$ 时，必须确保不产生内存泄漏(即确保 $lval$ 原来指向的内存块在赋值后不会访问不到)，这个副条件用逻辑命题 $\neg\,\text{leak}(lval)$ 表示，显然内存分配语句也会使用到这个副条件[12].

需要指出，定型规则规定了该语句需满足的定型条件以及副条件；但没有规定检查或证明这些副条件的方式. 在已有的研究中，传统类型系统的表达

能力不足以推导或证明关于值(特别是指针类型变量的值)的属性,这是我们设计指针逻辑并以逻辑系统来证明这些副条件的部分原因.

2.3 PointerC 的操作语义

PointerC 操作语义的定义基于一个为 PointerC 设计的抽象机器模型 $\mathcal{M}$[12]. PointerC 包括声明变量、动态分配的指针类型变量、函数定义、函数调用等语言特性,因此,它的抽象机器状态 $\mathcal{M}$ 由五元组构成:$\mathcal{M}=(S;H;G;R;K)$. 其中,存储器(Store)S: $Var\rightarrow Value$ 把程序中的声明变量映射到值;堆 H: $Addr\rightarrow Value$(Heap),把抽象的堆地址映射到值;G 是全局数据区,对程序所有的代码可见;归约栈(evaluation stack)K 规定语句和表达式的归约顺序;R 是调用栈(call stack),规定函数调用和返回的控制转移(负责保存调用者的归约控制栈 K 和存储器 S).

PointerC 的操作语义用抽象机器模型 $\mathcal{M}$ 上的状态转移规则来描述. 程序的执行总是从一个初始机器状态 $\mathcal{M}$ 开始,根据当前语句,使用状态转移规则到达后状态 $\mathcal{M}'$. 在每个程序点,都至多有一条状态转移规则可以使用;如果没有任何归约规则可以使用,则产生错误.

在 PointerC 的操作语义的规则中,每个归约状态的形式是

$$(S;H;G;R;K)\triangleright a,$$

其中 $(S;H;G;R;K)$ 是当前抽象机状态,a 是当前正在被归约但还未完成的某个语法对象(如表达式、语句等). $(S;H;G;R;K)\triangleright a$ 称为在当前机器状态上的一个"结构"(configuration). PointerC 操作语义的归约规则是结构间的二元关系"$\Rightarrow$":

$$(S;H;G;R;K)\triangleright a \Rightarrow (S';H';G';R';K')\triangleright a',$$

即在抽象机状态 $(S;H;G;R;K)$ 上归约语法对象 a 后,得到新的抽象机器状态 $(S';H';G';R';K')$,并在这个新状态下继续归约剩余的 a'. 典型的情况是 a 是语句序列,而 a' 是该语句序列的尾;特别的,当 a 是单条语句时,a' 是空语句,此时我们用特殊符号 ε 来表示 a'. 抽象机从初状态和完整程序开始执行,要么最终把所有的程序执行完,停在某个终状态;要么陷入死循环,执行不终止. 篇幅所限,此处略去 PointerC 操作语义的完整规则,细节可参考技术报告[12].

在本文接下来的讨论中,存储器 S 和堆 H 的性质最重要,为陈述简单清晰起见,下面将使用只包括存储器和堆的简化机器状态 $\mathcal{M}=(S;H)$,省略其它部分. 需要指出,实际的证明[15]使用的都是完整的机器状态定义.

2.4 PointerC 的安全性定理

给出 PointerC 的类型系统和操作语义后,本节证明 PointerC 语言是安全性的. 在给出形式证明之前,首先讨论语言安全性的概念,这个概念基于对程序运行时错误的区分.

程序运行时错误可分为两类. 一类程序运行时错误称为会被捕获的错误(trapped error),如除数为零错误等;在许多计算机系统结构上,这样的错误使得计算立即停止. 然而,还有一些难以捉摸的错误,它们引起数据遭破坏而不立即出现征兆,典型的例子是 C 中的数组越界访问. 这类错误当时未引起注意,而后引发难以预见的行为,因此,这类错误叫做不会被捕获的错误(untrapped error). 一个程序段是安全的,如果它不可能引起不会被捕获错误的出现. 所有程序段都是安全的语言叫做安全语言(safe language).

PointerC 的安全性定理的主要目的是证明:经过类型检查且满足副条件的代码,在执行过程中,不会出现不会被捕获的错误. 为此,我们把不会被捕获的错误形式化为特定的安全策略(safety policy),然后基于这些安全策略证明 PointerC 的安全性.

具体讲,该定理的描述和定理的证明分成如下两个步骤:

(1) 在 PointerC 的机器模型和操作语义上形式描述安全策略 $\mathcal{SP}$;安全策略 $\mathcal{SP}$ 实际上是定义在抽象机器模型 $\mathcal{M}=(S;H)$ 上的逻辑谓词,记作

$$\models \mathcal{SP}(\mathcal{M}),$$

需要指出,我们使用符号 $\models$ 来强调安全策略是在机器模型上定义的语义性质.

(2) 证明形式描述的安全策略 $\mathcal{SP}$ 对于程序的操作语义 $\Rightarrow$ 是一个不变式(invariant):即对于机器状态 $\mathcal{M}=(S;H)$ 和满足定型规则及所有副条件的语句 s,有

$$(\models \mathcal{SP}(\mathcal{M})\wedge(\mathcal{M}\triangleright s\Rightarrow \mathcal{M}'\triangleright s'))\supset \models \mathcal{SP}(\mathcal{M}').$$

非形式地说,第二点表明如果安全策略 $\mathcal{SP}$ 在某个前抽象机器状态 $\mathcal{M}$ 上成立,并且满足相应定型规则(包括副条件)的程序 s 在 $\mathcal{M}$ 上执行后得到新的机器状态 $\mathcal{M}'$,则安全策略 $\mathcal{SP}$ 在 $\mathcal{M}'$ 上仍然成立. 这样,不难得到结论:只要安全策略 $\mathcal{SP}$ 在抽象机初始状态 $\mathcal{M}_0$ 上成立,它将在程序执行中的所有机器状态 $\mathcal{M}$ 上成立.

第一点中,PointerC 的基本安全策略包括三个

方面:无空指针或悬空指针引用(也禁止把它们用作free函数的参数)、无数组下标越界、无内存泄漏.前两个比较简单,可参考技术报告和Coq实现[15].最复杂的是最后一个,为此,我们引入机器状态$\mathcal{M}=(S;H)$上的*NoLeak*谓词

$$NoLeak(S;H)\triangleq\forall l\in Dom(H).(\exists p.V(p)=l),$$

该谓词表达当前抽象机状态无内存泄漏的性质,即对抽象机状态$\mathcal{M}=(S;H)$,若对堆H上的任意地址l,都存在一个从存储器S出发的指针p指向,则该机器状态$\mathcal{M}$无内存泄漏.其中,对于给定的$\mathcal{M}=(S;H)$,函数V

$$V(S,H,p)=\begin{cases}S(p), & \text{若 } p \text{ 是声明变量}\\ H(V(S,H,q))(n), & \text{若 } p=q\rightarrow n\end{cases},$$

基于对p的语法形式归纳返回指针p的值.注意,在*NoLeak*谓词的定义中,为了简单,我们略去了V的机器状态参数$(S;H)$.

和V的定义类似,在PointerC的安全性定理证明中要用到求一个指针初始声明变量的*Root*操作:

$$Root(p)=\begin{cases}p, & \text{若 } p \text{ 是声明变量}\\ Root(q), & \text{若 } p=q\rightarrow n\end{cases}.$$

对无内存泄漏的安全策略,我们证明了它对语句的执行是一个不变式:如果在无内存泄漏的状态下执行所有副条件都成立的程序,执行后得到的状态也一定无内存泄漏.即证明了以下定理.

定理1(无内存泄漏). 对任意机器状态$\mathcal{M}=(S;H)$及语句s,其中s是良类型的且满足定型规则中所用到的所有副条件,若$\models NoLeak(S;H)$且存在归约$(S;H)\triangleright s\Rightarrow(S';H')\triangleright s'$,则$\models NoLeak(S';H')$.

证明. 基于对语句s的结构归纳.可按对语句s的定型是否使用了副条件把语句分为两类.如果s的定型规则中没有用到副条件,则证明是简单的.复杂的是s的定型规则中使用了副条件的情况.我们以语句s是指针赋值语句$p=$NULL为例说明证明思路.图1中的规则AsnNull给语句$p=$NULL定型,它包括的副条件$\neg$leak(p)说明

$$\exists p'.(Root(p)\neq Root(p'))\wedge(V(p)=V(p')).$$

对p的语法形式进行归纳:如果p是声明变量,则由条件$\models NoLeak(S;H)$可知

$$\forall l\in Dom(H).(\exists q.V(q)=l),$$

因此,在赋值前有

$$\exists l\in Dom(H).(V(p)=V(p')=l),$$

赋值后,显然有$\exists l\in Dom(H).(V(p')=l)$,并且新的状态$S'=S[p\mapsto\text{NULL}]$且$H'=H$,则可得到

$$\forall l\in Dom(H').(\exists q.V(q)=l),$$

此即$\models NoLeak(S';H')$.对其它情况的证明类似.对其它定型使用了副条件的语句同样可以证明.

证毕.

不同于通常的手工证明方法,为严格起见,该定理的完整描述和形式证明都已在证明辅助工具Coq中完成,细节可参考技术报告和Coq实现[15].

对一般C程序而言,完全静态推理其无内存泄漏的性质是困难的,在PointerC中加入的一些限制(如禁止取地址操作&等)使得静态推理这类性质成为可能.

3 指针逻辑及其可靠性证明

2.2节介绍了定型规则中的副条件,这些副条件表达了对程序语法表达式值的约束.对这些副条件的检查可以使用在Java等语言中流行的动态检查方法,由编译器在需要检查副条件的程序点插入额外指令;也可以使用静态检查的方法,在编译期间证明副条件.我们设计了一种指针逻辑[10-11],它以静态方式检查和保证这些副条件.和动态方法比较,使用指针逻辑这种静态机制的优点是:首先,程序漏洞可在开发阶段被捕获;其次,可以提高目标代码的运行效率,这在许多应用中至关重要.当然,指针逻辑的使用给静态证明提出了挑战,因此,我们使用自动定理证明工具,而不是只依赖类型检查器.

本节简要介绍指针逻辑的设计思想,然后给出指针逻辑的可靠性证明.

3.1 指针逻辑的设计

指针逻辑本质上是对Hoare逻辑的一种扩展,它的设计主要基于以下考虑:

(1)若在程序点能区分有效指针(effective pointers)、空指针、悬空指针(dangling pointers),并且知道有效指针之间是否相等,则不但能判断有关指针的操作是否安全,还可得出经过这步操作后指针信息的变化.

(2)用Hoare逻辑风格的推理规则来表达指针信息的变化,就可以把这些信息用于证明程序的其它性质,同时把程序分析、类型系统和程序验证联系起来,有更好的表达能力.

指针逻辑的设计难点在于指针类型和动态变量,并且有关指针性质的推导同程序其它性质的推导有很大的区别.例如,在free(p)语句执行后,为防止悬空指针引用,在p被重新赋值指向别的对象之

前，不能用 p 或与 p 相等的指针去访问 p 原来所指向的内存；再如，在对有效指针 q 赋值时，为防止出现内存泄漏，需要知道是否存在有与 q 相等的指针.

为解决这些难点，以使静态推理成为可能，指针逻辑在每个程序点区分三类指针：有效指针、空指针、悬空指针，分别用指针集合 $\Pi,\mathcal{N},\mathcal{D}$ 来表示它们，这些集合实质上只是相应逻辑命题的一种语法美化(syntactic sugar). 例如，对有效指针集合 $\Pi=\{s,t\}\wedge\{p,q\}$，Π 实际上等价于逻辑命题 $(s=t)\wedge(s\in\textbf{effective})\wedge(p=q)\wedge(s\in\textbf{effective})\wedge(s!=p)$. 可以看出，使用指针集合这样的语法美化，大大简化了记号. 下面用 Ψ 表示 $\Pi\wedge\mathcal{N}\wedge\mathcal{D}$.

基于这些记号，图 2 给出了用指针逻辑推理单链表插入代码片段的例程(为了和断言区别，代码进行了缩进). 其中，p,q 是具有如下单链表类型的指针：

```
struct List{
  int data;
  struct List * next;
}
```

```
{p}∧{p->next}∧{p->next->next, q}N
  q=alloc(struct List);
{p}∧{p->next}∧{p->next->next}N∧{q}∧{q->next}D
  q->next=p->next;
{p}∧{p->next, q->next}∧{p->next->next}N∧{q}
  p->next=NULL;
{p}∧{q->next}∧{q->next->next, p->next}N∧{q}
  p->next=q;
{p}∧{q->next}∧{q->next->next}N∧{q,p->next}
```

图 2　用指针逻辑推理单链表插入代码

图 2 的第 1 行断言给出指针在当前程序点的状态：p 和 $p\text{—>}next$ 是有效指针，并且指向不同的单元；而 $p\text{—>}next\text{—>}next$ 和 q 都是空指针. 第 2 行分配一个新的链表节点，并把它赋值给 q；这个操作所引起的指针变化反映在第 3 行的断言中，即 q 被加入到一个单元素的有效集合，而 $q\text{—>}next$ 被加入到悬空指针集合 $\mathcal{D}$ 中. 按这种方式，可以推理图 2 中剩余的代码.

图 3 列出了图 2 例子中使用到的部分指针逻辑推理规则[10-11]. 这些规则体现了指针逻辑的三个主要特点：首先是规则形式和 Hoare 逻辑规则的一致性. 以 Malloc 规则为例，它描述了进行内存分配并给变量 p 赋值的演算过程：如果前面的指针集合是 $\Pi\wedge\mathcal{N}\wedge\mathcal{D}$，并且 p 属于 $\mathcal{N}$ 集合(即 p 的值为 NULL. 这里使用符号 $<:$ 表示集合的属于关系，而没有使用集合符号 $\in$，是因为 $<:$ 比 $\in$ 表达了一种更广意义上的属于关系，细节参见文献[10])，则赋值后，应把 p 作为单元素的有效指针集合(因为只有 p 一个指针指向刚分配的单元)，而且要把该单元的所有指针域都加入悬空指针集合中(都是悬空的)，最后把 p 从空指针集合中删除. 其它规则的含义类似，只是加入了更多的基本函数和谓词来处理较复杂的情况[10-11]. 其次，这种从前向后的推理形式，使得对验证条件(verification condition)的计算也可使用最强后条件演算的方式[10]；最后，规则形式的一致性，方便我们借鉴传统 Hoare 逻辑可靠性证明的方式，证明指针逻辑的可靠性.

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

$$\frac{p<:\Pi\ q<:\mathcal{N}\neg\text{leak}(p)}{\{\Pi\wedge\mathcal{N}\wedge\mathcal{D}\}\ p=q\{\Pi/p\wedge(\mathcal{N}\backslash p\cup\{p\})\wedge\mathcal{D}\backslash p\}}\quad(\text{Asn\_EN})$$

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

$$\frac{p<:\mathcal{N}}{\{\Pi\wedge\mathcal{N}\wedge\mathcal{D}\}\ p=alloc(T)\ \{(\Pi+p)\wedge\mathcal{N}/p\wedge(\mathcal{D}\cup\{p\text{->}r_1,\cdots,p\text{->}r_n\})\}}\quad(\text{Malloc})$$

图 3　指针逻辑的代表性规则

3.2　指针逻辑的可靠性证明

我们证明了指针逻辑相对 PointerC 操作语义是可靠性的，所用方法同 Hoare 逻辑可靠性证明没有本质区别，也是先在机器模型和操作语义上给出断言的语义解释，然后证明每条公理和推理规则对操作语义都是可靠的[15]. 即要证明，若断言 $\{P\}s\{Q\}$ 在指针逻辑中可以证明，则在任何满足前断言 P 的状态 $\mathcal{M}$ 下执行程序段 s 后，得到的新状态 $\mathcal{M}'$ 一定满足后断言 Q. 我们证明了如下的定理.

定理 2(指针逻辑可靠性定理).　若 $(S;H)\models P$，且 $(S;H)\triangleright s\Rightarrow(S';H')\triangleright\varepsilon$，则有 $(S';H')\models Q$.

证明.　首先用定理证明工具 Coq 表示 PointerC 语法、类型系统、操作语义等；然后在 Coq 中形式化指针逻辑的断言和定理；最后在 Coq 中基于对语句 s 的结构归纳完成该证明. 需要注意，在操作语义的规则中得到的结构的形式是 $(S';H')\triangleright\varepsilon$，即整个语句归约到了空语句 ε.　　证毕.

限于篇幅，这里无法给出语义函数 $\models$ 的完整定义，细节可参考技术报告[15]. 和对 PointerC 语言的安全性证明相同，为更严格，证明过程在辅助工具 Coq[16] 中机械化完成，而非手工进行. 下面给出该证明的一些细节步骤和定理，对该证明的更详细的讨论(包括 PointerC 语法、类型系统和操作语义相应的 Coq 表示，定理在 Coq 中的表达等)可参考技术报告[15].

指针类型赋值语句的推理规则和 Hoare 逻辑赋值公理完全不一样,另外还有新增的 malloc 和 free 语句的规则,所以,指针逻辑可靠性的证明实际上分成如下 3 个步骤:

(1)证明基本语句 s(指针赋值、调用、malloc、free 等)的推理规则

$$\frac{P(\Psi)}{\{\Psi\}\ s\ \{\Psi'\}}$$

是可靠的,其中 $P(\Psi)$是指该规则中以 Ψ 为参数的前提.若对 s 语句定型时所用各定型规则的附加条件是 $C_1,\cdots,C_n$,则要证明(其中$\vdash$是通常的语法可证符号)如下引理.

引理 1(指针逻辑可靠性定理,指针部分). 若

$$\vdash \Psi \wedge P(\Psi) \Rightarrow C_1 \wedge \cdots \wedge C_n,$$

则对任何满足$(S;H) \models \Psi \wedge P(\Psi)$的状态$(S;H)$,有

$$(S;H) \models C_1 \wedge \cdots \wedge C_n.$$

且若$(S;H) \triangleright a \Rightarrow (S';H') \triangleright b$,则$(S';H') \models \Psi'$.

证明. 基于对语句 s 的结构归纳可证.

(2)证明整型赋值语句 a 的推理规则

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

($y_1,\cdots,y_n$构成 $closure(x)$的所有成员[9])

是可靠的.若对语句 $x=e$ 定型时所用各定型规则的附加条件是 $C_1,\cdots,C_n$,则要证明如下引理.

引理 2(指针逻辑可靠性定理,整型部分). 如果

$$\vdash \Psi \wedge P(\Psi) \Rightarrow C_1 \wedge \cdots \wedge C_n,$$

则对任何满足$(S;H) \models \Psi$的状态$(S;H)$,有

$$(S;H) \models C_1 \wedge \cdots \wedge C_n.$$

且若$(S;H) \triangleright s \Rightarrow (S';H') \triangleright s'$,则$(S';H') \models \Psi'$.

证明. 基于对语句 s 的结构归纳可证.

(3)复合、条件和循环语句的规则以及推论规则等的可靠性的证明.它们完全与 Hoare 逻辑可靠性中相应规则的证明方式相同.

4 相关工作

C 语言安全性问题主要是出于效率和编程灵活性的考虑.随着类型系统理论的发展和程序分析技术的成熟,最近几年,一些项目尝试改造 C,或设计它的安全变种. CCured[13] 是对 C 程序进行分析和变换的框架,它使用静态程序分析排除程序漏洞.为了保证分析的可行性,CCured 提出了两种方案:(1)允许程序员在 C 代码上加入额外标注作为给静态分析器的提示;(2)插入动态检查代码.CCured 的类型系统是不可靠的,所以为保证语言的安全性,仍需依赖动态检查机制,这在很多情况下降低了系统效率.而 PointerC 的安全性保证了通过静态验证的程序不需要进行动态检查.另外,与本文工作不同,CCured 使用垃圾收集器,回避了指针推理的难点;而 PointerC 支持 C 的动态存储管理的机制,并且能够用指针逻辑推理其安全性.

Cyclone[14] 为 C 设计新的类型系统,重点是加入更多高级语言特性和类型.本文工作和 Cyclone 的主要区别在于 Cyclone 使用基于区域的存储管理(region-based memory management),基于区域的存储管理除了自身有较大局限性外,还要求程序员在代码中加入对这些机制的支持代码,因此,难以改写现有的 C 遗留代码以适应 Cyclone 的要求.而 PointerC 的副条件生成由编译器自动完成.

Xi 等人的 ATS(Applied Type System)项目[6] 和本文工作类似,也把 Hoare 逻辑形式的断言引入类型系统,从而在类型系统上模拟 Hoare 逻辑的部分推理.但本文工作和 ATS 的重要区别在于 ATS 的类型系统需要程序员显式标注,加大了理解和使用难度;而且,由于类型检查的表达能力有限,ATS 仅能推理整数等较简单论域的性质.而本文工作使用单独的指针逻辑系统对副条件进行推理,从而能够证明更强的程序安全属性.

ESC(Extended Static Checking)[18] 和 Spec♯[19] 等将 Hoare 逻辑推理方法用于检查 Modular-3、Java、C♯等程序.本文工作与这些研究的本质区别在于:ESC 等只是程序查错工具,而不是程序设计或验证工具,它们没有给出可靠性证明.而本文工作使用指针逻辑来推理和证明验证条件,并证明了该指针逻辑的可靠性;另外,本文证明了 PointerC 语言是安全性的.

Caduceus[20] 是一个 C 程序验证框架,它支持多个定理证明器作为后端.本文工作和 Caduceus 的重要区别在于:Caduceus 的可靠性没有经过严格证明;并且 Caduceus 仅支持 C 很简单的一个小子集,对被验证程序提出了很多限制,例如,Caduceus 禁止程序中出现别名.

Clight[21] 是 Xavier Leroy 等人为在 Coq 中验证编译器的前端而设计的一个小类 C 的语言.和本文工作类似,Clight 的语义也被形式在 Coq 中.但和本文工作不同,Clight 仅包含了 C 的核心语法结构,完全去掉了动态存储特性,这限制了 Clight 的表达能力.而 PointerC 语言保留了 C 的动态存储管理机

制，适合验证更实际的指针程序.

5　结论和进一步的工作

本文在系统级安全语言设计和实现以及性质证明方面做了有益尝试. 我们采用一种类型系统和逻辑系统相结合的静态机制保证语言的安全性. 为了考察这种机制的实际可行性，我们设计了一个类 C 的命令式语言 PointerC，它的新颖之处在于定型规则中包括副条件断言. 基于形式定义的安全策略，本文证明定理 PointerC 的安全性. 为了静态推理指针类型的断言，我们在前期工作中提出了指针逻辑，以解决 Hoare 逻辑处理别名问题所面临的困难. 本文证明了指针逻辑相对 PointerC 的操作语义是可靠的.

我们正在研究放宽对 PointerC 指针运算的限制，有限制地允许指针算术，以适应编程中经常使用的 calloc 存储分配等. 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将是进一步研究的内容.

参　考　文　献

[1] CNCERT/CC. 2006 Annual report. Available at http://www.cert.org.cn/articles/docs/common/2007042923284.shtml

[2] Galen Hunt, James Larus. Singularity: Rethinking the software stack. Operating Systems Review, 2007, 41(2): 37-49

[3] Mandelbaum Y, Walker D, Harper R. An effective theory of type refinements//Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming. Uppsala, Sweden, 2003: 213-225

[4] Xi H. Imperative programming with dependent types//Proceedings of the 15th IEEE Symposium on Logic in Computer Science. Washington, DC: IEEE Computer Society, 2000: 375-387

[5] Xi H, Pfenning F. Dependent types in practical programming//Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. San Antonio, Texas, USA, 1999: 214-227

[6] Xi H W. Applied type system (extended abstract)//Post-Workshop Proceedings of the TYPES 2003. Lecture Notes in Computer Science 3085. Berlin: Springer-Verlag, 2004: 394-408

[7] Smith F, Walker D, Morrisett J G. Alias types//Proceedings of the 9th European Symposium on Programming Languages and Systems. Lecture Notes in Computer Science 1782. London: Springer-Verlag, 2000: 366-381

[8] Hoare C A R. An axiomatic basis for computer programming. Communications of the ACM, 1969, 12(10): 576-580

[9] Necula G C. Proof-carrying code//Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Paris, France, 1997: 106-119

[10] Chen Y Y, Ge L, Hua B J, Li Z P, Liu C, Wang Z F. A pointer logic and certifying compiler. Frontiers of Computer Science in China, 2007, 1(3): 297-312

[11] Chen Yi-Yun, Hua Bao-Jian, Ge Lin, Wang Zhi-Fang. A pointer logic for safety verification of pointer programs. Chinese Journal of Computers, 2008, 31(3): 372-380(in Chinese)
(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)

[12] Hua B J, Chen Y Y, Ge L, Wang Z F. The PointerC programming language specification. University of Science and Technology of China: Technical Report TR-1001-06, 2006

[13] Necula G C, McPeak S, Weimer W. CCured: Type-safe retrofitting of legacy code//Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages. New York, NY: ACM Press, 2002: 128-139

[14] Jim T, Morrisett J G, Grossman D, Hicks M W, Cheney J, Wang Y. Cyclone: A safe dialect of C//Proceedings of the General Track: 2002 USENIX Annual Technical Conference. Berkeley: USENIX Association, 2002: 275-288

[15] Hua B J, Chen Y Y, Li Z P. The pointer logic soundness proof and PointerC safety proof. University of Science and Technology of China: Technical Report TR-1101-07, 2007

[16] The Coq development team. The Coq proof assistant reference manual Version 8.0. Paris, France, 2004

[17] Ge Lin, Chen Yi-Yun, Hua Bao-Jian, Li Zhao-Peng, Liu Cheng. Automatic generation of formal specifications in assembly code certification. Mini-Macro Systems, to appear(in Chinese)
(葛琳，陈意云，华保健，李兆鹏，刘诚. 汇编代码验证中的形式规范自动生成. 小型微型计算机系统，已录用)

[18] Flanagan C, Leino K R M, Lillibridge M, Nelson G, Saxe J B, Stata R. Extended static checking for java//Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI′2002). New York: ACM Press, 2002: 234-245

[19] Barnett M, Leino K R M, Schulte W. The Spec# programming system: An overview//Proceedings of the CASSIS′04. LNCS 3362, Marseille, France, 2005: 49-69

[20] Filliâtre J-C, Marché C. Multi-prover verification of C programs//Proceedings of the ICFEM. LNCS 3308. Seattle, USA, 2004: 15-29

[21] Sandrine Blazy, Zaynah Dargaye, Xavier Leroy. Formal verification of a C compiler front-end//Proceedings of the Symposium on Formal Methods (FM′06). LNCS 4805. Hamilton, Canada, 2006: 460-475

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

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.

LI Zhao-Peng, born in 1978, Ph. D. candidate. His research interests include program verification, software safety and security, type system and theory.

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

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

Background

This research is supported by the National Natural Science Foundation of China (Verification and Compilation of Software Safety, grant No. 60673126), and the Intel China Research Center. These projects study the theory and method of program verification and compilation of software safety.

Program verification is an important method to improve software reliability. And axiomatic semantics and certifying compilation are also well-populated research fields with decades of history. In the past decade, the success of Proof-Carrying Code (PCC) renews researchers' interests in program verification. One important reason for PCC's success is that by concentrating on safety properties, instead of partial correctness, the safety proofs could be generated automatically by a certifying compiler. Their research group has focused on research of scaling the technology of PCC to the verification and compilation of more realistic programming languages and more practical safety policies.

This paper presents their research progress on the design and safety proof of the PointerC programming language. More specifically, they have designed a C-like imperative language PointerC which retains explicit memory allocation and deallocation features. By introducing side conditions, PointerC's type system is kept expressive yet simple. The safety proof for PointerC is presented, which is based on the technique of invariants. To reason PointerC programs statically, a pointer logic, as an extension of Hoare logic, has been proposed in their previous work, and this paper presents the soundness proof of the pointer logic.