ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.21, No.2, February 2010, pp.334−343
doi: 10.3724/SP.J.1001.2010.03783
© by Institute of Software, the Chinese Academy of Sciences. All rights reserved.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

处理指针相等关系不确定的指针逻辑*

梁红瑾[1,3+], 张 昱[1,3], 陈意云[1,3], 李兆鹏[1,3], 华保健[2,3]

[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 软件学院,安徽 合肥 230026)

[3](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)

Pointer Logic Dealing with Uncertain Equality of Pointers

LIANG Hong-Jin[1,3+], ZHANG Yu[1,3], CHEN Yi-Yun[1,3], LI Zhao-Peng[1,3], HUA Bao-Jian[2,3]

[1](School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China)

[2](School of Software Engineering, University of Science and Technology of China, Hefei 230026, China)

[3](Software Security Lab., Suzhou Institute for Advanced Study, University of Science and Technology of China, Suzhou 215123, China)

+ Corresponding author: E-mail: lhj1018@mail.ustc.edu.cn, http://kyhcs.ustcsz.edu.cn

Liang HJ, Zhang Y, Chen YY, Li ZP, Hua BJ. Pointer logic dealing with uncertain equality of pointers. *Journal of Software*, 2010,21(2):334−343. http://www.jos.org.cn/1000-9825/3783.htm

Abstract: A pointer logic is designed for a C-like programming language PointerC. The pointer logic is an extension of Hoare logic, and it uses the idea of precise alias analysis in pointer program verification to support safety verification of programs in which equality of pointers is well-regulated. This paper presents an extension to the pointer logic by introducing a set of uncertain-equality pointer access path sets, so as to reason in the extended pointer logic about properties of programs which manipulate data structures like directed graph in which equality of pointers is uncertain.
Key words: software safety; Hoare logic; pointer logic

摘 要: 为类C小语言PointerC设计的指针逻辑是Hoare逻辑的一种扩展,可用来对指针程序进行精确的指针分析,以支持指针相等关系确定的程序的安全性验证.通过增加相等关系不确定的指针类型访问路径集合,可扩展这种指针逻辑,使得扩展后的指针逻辑可以应用于有向图等指针相等关系不确定的抽象数据结构上的指针程序性质证明.
关键词: 软件安全;Hoare逻辑;指针逻辑
中图法分类号: TP311 **文献标识码**: A

随着软件日益广泛的使用,软件的安全性也日益重要.形式程序验证技术为证明程序安全性提供了一种方法.Hoare逻辑是目前使用广泛的证明命令式语言程序性质的方法,但是Hoare逻辑在处理赋值语句时的代换只会影响断言中涉及的特定变元.虽然可以很容易地扩展它,使它能够确定一些较简单的别名关系,但它很难处理别名关系复杂的指针程序.

* Supported by the National Natural Science Foundation of China under Grant Nos.90718026, 60928004 (国家自然科学基金)
Received 2009-06-14; Revised 2009-09-11;Accepted 2009-12-07

© 中国科学院软件研究所 http://www.c-s-a.org.cn

分离逻辑[1]通过扩展Hoare逻辑,证明含有共享易变数据结构(shared mutable data structure,即这类结构中的域可以通过指针访问并更新)的命令式程序的性质.在分离逻辑中,引入分离合取断言$P*Q$,使得断言P和Q分别对堆中不相交的部分成立.分离逻辑在汇编语言级程序验证中得到了广泛的应用,但它把堆视为互不影响的若干部分,而不提供描述它们之间联系的机制.

为了研究指针程序的验证,我们设计了有动态内存分配和回收机制的类C小语言PointerC[2],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)来存取所指向的对象的操作,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[3−5].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.所收集的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他性质的验证.后来,我们又扩展了指针逻辑,允许有限制的指针算术运算,使得对一维动态数组操作的安全性验证成为可能[6].我们也完成了PointerC语言安全性和指针逻辑可靠性的证明[7],并开发了相应的出具证明编译器原型plcc[4,8].plcc生成携带证明的目标代码.

但是,我们原有的指针逻辑只能处理指针相等关系确定的数据结构,如链表、二叉树等,尚不能描述指针相等关系部分确定或完全不确定的数据结构.例如,有向图中的任意两个指针都可以指向同一个节点,但无法静态确定是哪些指针同时指向了该节点;而原有的指针逻辑只能精确地收集指针间相等与否的信息,却没有任何机制描述这种指针相等关系不确定的情况,所以无法支持诸如有向图这样的数据结构.另一方面,原有的指针逻辑总是需要收集精确的指针相等信息,但是在证明程序性质的过程中常常并不需要如此精确的信息.所以有必要放宽限制,允许所收集的指针信息中存在一部分不确定的指针相等信息.因此,我们引入相等关系不确定的一组访问路径集合,每个这样的集合内的指针都是相等的,但它们还可能与其他这样的集合中的指针相等.放宽限制带来的问题是,当某个这样的集合只含有 1 个元素p时,对p赋值可能会(但不是一定会)引起内存泄漏;当释放某个这样的集合中的指针所指向的对象时,不能保证做到把所有指向该对象的指针都标注为悬空指针.如果程序中不会出现这些情况,则原有的指针逻辑的规则都可以安全地使用.我们扩展原有的指针逻辑的目的是,当出现这些情况时给出警告甚至放弃继续验证,而在其他时候则能按照原先的方式进行验证.

本文基于上面的想法对指针逻辑作出扩展,主要贡献如下:

(1) 在原有的访问路径集合的基础上增加相等关系不确定的访问路径集合,并在断言语言中引入指针不从属断言的概念,使得指针逻辑可以描述指针相等关系不确定的数据结构.

(2) 首次将指针逻辑应用于非单一结构的实际程序验证中.实际程序所使用的数据结构常常是多种典型数据结构(如单向链表和双向链表)混合的复杂形态,将指针逻辑应用于这样的程序验证中展现了指针逻辑的潜力和实用性.

本文第 1 节介绍因引入指针相等关系部分不确定而引起的指针逻辑扩展.第 2 节给出采用指针逻辑进行程序验证的两个证明实例.第 3 节介绍相关工作.第 4 节是总结.

1 指针相等关系不确定的指针逻辑的设计

为了将指针相等关系不确定引入到指针逻辑中,需要引入相等关系不确定的访问路径集合,扩展断言语言及推理规则,扩展规范语言及推理规则.

1.1 PointerC语言及指针逻辑简介

PointerC 是一种强调指针类型的类 C 小语言,它使用以指针类型声明变量开始的访问路径表示指针.例如,*s->r->l* 是一条长度为 2 的访问路径,在 PointerC 中,指针只能用于赋值、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

© 中国科学院软件研究所 http://www.c-s-a.org.cn

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,PointerC 中允许布尔表达式,但不采用短路计算,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.

在原有的为PointerC语言设计的指针逻辑[5]中,使用指针集合表示内存状态.我们设计了 3 种指针类型访问路径的集合:表示 NULL 指针集合的$\mathcal{N}$、表示悬空指针集合的$\mathcal{D}$以及表示一组有效指针集合的Π .Π 中的每个集合表示在某程序点处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.

我们可以用指针逻辑所定义的访问路径集合实现数据结构的归纳定义.例如,二叉树可以如下定义:

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

其中,s是指向struct BTNode{struct BTNode *r,*l;}类型的指针,$\{s\}_N$表示空树.另一种情况$\{s\}\wedge tree(s\text{->}l)\wedge tree(s\text{->}r)$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,则代表有效指针的各访问路径都在Π的不同集合中.

1.2 相等关系不确定的访问路径集合

在图等数据结构中,指针相等与否并无规律,即有效指针的相等关系存在不确定的情况.原有的指针逻辑难以描述这种指针相等关系不确定的数据结构,因此,我们在原有的指针逻辑的基础上增加相等关系不确定的指针类型访问路径的一组集合,用$\mathcal{U}$表示.在语义模型上,$\mathcal{U}$中的每个集合表示在某程序点处一组指向同一个对象的指针,但是与Π 不同的是,$\mathcal{U}$中不同集合的指针可能指向同一对象.Π 和$\mathcal{U}$中各集合内的指针都称为有效指针.用Ψ作为$\mathcal{N}$,$\mathcal{D}$,Π ,$\mathcal{U}$的总称.注意,虽然我们称其为相等关系不确定的访问路径集合,但是所谓的不确定性仅存在于这种集合之间,而每个集合内的指针都是确定相等的.这种做法可以理解为放宽了原先对Π集合的要求.

并非任意一组指针类型访问路径经随意划分形成的Ψ都能表达上面的意思.因此,我们需要定义能够描述数据结构的合法的Ψ.引入$\mathcal{U}$集合后,需要在原有的合法Ψ的定义[5]的基础上,考虑$\mathcal{U}$中访问路径的特点,如$\mathcal{U}$中访问路径以Π 或$\mathcal{U}$中的访问路径为前缀等,以及$\mathcal{U}$中的集合与其他集合之间的关系,如Π 中访问路径的前缀不会在$\mathcal{U}$中等.

首先需要扩展访问路径的别名的定义,以处理增加的$\mathcal{U}$集合.出现在Π 或$\mathcal{U}$的同一个集合中的各访问路径加上同一后缀形成的访问路径互为别名.按此定义,判断两条访问路径是否互为别名的谓词如下:

$$alias(p,q)\triangleq p\equiv q\vee\exists s.\exists r.\exists t.(s\text{ 不是空串}\wedge r\text{ 和 }t\text{ 属于}\Pi\text{或}\mathcal{U}\text{中的同一集合}\wedge p\equiv(p'\cdot s)\wedge q\equiv(q'\cdot s)\wedge alias(p',r)\wedge alias(q',t)).$$

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 或$\mathcal{U}$中的同一个集合中.若递归计算时出现再次判断 p 和 q 是否为别名,则计算不终止,这时也认为 p 和 q 不是别名.该谓词和文献[5]中的谓词有不同的表现:由于$\mathcal{U}$中指针相等关系不确定,$\mathcal{U}$中不同集合的指针也可能互为别名.因此,该谓词可靠但不完备,不过它不影响下面的合法Ψ的定义.

扩展后的合法Ψ需要满足下面几个条件:

(1) 所有声明为指针类型的变量都在Ψ中.

(2) 对Π 中的任何访问路径 p,若 p 所指向的结构类型有指针类型域 r,则 $p\text{->}r$ 的某个别名在Ψ中.

(3) 对Π,$\mathcal{N}$和$\mathcal{D}$中的任何访问路径,Π,$\mathcal{N}$和$\mathcal{D}$中的任何其他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 或$\mathcal{U}$中.

(5) 对$\mathcal{U}$中的任何访问路径 p,若 p 所指向的结构类型有指针类型域 r,则 $p\text{->}r$ 的某个别名在$\mathcal{U}$,$\mathcal{N}$或$\mathcal{D}$中.

1.3 断言语言的扩展

我们在原有的指针逻辑[5]的断言语言基础上作出扩展,以支持引入$\mathcal{U}$集合.另外,还需要在允许描述两个指针相等或不相等关系基础上,增加表达$\mathcal{U}$中的一个指针不从属其他$\mathcal{U}$集合的断言,以表达已知的$\mathcal{U}$中不同集合的指针之间的不相等关系.

扩展后的断言语言如下:

© 中国科学院软件研究所 http://www.c-s-a.org.cn

$$assertion::=boolexp|\neg assertion|assertion\wedge assertion|assertion\vee assertion|\forall ident:domain.assertion|$$
$$\exists ident:domain.assertion|(assertion)|ident(lvalset)|\{lvalset\}|\{lvalset\}_N|\{lvalset\}_D|$$
$$[lvalset]|lval\notin ident(lval)|lval\notin[lvalset]$$
$$domain::=\mathrm{N}|exp..exp$$
$$lvalset::=lvalset,lval|lval$$
$$lval::=ident|lval\text{->}ident|lval(\text{->}ident)^{exp}$$

其中,*boolexp* 是 PointerC 语法中的布尔表达式,*lval* 是指针类型访问路径.注意{*lvalset*}和新增加的[*lvalset*]的区别,前者表示Π的一个集合,后者表示$\mathcal{U}$的一个集合,带下标 *N* 或 *D* 的仍然分别表示$\mathcal{N}$集合和$\mathcal{D}$集合.新增加的 $lval\notin ident(lvalset)$和 $lval\notin[lvalset]$是指针不从属断言.其中,*ident*(*lvalset*)通常是数据结构形状谓词,如前面定义的二叉树 *tree*(*s*)等,可以根据定义加以展开.$p\notin ident(lvalset)$的含义是,将 *ident*(*lvalset*)展开得到的任意一个访问路径集合都不含有 *p* 的别名.$p\notin[lvalset]$的含义是,$\mathcal{U}$集合[*lvalset*]不含有 *p* 的别名.

直观上,指针不从属断言是对$\mathcal{U}$集合的一种限制,它可以部分消除指针关系的不确定性,是沟通$\mathcal{U}$集合和Π集合的桥梁.例如,二叉有向无环图的定义为

$$dag(p)\triangleq\{p\}_N\vee([p]\wedge dag(p\text{->}l)\wedge dag(p\text{->}r)\wedge p\notin dag(p\text{->}l)\wedge p\notin dag(p\text{->}r)),$$

其中,$\{p\}_N$表示空图.另一种情况$[p]\wedge dag(p\text{->}l)\wedge dag(p\text{->}r)\wedge p\notin dag(p\text{->}l)\wedge p\notin dag(p\text{->}r)$表示非空图,并且表达出*p*可能与其他指针相等,但却不会与*p*的左右子图内的指针相等.$p\notin dag(p\text{->}l)$是指*p*的别名不在归纳定义$dag(p\text{->}l)$所展开的有效指针集合中.当$dag(p\text{->}l)$不是空图时,有

$$p\notin dag(p\text{->}l)=p\notin[p\text{->}l]\wedge p\notin dag(p\text{->}l\text{->}l)\wedge p\notin dag(p\text{->}l\text{->}r).$$

指针不从属断言之前不允许使用¬,因为常用数据结构的定义中没有这个需求.

1.4 断言演算的扩展

在原有的指针逻辑[5]中,设计了一些与Ψ有关的逻辑等价或蕴涵公理,包括:1) $\mathcal{N}$和$\mathcal{D}$的等价公理:两个$\mathcal{N}$集合(或$\mathcal{D}$集合)与合并后的$\mathcal{N}$集合(或$\mathcal{D}$集合)等价;2) 非合法Ψ公理:含有非合法Ψ的断言为假;3) Ψ等价公理:当两个合法Ψ的对应访问路径集合等价时,合法Ψ等价;4) Ψ对布尔表达式的吸收或否定:若断言中的布尔表达式 *p*==NULL,*p*!=NULL,*p*==*q* 和 *p*!=*q*(*p* 和 *q* 都是指针类型)等和Ψ无矛盾则被吸收,有矛盾则整个断言(包括布尔表达式和Ψ)为假.

在增加了$\mathcal{U}$集合和指针不从属断言后,这些公理也需要进行相应扩展,如对原先的 4)部分:

4) Ψ对布尔表达式的吸收或否定

用$\mathcal{S}_1,\ldots,\mathcal{S}_n$表示构成$\Pi$的*n*个访问路径集合,用$\mathcal{T}_1,\ldots,\mathcal{T}_m$表示构成$\mathcal{U}$的*m*个访问路径集合,用*Q*作为除了$\Psi$以外的其他断言的总称.记号$\mathcal{S}^p$表示访问路径*p*的别名所在的集合,$\mathcal{S}^{p,q}$,$\mathcal{T}^p$,$\mathcal{T}^{p,q}$的含义类似.注意到,在这部分规则中,$\mathcal{U}$集合与$\Pi$集合的地位是平等的.所以,原来关于$\Pi$集合的规则[5]可以几乎原样复制为关于$\mathcal{U}$集合的规则.下面列举一些这样的蕴涵公理:

$$\mathcal{T}_1\wedge\ldots\wedge\mathcal{T}_{m-1}\wedge\mathcal{T}^p\wedge\Pi\wedge\mathcal{N}\wedge\mathcal{D}\wedge p{=}{=}\mathrm{NULL}\wedge Q\Rightarrow\mathrm{false}$$
$$\mathcal{T}_1\wedge\ldots\wedge\mathcal{T}_m\wedge\Pi\wedge\mathcal{N}^p\wedge\mathcal{D}\wedge p{=}{=}\mathrm{NULL}\wedge Q\Rightarrow\mathcal{T}_1\wedge\ldots\wedge\mathcal{T}_m\wedge\Pi\wedge\mathcal{N}^p\wedge\mathcal{D}\wedge Q$$
$$\mathcal{T}_1\wedge\ldots\wedge\mathcal{T}_{m-1}\wedge\mathcal{T}^p\wedge\Pi\wedge\mathcal{N}\wedge\mathcal{D}\wedge p!{=}\mathrm{NULL}\wedge Q\Rightarrow\mathcal{T}_1\wedge\ldots\wedge\mathcal{T}_{m-1}\wedge\mathcal{T}^p\wedge\Pi\wedge\mathcal{N}\wedge\mathcal{D}\wedge Q$$
$$\mathcal{T}_1\wedge\ldots\wedge\mathcal{T}_{m-1}\wedge\mathcal{T}^{p,q}\wedge\Pi\wedge\mathcal{N}\wedge\mathcal{D}\wedge p{=}{=}q\wedge Q\Rightarrow\mathcal{T}_1\wedge\ldots\wedge\mathcal{T}_{m-1}\wedge\mathcal{T}^{p,q}\wedge\Pi\wedge\mathcal{N}\wedge\mathcal{D}\wedge Q$$
$$\mathcal{T}_1\wedge\ldots\wedge\mathcal{T}_{m-1}\wedge\mathcal{T}^{p,q}\wedge\Pi\wedge\mathcal{N}\wedge\mathcal{D}\wedge p!{=}q\wedge Q\Rightarrow\mathrm{false}$$

此外,针对 *p* 的别名与 *q* 的别名分属于Π的集合与$\mathcal{U}$的集合的情况,需要增加了两条规则:

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

$\mathcal{U}$集合和不从属断言的特殊性导致还需要为它们增加一些公理.与 *p* 的别名和 *q* 的别名属于不同的Π集合的情形不同,当 *p* 的别名和 *q* 的别名属于不同的$\mathcal{U}$集合时,断言 *p*==*q* 引起它们分处的两个$\mathcal{U}$集合合并,而这个合

© 中国科学院软件研究所 http://www.c-s-a.org.cn

并会引起以 p 的别名和 q 的别名为前缀的访问路径所在集合的继续合并;断言 $p!=q$ 则保证了它们分处的两个$\mathcal{U}$集合的指针一定不相等,可转化为用不从属断言表示.此外,还需要为不从属断言设计专门的等价或蕴涵公理.下面是其中部分新增公理.

5) $\mathcal{U}$集合合并公理

当断言$p==q$为真时,引起集合$\mathcal{T}^p$和$\mathcal{T}^q$合并,包括删除合并后集合中的别名,蕴涵公理如下:

$$\mathcal{T}_1\wedge\ldots\wedge\mathcal{T}_{m-2}\wedge\mathcal{T}^p\wedge\mathcal{T}^q\wedge\Pi\wedge\mathcal{N}\wedge\mathcal{D}\wedge p==q\wedge Q\Rightarrow$$

$$\mathcal{T}_1\wedge\ldots\wedge\mathcal{T}_{m-2k-2}\wedge(\mathcal{T}^p\uplus\mathcal{T}^q)\wedge\mathcal{T}^{p->r1}\wedge\mathcal{T}^{q->r1}\wedge\ldots\wedge\mathcal{T}^{p->rk}\wedge\mathcal{T}^{q->rk}\wedge\Pi\wedge\mathcal{N}\wedge\mathcal{D}\wedge p->r1==q->r1\wedge\ldots\wedge p->rk==q->rk\wedge Q$$

其中,$\uplus$表示合并两个集合并删除其中的别名,$r1,\ldots,rk$是p所指向对象的指针类型域名.从这里看出,还需要继续合并$\mathcal{T}^{p->ri}$和$\mathcal{T}^{q->ri}$($i=1,\ldots,k$).该合并过程持续到相等的指针出现在$\mathcal{D}$集合中或$\mathcal{N}$集合中为止.限于篇幅,相应规则略去.

6) 指针不从属断言的等价公理

$$p\notin(assertion1\wedge assertion2)\Leftrightarrow p\notin assertion1\wedge p\notin assertion2$$

$$p\notin(\forall ident:domain.assertion)\Leftrightarrow\forall ident:domain.(p\notin assertion)$$

$$p\notin\mathcal{T}^q\Leftrightarrow q\notin\mathcal{T}^p\Leftrightarrow p!=q$$

7) 指针不从属断言对布尔表达式的吸收或否定

$$\mathcal{T}_1\wedge\ldots\wedge\mathcal{T}_{m-1}\wedge\mathcal{T}^q\wedge\Pi\wedge\mathcal{N}\wedge\mathcal{D}\wedge p==q\wedge p\notin\mathcal{T}^q\wedge Q\Rightarrow\text{false}$$

$$\mathcal{T}_1\wedge\ldots\wedge\mathcal{T}_{m-1}\wedge\mathcal{T}^q\wedge\Pi\wedge\mathcal{N}\wedge\mathcal{D}\wedge p!=q\wedge p\notin\mathcal{T}^q\wedge Q\Rightarrow\mathcal{T}_1\wedge\ldots\wedge\mathcal{T}_{m-1}\wedge\mathcal{T}^q\wedge\Pi\wedge\mathcal{N}\wedge\mathcal{D}\wedge p\notin\mathcal{T}^q\wedge Q$$

当p被限定为$\mathcal{U}$中的某个有效指针时,还可以给出$p\notin\{lvalset\}$,$p\notin\{lvalset\}_N$和$p\notin\{lvalset\}_D$等化简至true或false的蕴涵公理.

1.5 指针逻辑的规范和推理规则的扩展

我们使用一种 Hoare 风格的程序规范$\{P\}S\{Q\}$,其中:S 是程序片段,通常是语句;P 和 Q 分别是它的前、后条件.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性质,同时把程序分析、类型检查和程序验证联系起来,有更好的表达能力.

在原有的指针逻辑[5]中,我们定义了一些基本运算以表达指针逻辑的推理规则.如删除访问路径"–"、增加访问路径"+"、前缀替换"/"、测试内存泄漏 leak($\mathcal{S}$,p).其中,前缀替换的含义是: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 q 的同一个别名 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.下面所列规则中仍然使用这些基本运算.

引入相等关系不确定的访问路径集合后,对内存泄漏的判定引入了不确定性:当访问路径 p 的别名在Π 中集合$\mathcal{S}$内,删除 p 的别名时(出现在对 p 赋值时),若存在某个集合,其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;但当 p 的别名在$\mathcal{U}$中集合$\mathcal{T}$内,删除 p 的别名时,若存在某个集合,其中所有访问路径都是 p 的别名或以 p 的别名为前缀,则可能会出现内存泄漏,但不是一定会出现,因为可能存在其他$\mathcal{U}$集合的指针与 p 指向同一个对象.

下面给出处理指针相等关系不确定的指针逻辑在原先的推理规则[5]上的扩展.除了某些情况下难以判断是否有内存泄漏和难以标注悬空指针以外,这些推理规则与原来的指针逻辑的推理规则类似,因为$\mathcal{U}$的每个集合与Π的每个集合具有相似的特征,只是$\mathcal{U}$的集合之间存在不确定的指针相等关系.

1) 指针类型赋值语句 $p=q$ 和 p=NULL

对于绝大部分情况,原来关于Π 集合的规则[5]可以几乎原样复制为关于$\mathcal{U}$集合的规则.但是对于以下两种情况,由于对内存泄漏的判定出现了不确定性,所以需要特别处理:.

(1) p 的别名在Π 或$\mathcal{U}$的某个集合中,q 的别名在$\mathcal{N}$中.

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

即对于赋值前的Ψ,$leak(\mathcal{S}^p,p)$为假.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

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

对于赋值前的Ψ,测试$leak(\mathcal{S}^p,p)$.

注意到第 1 条规则要求赋值前$leak(\mathcal{S}^p,p)$为假;而第 2 条规则只是在赋值前测试$leak(\mathcal{S}^p,p)$,如果$leak(\mathcal{S}^p,p)$为真,则给出可能内存泄漏的警告信息.

(2) p 的别名在Π 或$\mathcal{U}$的某个集合中,q 的别名在$\mathcal{D}$中的情况类似.

2) 非指针类型的赋值语句、分配空间语句及复合、条件、循环语句的规则

仍然使用原来指针逻辑的规则[5].

3) 释放空间语句 free(p)

只有p的别名在Π 或$\mathcal{U}$的某个集合中,才能使用free函数.当p的别名在Π 的某个集合中时,free(p)的规则与原先的指针逻辑[5]一致.若p的别名在$\mathcal{U}$的某个集合中,那么所用规则虽然相同,但是因为可能存在其他$\mathcal{U}$集合的指针与 p 指向同一个对象,所以必须给出可能出现悬空指针的警告信息.

4) 分情况规则、结构规则和函数调用规则

仍然使用原来指针逻辑的规则[5].

2 证明实例

2.1 Schorr-Waite算法

Schorr-Waite算法[9]是一种非递归的有向图遍历算法,它从根节点开始深度优先遍历有向图的所有节点,但仅使用图本身的指针实现回溯.该算法的基本思想在于,由于遍历过程中总是有临时指针指向当前节点的前驱,使得从前驱指向下一个节点的指针是冗余的,可以利用它指向前驱节点的前驱,直至图的根节点.这种指针的翻转链(reverse pointer chain)实现了图的回溯,相当于一个栈.

我们使用指针逻辑证明 Schorr-Waite 算法满足 PointerC 的基本安全策略,即程序运行时不出现通过 NULL 指针或悬空指针来存取所指向的对象的操作,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.本例没有证明该算法的正确性(即该算法遍历图上所有节点),因为指针逻辑对这个问题的证明没有特别的优势.

使用该算法的 C 版本,与文献[10]中的做法一样.我们修改了循环条件和 if 条件中的布尔表达式,将布尔条件显式化,以便正确反映到代码的前后条件上,但为了避免过多地修改程序,假定程序中的布尔表达式仍按短路计算.其节点类型定义如下:

```
typedef struct node {
        boolean m, c;
        struct node *l, *r;
} node;
```

其中,l 和 r 分别是指向左子节点和右子节点的指针(可为 NULL),m 是节点的遍历标志,c 用于区分左、右子节点哪个已访问过,是算法内部使用的.

使用扩展的指针逻辑描述这种二叉有向图结构:

$$graph(p)\triangleq\{p\}_N\vee([p]\wedge graph(p\text{->}l)\wedge graph(p\text{->}r)),$$

其中,$\{p\}_N$表示空图.另一种情况$[p]\wedge graph(p\text{->}l)\wedge graph(p\text{->}r)$表示非空图.注意,该定义与二叉树和二叉有向无环图定义的区别.若仅依据该定义,一个带环有向图的展开式将是无限的,但是本例用节点的标志信息m和c来识别环,因此该定义在此是恰当的.

下面以 Schorr-Waite 图遍历算法的安全性证明为例,展示扩展的指针逻辑在指针相等关系不确定的数据结构中的应用.限于篇幅,只给出主要程序点的断言.

$\{graph(root)\}$

© 中国科学院软件研究所 http://www.c-s-a.org.cn
www.jos.org.cn

```
void schorr_waite (node *root){
    node *t, *p, *q;
    {graph(root)∧{t,p,q}_D}
    t=root; p=NULL; q=NULL; root=NULL;
    {graph(t)∧{root,p,q}_N}
    {graph(t)∧graph(p)∧{root,q}_N}    /*循环不变式*/
    while (p!=NULL∨(p==NULL∧t!=NULL∧¬t->m)){
        {(graph(t)∧graph(p->l)∧graph(p->r)∧[p]∧{root,q}_N)∨
         (graph(t->l)∧graph(t->r)∧[t]∧{p,root,q}_N∧¬t->m)}
        if (t==NULL∨(t!=NULL∧t->m)){
            {(graph(p->l)∧graph(p->r)∧[p]∧{t,root,q}_N)∨
             (graph(p->l)∧graph(p->r)∧[p]∧graph(t->l)∧graph(t->r)∧[t]∧{root,q}_N)}
            if (p->c){    /* pop */
                q=t; t=p; p=p->r; t->r=q; q=NULL;
                {(graph(t->l)∧[t]∧graph(p)∧{root,q,t->r}_N)∨
                (graph(t->l)∧[t]∧graph(p)∧graph(t->r->l)∧graph(t->r->r)∧[t->r]∧{root,q}_N)}
            }
            else {    /* swing */
                q=t; t=p->r; p->r=p->l; p->l=q; p->c=true; q=NULL;
                {(graph(p->r)∧graph(t)∧[p]∧{root,q,p->l}_N)∨
                 (graph(p->r)∧graph(t)∧[p]∧graph(p->l->l)∧graph(p->l->r)∧[p->l]∧{root,q}_N)}
            }
            {graph(t)∧graph(p)∧{root,q}_N}    /*循环不变式*/
        }
        else {    /* push, t!=NULL∧¬t->m */
            {(graph(p->l)∧graph(p->r)∧[p]∧graph(t->l)∧graph(t->r)∧[t]∧{root,q}_N∧¬t->m)∨
             (graph(t->l)∧graph(t->r)∧[t]∧{p,root,q}_N∧¬t->m)}
            q=p; p=t; t=t->l; p->l=q; p->m=true; p->c=false; q=NULL;
            {(graph(p->l->l)∧graph(p->l->r)∧[p->l]∧graph(t)∧graph(p->r)∧[p]∧{root,q}_N)∨
             (graph(t)∧graph(p->r)∧[p]∧{root,q,p->l}_N)}
            {graph(t)∧graph(p)∧{root,q}_N}    /*循环不变式*/
        }
    }    /* p==NULL∧(t==NULL∨(t!=NULL∧t->m)) */
    {graph(t)∧{root,p,q}_N}
}
```

2.2 Glibc中异步I/O删除请求函数

在以往的工作中,我们仅使用指针逻辑完成了典型数据结构(如单链表、二叉树等)上的典型操作(如插入节点等)的验证,尚未真正验证实际程序中使用共享易变数据结构的例子.通过调查实际程序发现,虽然程序中使用一般图和有向无环图时都是用邻接矩阵等方式来表示这样的数据结构,似乎指针逻辑该扩展的用处不大,但是在实际使用链表、二叉树的程序中,常常会针对特定问题设计特殊的数据结构以及其上的实现特定功能的操作,它们体现出对指针逻辑扩展的需求.本节以GNU C Library[11]中的一段源码为例,展示指针逻辑在实际中的应用.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

在 GNU C Library 中,使用请求队列实现异步 I/O 操作.请求队列中的每个节点保存对某文件描述符的 I/O 请求.同一文件描述符可以有多个 I/O 请求,这些请求对应的节点按优先级排序形成单链表(指针域为 *next_prio*);不同文件描述符的请求对应不同的单链表.这些单链表的首节点按照对应的文件描述符排序形成一个双向链表(指针域为 *last_fd*,*next_fd*).该双向链表以及链在该双向链表上的所有单链表构成请求队列,它们之间的指针相等关系都是精确的.针对请求队列中的所有节点,那些处于就绪状态的节点(必在请求队列的双向链表中)又链成一个单向链(指针域为 *next_run*),称为就绪队列,导致表示就绪队列的单链表和整个请求队列的双向链表的指针相等关系部分不确定.如图 1 所示,*s* 和 *r* 分别是请求队列和就绪队列的头指针.

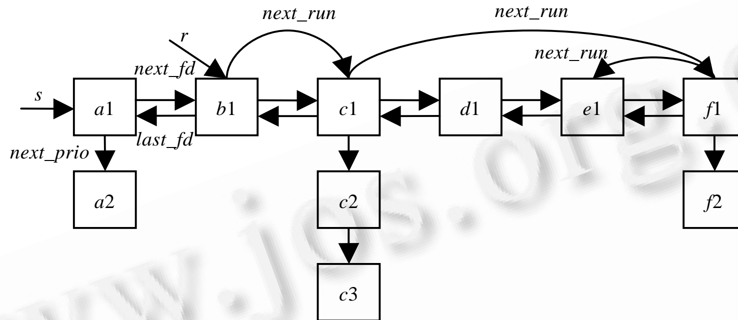

Fig.1 Request queue and ready queue

图 1 请求队列与就绪队列

删除请求函数 void __aio_remove_request(struct requestlist **last*, struct requestlist **req*, int *all*)在所有链关系(包括请求队列和就绪队列)中删除 *req* 节点.参数 *last* 是 *req* 所在的单链表上 *req* 的前驱节点指针,当 *req* 指向的节点在双向链表上时,*last* 为 NULL.参数 *all* 指示删除节点的方式,当 *all*=1 时,*req* 节点所链的部分都将随 *req* 一起移走;当 *all*=0 时,仅删除 *req* 节点,*req* 节点所链的部分将取代 *req* 的位置.

在验证该函数的性质时,由于请求队列和就绪队列涉及不确定的指针相等关系,不容易准确描述这种结构的对象以及在函数每一步操作后它的变化;另一方面,该函数代码十分简练却涵盖了多种情况,使得指针逻辑断言分情况较多,手工证明十分繁琐.因此,使用指针逻辑验证实际程序有待良好工具的支持.限于篇幅,比较详细的介绍和证明见文献[12].

3 相关工作

本文扩展的指针逻辑可以用来描述图等指针相等关系不确定的数据结构.指针逻辑本质上是一种精确的指针分析的方法,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.

Bornat也使用Hoare风格的逻辑来推导指针程序的性质[13].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.在数据结构的归纳定义方面, Bornat 通过归纳定义不同的操作符来描述不同的数据结构,例如有向图的定义:

$$A_{l,r}^{*} \triangleq \text{if } A=nil \text{ then } \{\} \text{ else } \{A\}\cup A.\,l_{l,r}^{*} \cup A.\,r_{l,r}^{*} \text{ fi.}$$

而有向无环图的定义则是通过在有向图定义的基础上,引入一个“exclusion set”破坏环结构:

$$A_{l,r,S}^{*} \triangleq \text{if } A=nil \vee A\in S \text{ then } \{\} \text{ else } \{A\}\cup A.\,l_{l,r,S\cup\{A\}}^{*} \cup A.\,r_{l,r,S\cup\{A\}}^{*} \text{ fi.}$$

我们扩展的指针逻辑则通过引入指针不从属断言来破坏环结构.在 Bornat 的描述方法中,节点的共享或分离关系并不明确,只是假定除了有向图外,其他数据结构都是无环的,这样很容易使树与图的定义产生混淆.另一方面,节点共享或分离关系不明确,也容易造成在对特定节点赋值时无意义地展开其他无关的结构.因此,

© 中国科学院软件研究所 http://www.c-s-a.org.cn

Bornat 引入了“spatial-separation assertions”表达对象或节点间的分离关系.我们扩展的指针逻辑虽然称作处理指针相等关系不确定的结构,但这种不确定性是有限制的,至少在同一个访问路径集合内的指针都是确定相等的;并且,当对象的指针信息完整时,我们的指针逻辑可以精确地描述它,而不会产生二义性.

分离逻辑也可以描述有向图等结构共享不明确的数据结构.由于分离逻辑强调分离堆之间互无影响的特点,所以在描述有向无环图时,分离逻辑可以清晰地表达无环的特征.但分离逻辑不提供描述分离堆之间联系的机制,而有向图中的别名关系错综复杂,使得难以用分离逻辑表达访问路径间的相等关系,即结构的共享关系.文献[14]中使用一种部分图(partial graph,即其允许出现不指明后继的节点)收集有向图的结构共享信息,仅明确标记并展开共享结构的第一次出现,其他出现则标记为第一次出现的引用而不再具体展开.这种做法利用部分图的概念克服了分离逻辑在表达共享关系上的困难,但是却需要太多的关于环境(environment)的信息.即便是证明很简单的例子,如copy DAGs,也需要很大的工作量[14].

在证明程序的安全性方面,由于指针逻辑将程序的安全性描述为定型规则的副条件,所以对程序的安全性验证及不安全因素的检测都是内在的.例如,用指针逻辑证明Schorr-Waite算法的安全性时非常简洁,没有明显的表述安全性的断言.Yang在使用分离逻辑证明Schorr-Waite算法的正确性时[15]需要引入*noDangling*(x)来描述x不是悬空指针的特性.在使用原型工具CADUCEUS证明Schorr-Waite算法的正确性时[10]引入了*valid*(x)表达类似的含义.

另外,无论是Bornat对Hoare逻辑的扩展还是分离逻辑,目前都只用于一些比较简单的例子,如Schorr-Waite算法、CopyDags函数等,而没有用于如第2.2节所述的较为复杂的实际程序.虽然该例是手工证明的,但我们将进一步开发plcc,以支持较为复杂程序的自动验证.

4 结束语

本文扩展了为 PointerC 语言设计的可对指针程序进行精确分析的指针逻辑,使其可以描述图等指针相等与否不确定的数据结构,以支持这种数据结构上的指针程序的性质证明.扩展的处理指针相等关系不确定的指针逻辑的推理规则都是应实际需求而提出的,因此可能并不完善.

到目前为止,除了需要工具支持以外,使用指针逻辑进行程序验证的另一个局限是需要程序员提供程序的前、后断言和循环不变式,这些都限制了本文方法的应用.为了尽量减轻程序员的负担,我们将尝试仅让程序员声明指针变量指向数据结构的种类(单链表、二叉树等)并提供函数的前、后断言,然后自动推导函数代码中的循环不变式.

References:

[1] Reynolds JC. Separation logic: A logic for shared mutable data structures. In: Proc. of the 17th Annual IEEE Symp. on Logic in Computer Science. Copenhagen: IEEE Computer Society Press, 2002. 55−74.

[2] Hua BJ, Chen YY, Ge L, Wang ZF. The PointerC programming language specification. Technical Report. http://kyhcs.ustcsz.edu.cn/old/lss/doc/

[3] Chen YY, Hua BJ, Ge L, Wang ZF. A pointer logic for safety verification of pointer programs. Chinese Journal of Computers, 2008,31(3):372−380 (in Chinese with English abstract).

[4] Chen YY, Ge L, Hua BJ, Li ZP, Liu C, Wang ZF. A pointer logic and certifying compiler. Frontiers of Computer Science in China, 2007,1(3):297−312.

[5] Chen YY, Li ZP, Wang ZF, Hua BJ. A pointer logic for verification of pointer programs. Technical Report, 2010. http://kyhcs.ustcsz.edu.cn/lss

[6] Wang ZF, Chen YY, Wang ZM, Wang W, Tian B. An extension to pointer logic for verification. In: Proc. of the 2nd IEEE/IFIP Int’l Symp. on Theoretical Aspects of Software Engineering. Nanjing: IEEE Computer Society Press, 2008. 49−56.

[7] Hua BJ, Chen YY, Li ZP, Wang ZF, Ge L. Design and proof of a safe programming language PointerC. Chinese Journal of Computers, 2008,31(4):556−564 (in Chinese with English abstract).

© 中国科学院软件研究所 http://www.c-s-a.org.cn

[8] Chen YY, Ge L, Hua BJ, Li ZP, Liu C. Design of a certifying compiler supporting proof of program safety. In: Proc. of the 1st IEEE/IFIP Int'l Symp. on Theoretical Aspects of Software Engineering. Shanghai: IEEE Computer Society Press, 2007. 127−136.

[9] Schorr H, Waite WM. An efficient machine independent procedure for garbage collection in various list structures. Communications of the ACM, 1967,10(8):501−506.

[10] Hubert T, Marché C. A case study of C source code verification: The Schorr-Waite algorithm. In: Aichernig BK, Beckert B, eds. Proc. of the 3rd IEEE Int'l Conf. on Software Engineering and Formal Methods. Koblenz: IEEE Computer Society Press, 2005. 190−199.

[11] The Glibc contributors. http://www.gnu.org/software/libc/

[12] Liang HJ, Zhang Y, Chen YY, Li ZP, Hua BJ. An example of reasoning in extended pointer logic: AIO remove request function in Glibc. Technical Report. http://home.ustc.edu.cn/~lhj1018/

[13] Bornat R. Proving pointer programs in Hoare logic. In: Backhouse RC, Oliveira JN, eds. Proc. of the 5th Int'l Conf. on Mathematics of Program Construction. LNCS 1837, Ponte de Lima: Springer-Verlag, 2000. 102−126.

[14] Bornat R, Calcagno C, O'Hearn PW. Local reasoning, separation and aliasing. In: Proc. of the 2nd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management. 2004.

[15] Yang H. An example of local reasoning in BI pointer logic: The Schorr-Waite graph marking algorithm. In: Henglein F, Hughes J, Makholm H, Niss H, eds. Proc. of the 1st Workshop on Semantics, Program Analysis and Computing Environments for Memory Management. London: IT University of Copenhagen, 2001. 41−68.

附中文参考文献:

[3] 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.

[7] 华保健,陈意云,李兆鹏,王志芳,葛琳.安全语言 PointerC 的设计及形式证明.计算机学报,2008,31(4):556−564.

梁红瑾(1989－),女,江苏徐州人,硕士生,主要研究领域为程序验证,软件安全.

李兆鹏(1978－),男,博士,CCF 会员,主要研究领域为程序验证,软件安全,类型系统和理论.

张昱(1972－),女,博士,副教授,CCF 会员,主要研究领域为程序设计语言理论与实现.

华保健(1979－),男,博士,CCF 学生会员,主要研究领域为程序设计语言的设计与实现,软件安全.

陈意云(1946－),男,教授,博士生导师,CCF 高级会员,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.

© 中国科学院软件研究所 http://www.c-s-a.org.cn